The 2011 International Planning Competition

Description of Participating Planners

Deterministic Track

The Seventh International Planning Competition

Description of Participant Planners
of the Deterministic Track

June, 2011

Ángel García-Olaya agolaya@inf.uc3m.es
Sergio Jiménez sjimenez@inf.uc3m.es
Carlos Linares López carlos.linares@uc3m.es

Planning and Learning Group (PLG)
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911 - Leganés (Madrid)
Spain

Preface

This booklet summarizes the participants on the Deterministic Track of the International Planning Competition (IPC) 2011. Papers describing all the participating planners are included.

After a 3 years gap, the 2011 edition of the IPC involved a total of 55 planners, some of them versions of the same planner, distributed among four tracks: the sequential satisficing track (27 planners submitted out of 38 registered), the sequential multi-core track (8 planners submitted out of 12 registered), the sequential optimal track (12 planners submitted out of 24 registered) and the temporal satisficing track (8 planners submitted out of 14 registered). Three more tracks were open to participation: temporal optimal, preferences satisficing and preferences optimal. Unfortunately the number of submitted planners did not allow these tracks to be finally included in the competition.

A total of 55 people were participating, grouped in 31 teams. Participants came from Australia, Canada, China, France, Germany, India, Israel, Italy, Spain, UK and USA.

For the sequential tracks 14 domains, with 20 problems each, were selected, while the temporal one had 12 domains, also with 20 problems each. Both new and past domains were included. As in previous competitions, domains and problems were unknown for participants and all the experimentation was carried out by the organizers. To run the competition a cluster of eleven 64-bits computers (Intel XEON 2.93 Ghz Quad core processor) using Linux was set up. Up to 1800 seconds, 6 GB of RAM memory and 750 GB of hard disk were available for each planner to solve a problem. This resulted in 7540 computing hours (about 315 days), plus a high number of hours devoted to preliminary experimentation with new domains, reruns and bugs fixing.

The detailed results of the competition, the software used for automating most tasks, the source code of all the participating planners and the description of domains and problems can be found at the competition's web page:

`http://www.plg.inf.uc3m.es/ipc2011-deterministic`

Leganés, Spain, June 2011

Angel García-Olaya, Sergio Jiménez and Carlos Linares López,
The IPC 2011 chairs

Acknowledgement

The Deterministic Part of the Seventh International Planning Competition would have not been the same without the collaboration of a long list of people. Therefore we wanted to include this section in appreciation of all the assistance provided by them.

First of all, we want to express our most sincere thanks to all the people who submitted a planner to the Competition.

Also, to all of you who suggested a domain to be included in the tracks, even if some were not accepted in the end: **Bharatranjan Kavuluri** suggested the *matchcellar* domain; **Frédéric Maris** sent us the *cooking* and *temporal machine shop* domains; **Jörg Hoffmann** and **Hootan Nakhost** devised the *nomystery* specification; **Ron Petrick** submitted the domain *crisp*; **Héctor Geffner** and **Nir Lipovetzky** coded the *visitall* domain; **Amanda Coles** and **Andrew Coles** sent us the *market* domain; **Héctor Luis Palacios** prepared various conformant domains such as *1-dispose*, *dispose*, *grid*, *look-and-grab*, *push-to* and *classical*; **Bhaskara Marthi** is behind the *tidybot* domain; **Guy Shani** shall be acknowledged for sending the domains *colorballs doors*, *medicalPKS150*, *medicalPKS199*, *sliding-doors*, *unix* and *wumpus*. At last, but not least, **Tomás de la Rosa** worked in the *floortile* domain.

We do also want to explicitly express our gratitude to **Derek Long** for his assistance and support with the automated validation tool VAL.

Also, to **Daniel L. Kovacs** for making available through the wiki site of the Competition a couple of manuscripts with a formal specification of PDDL 3.1.

Besides, we do feel in debt with **Óscar Pérez**, **Jaime Pons**, **Roberto Fuentes** and, in general, to the Laboratory of the Computer Science Department of the University Carlos III de Madrid for their assistance in installing, configuring and maintaining the cluster.

Very importantly as well, to the IPC council for providing extensive comments and offering a lot of helpful suggestions. In particular, to **Malte Helmert** for inviting us to his university, his assistance with so much insight and all the material produced at the previous IPC.

To all the members of the Planning and Learning research group (PLG) of the University Carlos III de Madrid for their encouragement and, in so many cases, for being in charge of our daily duties to allow us to work in the Competition. In particular, to **Daniel Borrajo** for all the support.

Finally, we have to acknowledge the sponsorship of **Decide Soluciones**, **iActive**, the **University Carlos III de Madrid** and **ICAPS**. The hardware platform used during the competition was funded by Spanish Science Ministry under project MICIIN TIN2008-06701-C03-03.

Leganés, Spain, June 2011

Angel García-Olaya, Sergio Jiménez and Carlos Linares López,
The IPC 2011 chairs

Index

1. Sequential satisficing track

2. Sequential optimal track

3. Sequential multicore track

4. Temporal satisficing track

The ACOPlan Planner

Marco Baioletti, Alfredo Milani, Valentina Poggioni, Fabio Rossi
Università degli Studi di Perugia
Dip. di Matematica e Informatica
Perugia Italy

Abstract

ACOPlan is a planner based on the ant colony optimization framework, in which a colony of planning ants searches for near optimal solution plans with respect to an overall plan cost metric. This approach is motivated by the strong similarity between the process used by artificial ants to build solutions and the methods used by state–based planners to search solution plans. Planning ants perform a stochastic and heuristic based search by interacting through a pheromone model.

Introduction

The main idea underlying ACOPlan system is to use the well known *Ant Colony Optimization* metaheuristic (*ACO*) (Dorigo and Stuetzle 2004) to solve planning problems with the aim of optimizing the quality of the solution plans. The approach is based on the strong similarity between the process used by artificial ants to build solutions and the way used by state–based planners to find solution plans. Therefore, we have defined an ACO algorithm which handles a colony of planning ants with the purpose of solving planning problems by optimizing solution plans with respect to the overall plan cost. According to the main features of ACO the ant–planners of the colony are stochastic and heuristic–based.

ACO is a metaheuristic inspired by the behavior of natural ants colony which has been successfully applied to many *Combinatorial Optimization* problems. Being ACO a stochastic incomplete algorithm, there is no guarantee that optimal solutions are ever found, but a number of successful applications confirm ACO as a generally promising metaheuristic to find efficiently and effectively good suboptimal solutions.

In this paper the *ACOPlan* system for planning with action costs is briefly described. For a more detailed paper refer to (Baioletti et al. 2010) where an experimental evaluation and some comparisons with other state-of-art planners are presented.

In the next sections a brief introduction to the metaheuristic ACO, the ACOPlan model and the ACOPlan algorithm in the framework of planning with non uniform action costs are described.

Ant Colony Optimization

ACO is a well–known metaheuristic to tackle Combinatorial Optimization problems introduced since early 90s by Dorigo et Al. (Dorigo and Stuetzle 2004). It is inspired by the foraging behavior of natural ant colonies. When walking, natural ants leave on the ground a chemical substance called *pheromone* that other ants can smell. This stigmergic mechanism implements an "indirect communication way" among ants, in particular when looking for the shortest path to reach food.

ACO is usually applied to optimization problems whose solutions are composed by discrete components. A Combinatorial Optimization problem is described in terms of a *solution space* S, a set of (possibly empty) *constraints* Ω and an *objective function* $f : S \to \mathbb{R}^+$ to be minimized (maximized).

The colony of artificial ants builds solutions in an incrementally way: each ant probabilistically chooses a component to add to a partial solution built so far, according to the problem constraints. The random choice is biased by the *artificial* pheromone value τ related to each component c and by a heuristic function η. Both terms evaluate the desirability of each component. The probability that an ant will choose the component c is

$$p(c) = \frac{[\tau(c)]^\alpha [\eta(c)]^\beta}{\sum_x [\tau(x)]^\alpha [\eta(x)]^\beta} \quad (1)$$

where the sum on x ranges on all the components which can be chosen, and α and β are tuning parameters which differentiate the pheromone and heuristic contributions.

The pheromone values represent a kind of memory shared by the whole ant colony and are subject to *u*pdate and *e*vaporation. In the most applications only the best solutions are considered in the pheromone update phase: the global best solution found so far (*best–so–far*) and/or the best solution found in the current iteration (*iteration–best*). Moreover, most ACO algorithms use the following update rule (Blum 2005):

$$\tau(c) \leftarrow (1-\rho)\cdot\tau(c) + \rho \cdot \sum_{s \in \Psi_{upd} : c \in s} F(s) \quad (2)$$

where Ψ_{upd} is the set of solutions involved in the update, F is the so called *quality function*, which is a decreasing

function of the objective function f (increasing if f is to be maximized), and $\rho \in]0, 1[$ is the pheromone evaporation rate. ρ is a typical ACO parameter which was introduced to avoid a premature convergence of the algorithm towards sub–optimal solutions.

The simulation of the ant colony is iterated until a satisfactory solution is found, a termination criterion is satisfied or a given number of iterations is reached.

Planning by ACO

We have defined an ACO algorithm which handles a colony of planning ants with the purpose of solving planning problems by optimizing solution plans with respect to the a given metric. According to the main features of ACO the ants–planners of the colony are stochastic and heuristic–based.

Each ant–planner executes a forward search, starting from the initial state $\mathcal{I}$ and trying to reach a state in which the goal $\mathcal{G}$ is satisfied. The solution is built step by step by adding components. At each step, the ant–planner search process performs a randomized weighted selection of a solution component c which takes into account both the pheromone value $\tau(c)$ associated to the component and the heuristic value $\eta(a)$ computed for each action a executable in the current state and related to the chosen solution component. Once an action a has been selected, the current state is updated by means of the effects of a.

The construction phase stops when at least one of the following termination condition is verified

1. a solution plan is found, i.e. a state where all the goals are true is reached;
2. a dead end is met, i.e. no action is executable in the current state;
3. an upper bound L_{max} for the number of execution steps is reached.

In the ACOPlan algorithm here described, all the ant–planners in the colony are homogeneous and they can fully share information about the pheromone values distribution. In general, it is possible to define colonies composed by heterogeneous planners with different degrees and types of information sharing.

The algorithm

The generic ACOPlan algorithm (Baioletti et al. 2009a; 2009b; 2009c; 2009d) is described in Fig. 1. Both the two versions of ACOPlan partepating at the IPC-2011 implements the same algorithm; they differ in the implementation and in the plan evaluation function. Let $(\mathcal{I}, \mathcal{G}, \mathcal{A})$ be the planning problem, the optimization process is iterated for a given number N of iterations, in which a colony of n_a ants build plans with a maximum number of steps L_{max}. At each step, each ant chooses an action among the executable ones by the *ChooseAction* function that encodes the transition probability function previously described. When all ants have completed the search phase, the best plan π_{iter} of the current iteration is selected and the global best plan π_{best} is possibly updated. Finally, the pheromone values of the solution components are updated by means of the function *UpdatePheromone* that implements the updating rules 2. Relevant parameters are c_0 which denotes the initial value for the pheromone, ρ which represents the evaporation rate and σ that is a parameter of the pheromone update rule (see 5).

Algorithm 1 The algorithm ACOPlan

```
1: π_best ← ∅
2: InitPheromone(c_0)
3: for g ← 1 to N do
4:     for m ← 1 to n_a do
5:         π_m ← ∅
6:         s ← I
7:         A_1 ← executable actions in I
8:         for i ← 1 to L_max while A_i ≠ ∅ and G ⊈ s do
9:             a ← ChooseAction(A_i)
10:            extend π_m with a
11:            s ← Res(s, a)
12:            A_{i+1} ← executable actions on s
13:        end for
14:    end for
15:    find π_iter
16:    update π_best
17:    UpdatePheromone(π_best, π_iter, ρ, σ)
18: end for
```

The Pheromone Model

The effectiveness of an ACO algorithm firstly depends on the choice of the pheromone model and its data structures. A good model should be simple to compute but enough informative to characterize the context in which an ant–planner can choose a specific action. Moreover it should allow them to distinguish the context of most successful choices from the worst ones. On the other hand the characterization of the component should not be too much detailed in order to allow the pheromone to deposit in a significant quantity.

In (Baioletti et al. 2010) several pheromone models have been proposed and empirically compared by systematic experiments. In the ACOPlan setting for IPC2011 the pheromone model *Action–Action (AA)* is used. In this model a notion of *local history* is introduced: the pheromone depends both on the action a under evaluation and on the last executed action a', i.e. the pheromone is a function $\tau(a, a')$. Considering only the previous action is the simplest way in which the action choice can be directly influenced by history of previous decisions. The pheromone model AA allows a manageable representation and defines a sort of local first order Markov property.

The heuristic *FFAC* for actions costs

The heuristic function is a key feature of ACOPlan because it directly affects the transition probability function (1) used to synthesized the solution plan. The heuristic value for a component c is defined by

$$\eta(c) = \frac{1}{h(s_c)}$$

where h is an heuristic function which evaluates the state s_c resulting from the execution of the action a_c associated to the component c in the current state.

The versions of ACOPlan partecipating at IPC-2011 use a heuristic function which takes into account of the action costs and which is very similar to the heuristic function used in SAPA (Do and Kambhampati 2003) with the sum propagation for action cost aggregation.

This heuristic exploits the idea of FF to use the *relaxed planning graph* structure to compute a relaxed solution plan and use the provided information to estimate the cost of the actual solution. The relaxed planning graph is derived by the classical planning graph introduced in (Blum and Furst 1997) by ignoring the delete list, i.e. the negative effects of the actions.

To compute the heuristic value of a state s, first a relaxed planning graph from s to $\mathcal{G}$ is built, then a relaxed plan π^+ is extracted from it by a backward search. The technique used for the π^+ extraction is different to the one used in the FF system because the concept of *action difficulty* has been redefined taking into account the action costs. The cost of the relaxed plan $c(\pi^+)$ is the heuristic value that estimates the cost needed to reach the goals by the current state s:

$$c(\pi^+) = \sum_{a \in \pi^+} c(a)$$

During the graph construction phase a minimum estimated cost c_k is assigned to each fact and each action for every level k of the graph in the following way:
for each $a \in A_k$

$$c_k(a) = c(a) + \sum_{f \in pre(a)} c_k(f). \quad (3)$$

for each $f \in F_k$

$$c_k(f) = \min\{c(a) + \sum_{b \in pre(a)} c_{k-1}(b) \ : \ a \in A_{k-1}, f \in add(a)\}. \quad (4)$$

while, for each $f \in F_0$, we have $c_0(f) = 0$.

It is worth to note that the preliminary costs estimation introduced by the formulas (3) and (4) depends on the level in the relaxed planning graph, i.e. we do not have static costs as in the heuristic proposed by Keyder and Geffner in (Keyder and Geffner 2008) but dynamic costs depending on the level at hand.

The entire process of computation of the heuristic function is shown in Algorithm 2 where the core technique used to compute the relaxed plan π^+ relies in functions λ and $BestAction$. The function λ, applied to an action a or to a fact f, returns the first level k in which $c_k(a)$ or $c_k(f)$ gets the minimum value, while the function ***BestAction(g,k)*** returns the action $a \in A_{k-1}$ with the minimum ***difficulty*** which can adds g at the level k. The concept of action difficulty is defined in this context by the cost of the action, i.e. $difficulty_{k-1}(a) = c_{k-1}(a)$.

Note that when an action a is added to the plan π^+, the effects of a are deleted from G_k. This feature allows to take into account the so called *positive interactions*, i.e. it avoids to use a new action a' to reach a goal g, when an already selected another action also reaches g. This feature is one of the most important inheritance we receive by FF: it allows to the heuristic FFAC to differ in one more point to the heuristic h_{sa} used in *FF*(h_a) (Keyder and Geffner 2008) that cannot take into account *positive interactions*. As *FF*, also *FFAC* can compute the set of *Helpful Actions*.

Algorithm 2 Computing FFAC

```
1: function FFAC(s)
2:     build the relaxed planning graph from s and compute c_k's
3:     for k ← 0 to L do G_k ← {g ∈ G : λ(g) = k}
4:     π+ ← ∅
5:     for k ← L downto 0 do
6:         while G_k ≠ ∅ do
7:             take g from G_k
8:             a ← BestAction(g, k)
9:             add a to π+
10:            for each p ∈ pre(a) do add p to G_λ(p)
11:            for each e ∈ add(a) do remove e from G_k
12:        end while
13:    end for
14:    return c(π+)
15: end function
```

Since the computation of the heuristic function is computationally demanding and the number of evaluation for h is huge (each ant at each step should compute h for each reachable state), a state–cache data structure is used to store heuristic values and some other informations for each visited state. This structure is implemented through a hash table to speed up the search process. Moreover in order to avoid memory problems the size of the hash table is bounded and table entries are eventually removed using a Least Recently Used strategy.

Plan comparison and Pheromone updating

A critical point of the optimization process is the ability of comparing plans found by the colony of planner ants. This is particularly important in pheromone updating phase where the best plan of the iteration must be selected, as well as in the general ACOPlan which returns the best plan found in all the iterations.

Any comparison criteria should obviously prefer a *solution plan* to a *non solution plan*. On the other hand comparison of two *solution plans* π, π' can be easily based on actions costs, because it is possible to compute their respective costs and prefer the plan with the lowest cost or, when they are equal in cost, let prevail the one with the lesser plan length metric, either sequential or parallel.

A comparison criteria cannot be easily defined when both plans π and π' are not solution plans. In this case a plan π is evaluated by a combination of the heuristic value on the best state s ever reached by π with the cost of reaching s. The two versions of ACOPlan partecipating at IPC-2011 implement two different evaluation schema that use two different combinations of the heuristic value $h(s)$ and the cost of reaching

s.

The *pheromone update phase* evaporates all the pheromone values and increases the pheromone value of the components belonging to π_{iter} and π_{best} according to the formula

$$\tau(c) \leftarrow (1-\rho)\cdot\tau(c) + \rho\cdot\Delta(c) \quad (5)$$

where

$$\Delta(c) = \begin{cases} \sigma & \text{if } c \text{ belongs to } \pi_{iter} \\ 1-\sigma & \text{if } c \text{ belongs to } \pi_{best} \\ 1 & \text{if } c \text{ belongs to both} \\ 0 & \text{otherwise} \end{cases}$$

and σ is usually set to $\frac{2}{3}$ which increases a component belonging to the best plan in the iteration by a double quantity with respect to a component belonging to the best plan so far, thus enforcing the exploration, instead of the exploitation.

Conclusions

ACOPlan employs a colony of stochastic and heuristic–based ants–planners in the framework of ACO metaheuristic. The two versions of the planner run in the IPC-2011 use the *FFAC* cost-based heuristic and the *Action–Action* pheromone model. They differ for the implementation and the plan evaluation function.

The comparison of ACOPlan with state of the art planners shows that the stochastic ACO based approach is optimal in many hard problems, and is competitive with respect to the percentage of solved problems and from the point of view of distribution of solution quality.

Acknowledgments. This paper is partially supported by the INDAM-GNCS project "Fairness, Equità e Linguaggi"

References

Baioletti, M.; Milani, A.; Poggioni, V.; and Rossi, F. 2009a. An ACO approach to planning. In *Proc of the 9th European Conference on Evolutionary Computation in Combinatorial Optimisation, EVOCOP 2009*.

Baioletti, M.; Milani, A.; Poggioni, V.; and Rossi, F. 2009b. Ant search strategies for planning optimization. In *Proc of the International Conference on Planning and Scheduling, ICAPS 2009*.

Baioletti, M.; Milani, A.; Poggioni, V.; and Rossi, F. 2009c. Optimal planning with ACO. In *Proc of AI*IA 2009, LNCS 5883*, 212–221.

Baioletti, M.; Milani, A.; Poggioni, V.; and Rossi, F. 2009d. PlACO: Planning with Ants. In *Proc of The 22nd International FLAIRS Conference. AAAI Press*.

Baioletti, M.; Milani, A.; Poggioni, V.; and Rossi, F. 2010. Experimental Evaluation of Pheromone Models in ACOPlan. In *Proc. of the 17th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Blum, C. 2005. Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews* 2(4):353–373.

Do, M. B., and Kambhampati, S. 2003. Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research (JAIR)* 20:155–194.

Dorigo, M., and Stuetzle, T. 2004. *Ant Colony Optimization*. Cambridge, MA, USA: MIT Press,.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proc. of ECAI 2008*, 588–592.

Arvand: the Art of Random Walks

Hootan Nakhost
University of Alberta
Edmonton, Canada
nakhost@ualberta.ca

Martin Müller
University of Alberta
Edmonton, Canada
mmueller@ualberta.ca

Richard Valenzano
University of Alberta
Edmonton, Canada
valenzan@ualberta.ca

Fan Xie
University of Alberta
Edmonton, Canada
fxie2@ualberta.ca

Abstract

Arvand is a stochastic planner that uses Monte Carlo random walks (MRW) planning to balance exploration and exploitation in heuristic search. Herein, we focus on the latest developments of Arvand submitted to IPC'11: smart restarts, the online parameter learning system, and the integration of Arvand and the postprocessing system Aras.

Introduction

Most of the state of the art heuristic planners such as FF (Hoffmann and Nebel 2001), the top performer at IPC'00; Fast Downward (FD) (Helmert 2006), the top performer at IPC'04; and LAMA (Richter *et al.* 2008), the top performer at IPC'08, use variations of greedy search algorithms such as best first search and enforced hill climbing. These methods totally exploit the heuristic information and do not do much exploration in the search space. While this exploitive nature contributes to very fast performance in many IPC benchmarks, it can lead to serious inefficiencies where the heuristic values are misleading. For example, most of the current planners have poor performance on problems involving resource management (Nakhost *et al.* 2010).

Arvand (Nakhost and Müller 2009) is among the new wave of heuristic planners such as identidem (Coles *et al.* 2007) and the planner introduced by (Lòpez and Borrajo 2010), that try to use more explorative search algorithms to handle heuristic deficiencies. Arvand's main idea is to combine the exploration power of fast random walks with the strength of the available heuristic functions. Although still the heuristic function is the main guiding engine of the algorithm, local exploration using random walks and fast transition in the search space using long jumps are significantly helpful in regions such as plateaus where heuristic values do not help much.

The Arvand Planner

Arvand uses a forward-chaining local search algorithm. Each run of Arvand consists of one or more "search episodes". Each episode starts from the initial state, and transitions through the search space by jumping to the states found in the local neighborhood until a termination criterion is met. At each transition or search step the next state s is chosen from a set of random samples obtained by random walks. Before each transition, n bounded random walks, sequences of randomly selected actions, are run and the obtained states are evaluated. Since usually the heuristic evaluation is orders of magnitude slower than state generation, only the endpoints of random walks are evaluated using a heuristic function. The current Arvand uses the FF heuristic function. This enables the algorithm to get a large set of samples in a short period of time. After running n random walks, the planner jumps to the endpoint with minimum heuristic value. The episode terminates if either a goal state is found, or it fails to improve the minimum heuristic value reached for several jumps. After termination a new search episode starts by restarting from the initial state. For the details of the algorithms that control the number and length of random walks see (Nakhost and Müller 2009). Arvand is implemented based on Fast Downward (FD) code base.

Smart Restarts

In the original version of Arvand, the search always restarts from the initial state (basic restarts). However, it is shown that in more constrained problems a more evolved restarting mechanism called smart restarts (SR) is helpful (Nakhost *et al.* 2010). In SR a pool of most promising search episodes are kept in the memory and each time that the algorithm restarts, a state visited by one of the episodes in the pool is selected as the next restarting point. Search episodes are compared based on the minimum heuristic value they have reached. Each time an episode fails, it can replace the worst episode in the pool if the minimum heuristic value reached by the new episode is lower. To select the restarting point, first an episode e is selected from the pool randomly and then a state visited by e is randomly selected. Therefore, the states that are repeated in several episodes have a higher chance to be chosen. If SR is used from the beginning, then the information inside the pool get too biased to the first episode. In Arvand, SR gets activated after N restarts. Before that basic restarting is used. Therefore when SR is activated the pool contains N episodes. The parameter N and the size of the pool are both set to 50 for the competition.

Biasing the Random Walks

Two different methods are used to bias the random action selection inside Arvand: Monte Carlo Helpful Actions (MHA)

and Monte Carlo Deadlock Avoidance (MDA) (Nakhost and Müller 2009). The main idea is to use the statistics from the earlier random walks to bias the action selection towards more promising actions and away from non-promising ones. MHA uses Gibbs sampling to give priority to actions that have been more often selected as a helpful action at an endpoint. Helpful actions are computed as a by product of the heuristic function at the endpoints. MDA keeps track of the number of times that each action has appeared in a failed random walk (a walk that reaches a state with no applicable actions) and tries to sample actions with higher failure rate less often.

Parameter Learning

(Nakhost and Müller 2009) showed that different configurations of Arvand perform well in different types of domains. In the current version of Arvand an online learning algorithm is used to find the best configuration of the parameters for the given problem. The problem of selecting the best configuration from a set of possible configurations C can be viewed as a multi-armed bandit problem, where pulling an arm corresponds to using the corresponding configuration for the next search episode of Arvand. Each time that Arvand restarts, a bandit algorithm is used to select one of the configurations and then based on the minimum heuristic value (h_{min}) obtained in the search episode a reward is assigned to the corresponding arm. Let h_i be the heuristic value of the initial state, then the reward r is computed as follows: $r = max(0, 1 - (h_i/h_{min}))$. Arvand uses the upper confidence bounds (UCB) algorithm (Auer *et al.* 2002) to balance the exploration and the exploitation in the configuration selection.

Currently, C includes three configurations: two MHA versions with initial length of random walks 1 and 10; and one MDA with initial length 1. Since in some problems running a search episode might be quite slow, and in the initial phase of UCB all the configurations are tried once, the best configuration might not be selected enough times to be able to solve the task. To remedy this problem, for the initial episodes a smaller number of random walks n per search step is used to speed up the learning process. Specifically, for the first three episodes n is set to 100 and for the next episodes n is doubled up until it reaches the maximum 2000.

Integration with Aras

Since IPC'08 there has been an emphas on the quality of the generated plans. This has been perfectly reflected in the competition's scoring function: the cost of the best known plan divided by the cost of the plan. Aras (Nakhost and Müller 2010) is a fast postprocessing tool that is able to improve the quality of a given plan generated by any planner. Aras uses simple fast techniques to locally search the neighborhood of the given plan and works well for a wide range of planners including LAMA, which is designed to generate high-quality plans. The most effective technique in Aras, called Plan Neighborhood Graph Search (PNGS), builds a graph encapsulating the search space close to the original plan and then finds the lowest-cost path inside the graph. The anytime version of Aras starts with a small initial size for the graph and then iteratively increases the size of the graph until the memory limit is reached.

An alternation of Aras and Arvand is used to obtain high-quality plans. First Arvand is run until a solution is found. The solution is saved and fed into anytime Aras to be improved. After Aras reaches the memory limit, which is set to 2 GB, a new search episode of Arvand is used to find another plan and again it is fed into Aras. This process continues until the time limit is reached. In the whole process, as soon as a plan with better quality is found it is saved.

Arvand also uses a bounding mechanism to stop episodes or random walks that already exceed the cost of a previously found solution. However, the solution bound is only updated by plans generated by Arvand itself. The reason is that usually the bounds obtained from Aras' solution are too tight for Arvand and significantly lower the probability of reaching any solution. As the result the number and diversity of the plans that are fed into Aras gets much lower and this has a detrimental effect in the best quality plan reached by the system.

Acknowledgements

We would like to thank Malte Helmert for giving us access to the Fast Downward code. We would also like to acknowledge the support of NSERC and Alberta Ingenuity.

References

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47:235–256, May 2002.

Andrew Coles, Maria Fox, and Amanda Smith. A new local-search algorithm for forward-chaining planning. In *Proc. ICAPS'07*, pages 89–96, 2007.

Malte Helmert. The Fast Downward planning system. *JAIR*, 26:191–246, 2006.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

Carlos Linares Lòpez and Daniel Borrajo. Adding diversity to classical heuristic planning. In *Proc. SOCS'10*, pages 73–80, Atlanta, USA, 2010.

Hootan Nakhost and Martin Müller. Monte-Carlo exploration for deterministic planning. In *Proc. IJCAI'09*, pages 1766–1771, 2009.

H. Nakhost and M. Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In *Proc. ICAPS'10*, pages 137–144, 2010.

H. Nakhost, J. Hoffmann, and M. Müller. Improving local search for resource-constrained planning. Technical Report TR 10-02, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2010.

Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *Proc. AAAI'08*, pages 975–982, 2008.

BRT: Biased Rapidly-exploring Tree

Vidal Alcázar
Universidad Carlos III de Madrid
Avenida de la Universidad, 30
28911 Leganés, Spain

Manuela Veloso
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA

Abstract

This paper describes the planner BRT (Biased Rapidly-exploring Tree). This planner is basically a Rapidly-exploring Random Tree (RRT) adapted to automated planning that employs Fast-Downward as the base planner. The novelty in this case is that it does not sample the search space in a random way; rather, it estimates which propositions are more likely to be achieved along some solution plan and uses that estimation (called bias) in order to sample more relevant intermediates states. The bias is computed using a message-passing algorithm on the planning graph with landmarks as support.

Overview

Landmarks are disjunctive sets of propositions or actions of which at least one component must be achieved or executed at least once in every solution plan to a problem (Richter, Helmert, and Westphal 2008). However, computing the complete set of landmarks or even proving that a proposition or action is indeed a landmark is PSPACE-complete (Hoffmann, Porteous, and Sebastia 2004).

A notable similarity exists between landmarks in automated planning and backbone variables (Kilby et al. 2005) in CSPs. A backbone variable is defined as a variable that takes the same value in every solution for a given problem. Again, determining which are the backbone variables in a problem is intractable in general. In order to overcome this, approximate methods that compute an estimation of the probability that a variable has of taking a given value - also known as bias - have been proposed. In particular, message-passing algorithms appear to be well suited to this kind of probabilistic environment.

Biases can be seen in automated planning as the probability of a proposition or an action being respectively achieved or executed in a solution plan. Extending the links between these two cases, just like backbones variables are variables with a bias of 1 for a given value, landmarks could also be defined as having also a bias of 1. Because of the characteristics of planning problems, though, a straightforward correlation between variable bias and proposition/action bias does not exist. In particular, the fact that it is relevant at which time step variables and actions must be achieved or executed invalidates these assumptions. To take into account this fact, a planning graph (Blum and Furst 1997) is used so the bias is computed for every variable and action at each time step in which they can appear.

The constraints encoded by the planning graph alone may not suffice to find a good bias. To solve this, landmarks are added to the planning graph as sources of probability. This requires first having an estimation of where landmarks should appear in the planning graph, which is obtained by turning the landmark graph into a SAT problem that encodes time steps and using a SAT solver to get a possible assignment.

The general process is as follows:

- First, the landmark graph is encoded as a SAT problem and an estimation of when the landmarks are needed is obtained.
- Second, a planning graph enriched with the landmarks at the aforementioned estimated positions is generated and a message-passing algorithm is used to compute the bias - in this case Expectation Maximization Survey Propagation (Hsu and McIlraith 2009).
- Finally, biases are used to introduce probabilities in the sampling process of the RRT.

A SAT Compilation of the Landmark Graph

Current methods for computing landmarks are able not only to find landmarks, but also to establish partial orders between them. These orderings connect the landmarks forming a directed graph, the landmark graph. Figure 1 shows the landmark graph of the Sussman's anomaly slightly simplified for the sake of clarity. There are several ordering relationships between landmarks propositions (Richter, Helmert, and Westphal 2008):

- Natural ordering: A proposition *a is naturally ordered before b* if *a* must be true at some time before *b* is achieved
- Necessary ordering: A proposition *a is necessarily ordered before b* if *a* must be true one step before *b* is achieved
- Greedy-necessary ordering: A proposition *a is greedy necessarily ordered before b* if *a* must be true one step before *b* when *b* is first achieved
- Reasonable ordering: A proposition *a is reasonably ordered before b* if, whenever *b* is achieved before *a*, any

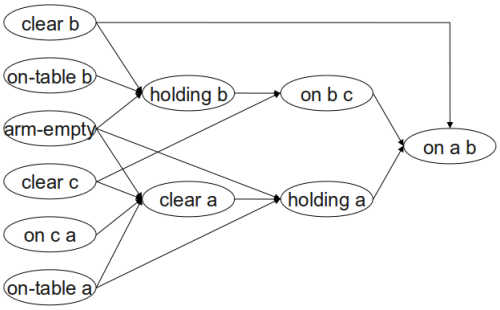

Figure 1: Simplified landmark graph of the Sussman's anomaly.

plan must delete *b* on the way to *a*, and re-achieve *b* after or at the same time as *a*

This representation does not take into account that a landmark may be needed at several time steps and does not provide an insight of the total order of the landmarks, as it only considers relative orders between them. Besides, it usually underestimates the minimum parallel length of the solutions to the problem. Figure 1 illustrates these facts: *(arm-empty)* should appear at every even level as opposed to only at level 0; the number of propositional layers is four, meaning that in theory a solution plan with only three actions may be possible, whereas the shortest parallel plan has 6 actions; *(holding a)* and *(holding b)* appear at consecutive levels, despite needing 2 actions to achieve either proposition from a state in which the other is true,... An earlier work that tried to group landmarks into layers in order to partition the problem (Sebastia, Onaindia, and Marzal 2006) partially addressed these issues, although not in an explicit way and with the inconvenience of being computationally expensive in some planning domains.

To solve this, a SAT compilation of the landmark graph is proposed. It is inspired by the SAT-compilation process of optimal makespan planners, in which propositions and actions are represented by a different variable at every time step. In this case the landmark propositions at the different levels are variables, and the clauses represent the different constraints that exist between them. Constraints are either the orderings that define the landmark graph or relations of binary mutual exclusivity. In particular, long distance mutexes (Chen, Xing, and Zhang 2007), which are a generalization of regular mutexes, will be used. The different types of clauses are the following (*ini* represents the time step at which a given proposition can be true for the first time):

- Existential clauses: Every landmark must be true at at least one time step ($a_{ini} \vee ... \vee a_n$)
- Natural orderings: *a* must be true at some time step before *b* is true ($a_{ini} \vee ... \vee a_{t-1} \cup \neg b_t$)
- Necessary orderings: Either *a* or *b* must be true at the time step before *b* is true ($a_{t-1} \vee b_{t-1} \vee \neg b_t$)
- Greedy-necessary orderings: Either *a* or *b* must be true at some time step before *b* is true ($a_{ini} \vee ... \vee a_{t-1} \vee b_{ini} \vee ... \vee b_{t-1} \vee \neg b_t$)
- Reasonable orderings: If *a* and *b* are true at the same level, *a* must be true at the time step before that level ($a_{t-1} \vee \neg a_t \vee \neg b_t$)
- Distance between londexes: *a* cannot be true at a time step *t'* if *b* is true at *t* such that $t - distance(a, b) > t' \leq t$ $(\neg a_{t-(distance-1)} \vee \neg b_t) \wedge ... \wedge (\neg a_t \vee \neg b_t)$

The resulting encoding is solved using the "ramp-up" method most SAT-based planners employ: compile the problem with a given maximum length, try to find a solution using a SAT solver and, if no solution is found, repeat the process incrementing the maximum length. The solution may not be unique, so the final position of the landmarks is not sound and should be used only as an estimate.

By solving the problem, a time-stamped landmark graph is obtained. The first consequence is that the depth of the graph is a lower bound on the parallel length of the original problem. The second is that landmarks may appear as true several times, meaning that they must be true at different time steps. The third is that an intuition about the time steps at which a landmark may be necessary is obtained. It should be noted that the total order obtained does not have to be respected by every solution plan.

Table 1 represents the output of this process in the Sussman's anomaly. The table shows at which levels landmarks must be true to satisfy the constraints. The opposite is not true; a landmark does not have to be false when it appears as not required. That a landmark must be false may be useful in some cases, although this is trivially computable using the original constraints along with the solution.

Level:	0	1	2	3	4	5	6
(arm-empty)	x	-	x	-	x	-	-
(on a b)	-	-	-	-	-	-	x
(on b c)	-	-	-	-	x	x	x
(on c a)	x	-	-	-	-	-	-
(clear a)	-	x	x	x	x	-	-
(clear b)	x	-	x	-	-	x	-
(clear c)	x	-	x	x	-	-	-
(on-table a)	x	-	-	-	-	-	-
(on-table b)	x	-	-	-	-	-	-
(holding a)	-	-	-	-	-	x	-
(holding b)	-	-	-	x	-	-	-
(holding c)	-	x	-	-	-	-	-

Table 1: Output of the solution of the SAT problem generated from the landmark graph

This solution is a good example of how some of the shortcomings of the landmark graph can be overcome. First, the number of levels is the minimum required; second, trivial landmarks like *(arm-empty)* being required on every even level but the last one are detected; third, the positions at which landmarks appear as needed offer a great deal of information with regards to possible solutions.

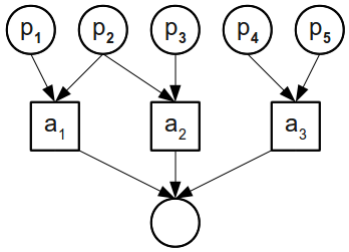

Figure 2: An example of bias computation. In this case the positive bias of p_2 is higher than the bias of the other propositions, as it appears as true in a higher number of solutions.

Estimating the Bias of Propositions and Actions

Message-passing compute the marginal distribution on the set of variables by relying on the structure of a factor graph, a bipartite graph whose nodes are either variables or constraints. Message-passing algorithms send values along the edges of the graph representing the influence between variables given the constraints that affect them. The method used in this work is Estimation-Maximization Survey Propagation (EMSP) with a global update rule (Hsu and McIlraith 2009). EMSP is a set of update rules derived using the Estimation Maximization framework for maximum-likelihood parameter estimation that has proved to be effective to solve random SAT problems. Survey propagation is essentially a variation of belief propagation in which the bias, instead of being split as positive or negative, accepts a third value, the "joker" or "don't care" one. This value represents the probability of the value of the variable not being critical for the satisfiability of the solution; this is, whether a solution remains valid when changing the value of the variable. On the other hand, the Estimation-Maximization framework has the advantage of guaranteeing convergence even in graphs with loops, unlike regular message-passing algorithms. This way, EMSP initializes the bias of the variables with random values and iterates by doing a two-step process: first, every variable sends its bias to the constraints it appears in, allowing the computation of the probability of those constraints to be satisfied. Second, every variable updates its bias based on the previously computed probability.

Figure 2 shows an example of what biases mean in a SAT problem. In this case, there are three actions a_1, a_2, a_3 that achieve a goal and five propositions p_1, p_2, p_3, p_4, p_5 that are the preconditions of the actions as represented by the arrows. Only one action can be executed, which means that they are mutex between them. Every action and proposition besides the goal is a variable. The clauses are $(a_1 \vee a_2 \vee a_3)$, $(\neg a_i \vee \neg a_j)$ for each $i \neq j$ and $(p_i \vee \neg a_j)$ for each $p_i \in pre(a_j)$. If EMSP is used, the positive bias of p_2 is 0.5 whereas the bias of the rest of the propositions is around 0.3 This is because out of all the possible solution assignments to the problem, p_2 appears as true in more cases. Also this example shows why choosing Survey Propagation instead of regular Belief Propagation may be useful: let's suppose a_1 was chosen as the supporting variable for the clause $(a_1 \vee a_2 \vee a_3)$. In that case the values of p_1, p_2, a_2, a_3 are enforced by the constraints. However, the value of the rest of the propositions does not matter - they could be true or false and the assignment would still satisfy the constraints. Survey Propagation captures this notion, which allows to compute the positive and negative biases without being affected by situations like the one described.

This planning graph can be easily seen as a factor graph in which the factor vertices are the constraints that are represented by the edges and mutexes of the planning graph. A reasonable formulation of the planning graph as a factor graph is its compilation to a SAT problem, as done by SAT-based planners like Blackbox (Kautz and Selman 1999). In this case, the variables represent the different propositions and actions at different time steps, and the clauses are the constraints between them. This choice implies that the estimated bias would be about actions and propositions *at given time steps*. This is in opposition to the concept of landmark, which states that they must be true along every solution plan without any particularization of when.

To improve the inference process landmarks are inserted in the positions obtained from the solution of the SAT encoding of the landmark graph and given a bias of 1. Furthermore, the number of levels of the solution obtained from the SAT compilation of the landmark graph is a sound lower bound on the parallel length of any possible solution plan, so the planning graph should have at least that many levels. Algorithm 1 gives an outline of the whole process done when computing the bias of actions and propositions.

Algorithm 1: Computation of biases enhanced by landmarks

```
Data: Current problem P = (S, A, I, G)
Result: Survey survey
begin
    LandmarkGraph ⟵ computeLandmarks(P)
    MinimumLevel ⟵ computeHmax(P)
    LandmarkSAT ⟵ NULL
    while notLandmarkSAT do
        LandmarkEncoding ⟵
        compileLandmarks(LandmarksGraph,
        MinimumLevel)
        LandmarkSAT ⟵
        solve(LandmarkEncoding)
        MinimumLevel ⟵ MinimumLevel + 1
    PlanningSAT ⟵
    generateCNF(P, MinimumLevel − 1)
    PlanningSAT ⟵
    insertLandmarks(PlanningSAT, LandmarkSAT)
    survey ⟵ computeSurvey(PlanningSAT)
    return survey
end
```

Biased RRT

One of the possible uses of the biases is sampling the search space. The goal is to find intermediate states that may be

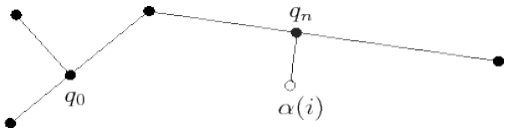

Figure 3: Search phase of the RRT. The local search expands towards the rightmost sampled state with a limit. When the limit is reached, q_n is added to the tree.

relevant to the search. Doing this the problem can be partitioned into several smaller problems, which greatly decreases the complexity of the task. An incremental algorithm can be used to try to reach those states and build a path to the goal state. A baseline planner is used to solve the different subproblems. In this case, we will use a Rapidly-Exploring Random Tree (RRT) (Kuffner and LaValle 2000) adapted to automated planning following some of the ideas introduced by RRT-Plan (Burfoot, Pineau, and Dudek 2006). Fast-Downward with the FF heuristic, preferred operators and greedy best first search with lazy evaluation will be the base planner.

RRTs try to find a path to the goal by randomly sampling the search state and trying to join the closest node of the tree to the sampled state. The algorithm works as follows: first, a state that is not inside an obstacle is sampled; then, the closest node of the tree is found using a measure of distance. After having found the closest node, a baseline planner is called with a limit on time or length that tries to reach the sampled state. When it is reached or the planner surpasses the imposed limit, the last expanded state is added as a new node of the tree with an edge connecting it to the node that served as initial state for the subproblem. In single-query RRTs after expanding towards a sampled state often a new search is invoked trying to join the newly created node with the goal state. Another approach used to speed up convergence is, instead of sampling, with a probability *p* the closest node to the goal is chosen and expanded towards it. Figure 3 shows how the tree is built towards the goal while sampling at the same time.

Mutexes are used when sampling to avoid selecting unreachable steps and a reachability analysis from the sampled state is done to avoid dead-end states. Besides, sampling with a random probability may lead to exploring areas of the search space which are not relevant to the solution of the problem. To prevent this from happening, the use of biases is proposed. Sampling is thus a several steps process: first, a random proposition level of the planning graph is chosen, as biases differ from level to level. Second, a variable from the SAS+ formulation of the problem is randomly chosen. After that, a proposition belonging to the invariant is picked using roulette wheel selection with the bias as the probability of selection. When a proposition is chosen all the propositions belonging to other variables that are mutex with the chosen one are discarded. This process is repeated until a proposition from each variable has been selected. If a variable is selected and has no valid propositions the sampled state is unreachable due to mutexes and the process is restarted from scratch.

Acknowledgments

This work has been developed when Vidal Alcázar was visiting Manuela Veloso at Carnegie Mellon University supported by a FPI grant from the Spanish government associated to the MICINN project TIN2008-06701-C03-03.

References

Blum, A., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artif. Intell.* 90(1-2):281–300.

Burfoot, D.; Pineau, J.; and Dudek, G. 2006. RRT-Plan: A randomized algorithm for STRIPS planning. In *ICAPS*, 362–365. AAAI.

Chen, Y.; Xing, Z.; and Zhang, W. 2007. Long-distance mutual exclusion for propositional planning. In *IJCAI*, 1840–1845.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *J. Artif. Intell. Res. (JAIR)* 22:215–278.

Hsu, E. I., and McIlraith, S. A. 2009. Varsat: Integrating novel probabilistic inference techniques with DPLL search. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, 377–390. Springer.

Kautz, H. A., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In *IJCAI*, 318–325. Morgan Kaufmann.

Kilby, P.; Slaney, J. K.; Thiébaux, S.; and Walsh, T. 2005. Backbones and backdoors in satisfiability. In *AAAI*, 1368–1373. AAAI Press / The MIT Press.

Kuffner, J. J., and LaValle, S. M. 2000. RRT-Connect: An efficient approach to single-query path planning. In *ICRA*, 995–1001. IEEE.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *AAAI*, 975–982. AAAI Press.

Sebastia, L.; Onaindia, E.; and Marzal, E. 2006. Decomposition of planning problems. *AI Commun.* 19(1):49–81.

The CBP planner

Raquel Fuentetaja
Departamento de Informática, Universidad Carlos III de Madrid
rfuentet@inf.uc3m.es

Abstract

This short paper provides a high-level description of the planner CBP (Cost-Based Planner). CBP performs heuristic search in the state space using several heuristics. On one hand it uses look-ahead states based on relaxed plans to speed-up the search; on the other hand the search is also guided using a numerical heuristic and a selection of actions extracted from a relaxed planning graph. The relaxed planning graph is built taking into account action costs. The search algorithm is a modified Best-First Search (*BFS*) performing Branch and Bound (B&B) to improve the last solution found.

Introduction

Usually, in cost-based planning the quality of plans is inversely proportional to their cost. Many planners that have been developed for being able to deal with cost-based planning use a combination of heuristics together with a search mechanism, as METRIC-FF (Hoffmann 2003), SIMPLANNER (Sapena & Onaindía 2004), SAPA (Do & Kambhampati 2003), SGPlan5 (Chen, Hsu, & Wah 2006), LPG-td(quality) (Gerevini, Saetti, & Serina 2004) or LAMA (Richter, Helmert, & Westphal 2008).

One of the problems of applying heuristic search in cost-based planning is that existing numerical heuristics are in general more imprecise than heuristics for classical planning. The magnitude of the error the heuristic commits can be much larger since costs of actions can be very different. Therefore, making numerical estimations using the cost of actions which are not part of the actual optimal solution can lead to a great difference between the actual optimal value and the estimation. For this reason, some additional technique, apart from a numerical heuristic, is needed to help the search algorithm. In CBP we combine the use of a numerical heuristic with the idea of look-ahead states (Vidal 2004), both extracted from a relaxed planning graph aware of cost information. There are multiple ways to implement the idea of look-ahead states: different methods can be defined to compute relaxed plans and to compute look-ahead states from relaxed plans. Also different search algorithms can be used. In CBP we test one such combinations. CBP has been described before in (Fuentetaja, Borrajo, & Linares 2009).

The Numerical Heuristic in CBP

To compute the numerical heuristic, we build a relaxed planning graph in increasing levels of cost. The algorithm (detailed in Figure 1) receives the state to be evaluated, s, the goal set, $\mathcal{G}$, and the relaxed domain actions, $\mathcal{A}^+$. Then, it follows the philosophy of the Dijkstra algorithm: from all actions whose preconditions have become true in any Relaxed Planning Graph (*RPG*) level before, generate the next action level by applying those actions whose cumulative cost is minimum. The minimum cumulative cost of actions in an action layer i, is associated with the next propositional layer (that contains the effects of those actions). This cost is denoted as $cost\_limit_{i+1}$ (initially $cost\_limit_0 = 0$). The algorithm maintains an open list of applicable actions ($OpenApp$) not previously applied. At each level i, all new applicable actions are included in $OpenApp$. Each new action in this list includes its cumulative cost, defined as the cost of executing the action, $cost(a)$, plus the cost of supporting the action, i.e. the cost to achieve the preconditions. Here the support cost of an action is defined as the cost limit of the previous propositional layer, $cost\_limit_i$, that represents the cost of the most costly precondition (i.e it is a *max* cost propagation process). Actions with minimum cumulative cost at each iteration are extracted from $OpenApp$ and included in the corresponding action level, A_i, i.e. only actions with minimum cumulative cost are executed. The next proposition layer P_{i+1} is defined as usual, including the add effects of actions in A_i. The process finishes with success, returning a Relaxed Planning Graph (*RPG*), when all goals are true in a proposition layer. Otherwise, when $OpenApp$ is the empty set, it finishes with failure. In such a case, the heuristic value of the evaluated state is ∞.

Once we have a *RPG* we extract a *RP* using an algorithm similar to the one applied in METRIC-FF (Hoffmann 2003). Applying the same tie breaking policy in the extraction procedure, and unifying some details, the *RPs* we obtain could be obtained with the basic cost-propagation process of SAPA with *max* propagation and ∞-look-ahead, as described in (Bryce & Kambhampati 2007). However, the algorithm to build the *RPG* in SAPA follows the idea of a breadth-first search with cost propagation instead of Dijkstra. The main difference is that our algorithm guarantees the minimum cost for each proposition at the first propositional level containing the proposition.

function compute_RPG_hlevel $(s, \mathcal{G}, \mathcal{A}^+)$
let $i = 0$; $P_0 = s$; $OpenApp = \emptyset$; $cost\_limit_0 = 0$;
while $G \not\subseteq P_i$ **do**
$OpenApp = OpenApp \cup \left\{ a \in \mathcal{A}^+ \setminus \bigcup_{j<i} A_j \mid pre(a) \subseteq P_i \right\}$
forall new action in $OpenApp$ **do**
$cum\_cost(a) = cost(a) + cost\_limit_i$
$A_i = \left\{ a \mid a \in \arg\min_{a \in OpenApp} cum\_cost(a) \right\}$
$cost\_limit_{i+1} = \min_{a \in OpenApp} cum\_cost(a)$
$P_{i+1} = P_i \bigcup_{a \in A_i} add(a)$
if $OpenApp = \emptyset$ **then return** fail
$OpenApp = OpenApp \setminus A_i$
$i = i + 1$
return $P_0, A_0, P_1, ..., P_{i-1}, A_{i-1}, P_i$

Figure 1: Algorithm for building the *RPG*.

Finally, the heuristic value of the evaluated state is computed as the sum of action costs for the actions in the *RP*. Since we generate a relaxed plan, we can use the *helpful actions* applied in (Hoffmann 2003) to select the most promising successors in the search. Helpful actions are the applicable actions in the evaluated state that add at least one proposition required by an action of the relaxed plan and generated by an applicable action in it.

For the 2011 IPC we present two versions of the planner. The first one computes the *RPG* as have been described before in this section. The second one replaces the *max* cost propagation process with an additive propagation process, which involves to compute the $cum\_cost(a)$ in the line 5 of the algorithm in Figure 1 as:

$$cum\_cost(a) = cost(a) + \sum_{q \in pre(a)} cost\_limit_{level(q)}$$

where $level(q)$ is the index of the first propositional layer containing q. Applying the same tie breaking policy in the extraction procedure this additive version will be equivalent to the cost-propagation process of SAPA with *additive* propagation and ∞-look-ahead, as described in (Bryce & Kambhampati 2007) and also to the heuristic h_a (Keyder & Geffner 2008).

Computing Look-ahead States

Look-ahead states are obtained by successively applying the actions in the *RP*. There are several heuristic criteria we apply to obtain a *good* look-ahead state. Some of these considerations have been adopted from Vidal's work in the classical planning case (Vidal 2004), as: (1) we first build the *RP* ignoring all actions deleting top level goals. Relaxed actions have no deletes. So, when an action in the *RP* deletes a top-level goal, probably there will not be another posterior action in the *RP* for generating it. In this case, the heuristic value will be probably a bad estimate; and (2) we generate the look-ahead state using as many actions as possible from the *RP*. This allows to boost the search as much as possible. We do not use other techniques applied in classical planning as the method to replace one *RP* action with another domain action, when no more *RP* actions can be applied (Vidal 2004). Initially, we wanted to test whether the search algorithm itself is able to repair *RPs* through search.

Determining the *best* order to execute the actions in the *RP* is not an easy task. One can apply the actions in the same order they appear in the *RP*. However, the *RP* comes from a graph built considering that actions are applicable in parallel and they have no deletes. So, following the order in the *RP* can generate plans with bad quality. The reason is that they may have many *useless* actions: actions applied to achieve facts other actions delete. We have the intuition that delaying the application of actions as much as possible (until the moment their effects are required) can alleviate this problem. So, we follow a heuristic procedure to give priority to the actions of the *RP* whose effects are required before (they are subgoals in lower layers).[1] To do this, during the extraction process, each action selected to be in the *RP* is assigned a value we call *level-required*. In the general case, this value is exactly the minimum layer index minus one of all the selected actions for which the subgoal is a precondition.

The *RPGs* built propagating costs differs from the *RPGs* built for classical planning. One of the differences is that in the former we can have a sequence of actions to achieve a fact that in the classical case can be generated using just one action. The reason is that the total cost of applying the sequence of actions is lower than the cost of using only one action. In such a case, all the intermediate facts in the sequence are not really necessary for the task. They only appear for cost-related reasons. The only necessary fact is the last one of the sequence. In this case, the way of computing the *level-required* differs from the general case: we assign the same *level-required* to all actions of the sequence, that is the *level-required* of the action that generate the last (and necessary) fact. The justification of this decision is that once the actions have been selected to be in the *RP* we want to execute as more actions as possible of the *RP*, so we have to attend to causality reasons and not to cost-related reasons.

The *level-required* values provide us a partial order for the actions in the *RP*. We generate the look-ahead state executing first the actions with lower *level-required*. In case of ties we follow the order of the *RP*.

The Figure 2 shows a high-level algorithm describing the whole process for computing the look-ahead state given the source state and the relaxed plan (*RP*). The variables min_order and max_order represent the minimum and maximum *level-required* for the actions in the *RP* respectively. Initially, all the actions in the *RP* are marked as no executed, and the variable *current_order* is set to the minimum level required. The `for` sentence covers the relaxed plan executing the actions applicable in the current state and not yet executed, whose level required equals the current order. The execution of one action implies to update the current state and to initialize the current order to the minimum level required. After covering the relaxed plan, if no new ac-

[1] The heuristic procedure described here slightly differs from the one described in (Fuentetaja, Borrajo, & Linares 2009).

```
function obtain_lookahead_state(state, RP)
let min_order = minimum_level_required(RP)
let max_order = maximum_level_required(RP)
let executed[a] = FALSE, ∀a ∈ RP
let current_order = min_order
let current_state = state
while current_order <= max_orden do
    new_action_applied = FALSE
    forall a ∈ RP do
        if not executed[a] ∧ applicable(a, current_state)∧
          level_required(a) = current_order then
            new_state = apply(a, current_state)
            executed[a] = TRUE
            new_action_applied = TRUE
            current_state = new_state
            current_order = min_ordern
    if not new_action_applied then
        current_order = current_order + 1
if current_state ≠ state
    return current_state
else
    return fail
```

Figure 2: High-level algorithm for computing the look-ahead state.

tion has been executed, the current order is increased by one. This process is repeated until the current order is higher than the maximum level required (`while` sentence). Finally the algorithm returns the last state achieved (i.e. the look-ahead state).

The Search Algorithm

The search algorithm we employ is a *weighted-BFS* with evaluation function $f(n) = g(n) + \omega \cdot h(n)$, modified in the following aspects: first, the algorithm performs a Branch and Bound search. Instead of stopping when the first solution is found, it attempts to improve this solution: the cost of the last solution plan found is used as a bound such that all states whose *g-value* is higher than this cost bound are pruned. As the heuristic is non-admissible we prune by *g-values* to preserve completeness; second, the algorithm uses two lists: the *open* list, and the secondary list (the *sec-non-ha* list). Nodes are evaluated when included in *open*, and each node saves its relaxed plan. Relaxed plans are first built ignoring all actions deleting top-level goals (when this produces an ∞ heuristic value, the node is re-evaluated considering all actions). When a node is extracted from *open*, all look-ahead states that can be generated successively starting from the node are included in *open*. Then, its helpful successors are also included in *open*, and its non-helpful successors in the *sec-non-ha* list. When after doing this, the *open* list is empty, all nodes in *sec-non-ha* are included in *open*. The algorithm finishes when *open* is empty. Figure 3 shows a high-level description of the search algorithm (BB-LBFS, Branch and Bound Look-ahead Best First Search). For the sake of simplicity the algorithm does not include the repeated states prune and the *cost_bound* prune for the Branch and Bound. Repeated states prune is performed pruning states with the same facts and higher *g-values*.

```
function BB-LBFS ()
let cost_bound = ∞; plans = ∅; open = I; sec_non_ha = ∅
while open ≠ ∅ do
    node ← pop_best_node(open)
    if goal_state(node) then /* solution found */
        plans ← plans ∪ {node.plan}
        cost_bound = cost(node.plan)
    else
        lookahead = compute_lookahead(node)
        while lookahead do
            open ← open ∪ {lookahead}
            if goal_state(lookahead) then
                plans ← plans ∪ {lookahead.plan}
                cost_bound = cost(lookahead.plan)
                lookahead = compute_lookahead(lookahead)
        open ← open ∪ helpful_successors(node)
        sec_non_ha ← sec_non_ha ∪ non_helpful_successors(node)
        if open = ∅ do
            open ← open ∪ sec_non_ha
            sec_non_ha = ∅
return plans
```

Figure 3: High-level BB-LBFS algorithm.

Summary

This paper described the main parts of the CBP planner: (1) the computation of the relaxed planning graph to obtain a numerical heuristic and the *helpful actions*; (2) the generation of lookahead states based on the same relaxed planning graphs, and (3) the search algorithm. As aforementioned, the relaxed planning graphs are built making use of action costs. The search algorithm is a best first algorithm modified to give priority to *helpful actions* and to include it look-ahead states. Instead of stopping at first solution it continues the search performing Branch and Bound until a time bound is reached. For the competition we have set the ω value in the evaluation function to three. CBP has been implemented in C, reusing part of the code of METRIC-FF.

References

Bryce, D., and Kambhampati, S. 2007. How to skin a planning graph for fun and profit:a tutorial on planning graph based reachability heuristics. *AI Magazine* 28 No. 1:47–83.

Chen, Y.; Hsu, C.; and Wah, B. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research (JAIR)* 26:323–369.

Do, M. B., and Kambhampati, S. 2003. Sapa: A scalable multi-objective heuristic metric temporal planner. *Journal of Artificial Intelligence Research (JAIR)* 20:155–194.

Fuentetaja, R.; Borrajo, D.; and Linares, C. 2009. A look-ahead B&B search for cost-based planning. In *Proceedings of the 13th Conference of the Spanish Association for Artificial Intelligence (CAEPIA)*, 105–114.

Gerevini, A.; Saetti, A.; and Serina, I. 2004. Planning with numerical expressions in LPG. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, 667–671.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state vari-

ables. *Journal of Artificial Intelligence Research (JAIR)* 20:291–341.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI)*.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the 23rd American Association for the Advancement of Artificial Intelligence Conference (AAAI)*, 975–982.

Sapena, O., and Onaindía, E. 2004. Handling numeric criteria in relaxed planning graphs. In *Advances in Artificial Intelligence. IBERAMIA, LNAI 3315*, 114–123.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS)*, 150–159.

CPT4: An Optimal Temporal Planner Lost in a Planning Competition without Optimal Temporal Track

Vincent Vidal
ONERA – DCSD
Toulouse, France
Vincent.Vidal@onera.fr

Overview

Once again, in the 7th International Planning Competition (IPC) of 2011, the optimal temporal track has been cancelled due to a lack of competitors: CPT4, the fourth version of the CPT planner (Vidal and Geffner 2006), was the only submitted planner. This already happened in 2008 for the 6th IPC where CPT3 was alone, while in 2006 for the 5th IPC CPT2 had been compared with CPT1 only and awarded with a "Distinguished Performance" in temporal domains of the optimal track (but well, was it useful?...). In that competition, SAT-based planners were the most efficient for parallel domains, a special case of temporal planning with the conservative semantics first introduced in (Smith and Weld 1999) where actions have uniform durations and overlapping of mutex actions is forbidden.

The only meaningful comparison with other temporal planners, during a planning competition, has been made in 2004 for the 4th IPC, where the first version of CPT had been ranked second in the optimal track (SATPLAN'04 being ranked first), generally being the more efficient in temporal domains against TP4 and HSP^*_a (Haslum and Geffner 2001; Haslum 2004).

We think that optimal temporal planning is a meaningful problem, and that more planners should be submitted to the International Planning Competition in order to try to characterize some "state-of-the-art" among implemented systems. One difficulty is perhaps to follow the full PDDL2.1 semantics, which makes temporal planning EXPSPACE-complete (Rintanen 2007). Actually, the submitted version of CPT4 does not follow PDDL2.1 semantics, but rather the conservative one (Smith and Weld 1999). We think that temporal planning with the conservative semantics is difficult enough, and more closely related to typical scheduling problems. From some preliminary experiments with CPT4 on classical open-shop and job-shop problems, we can say that CPT4 is a relatively good scheduler (i.e. it solves some instances generally considered as difficult).

We decided to enter CPT4 into the 7th International Planning Competition in all tracks, in order to evaluate how far an optimal temporal planner can be from specialized planners in each track. But to make things clear, even if CPT4 participates to this IPC, it should not: the only track for which it has been designed for, the optimal temporal track, has been cancelled...

Implementation

The actual implementation of CPT is based on the version written in C described in (Vidal and Tabary 2006). It includes some additional pruning rules that first appeared in (Vidal and Geffner 2005), as well as the last conflict based reasoning (Lecoutre et al. 2009). The improvements brought to CPT4 are mostly minor: a few bug fixes and slight improvements in the way the constraint propagation rules are written.

One notable improvement is the use of a conflict counting heuristic, inspired by the $wdeg$ heuristic (Boussemart et al. 2004). Each time a contradiction occurs in the constraint propagation engine, a weight attached to the variables and constraints in relation with the violated constraint is incremented. Branching rules are then tweaked to always follow branches that constrain more variables with the highest weights.

Another improvement is a slightly better way of producing optimal sequential plans. As in CPT3, the cost associated to an action (1 for optimal length, any positive value for cost-based planning) is treated as a duration. But instead of forcing all actions to be pairwise mutex, an additional constraint is enforced: an action is excluded from any subsequent partial plan during search, when the sum of the costs of the actions that belong to the current partial plan plus the cost of the considered action exceeds the current bound on the makespan (interpreted as length or cost sum). This means that CPT4 still explores the space of partially ordered plans with concurrent actions, with a constraint that guarantees optimality.

Some Experiments

CPT has been designed with two objectives in mind: to match the performance of the best parallel planners, and to be a powerful planner for optimal temporal planning. While the former can be easily checked by a comparison with SAT-based planners, the latter is a bit more problematic due to the lack of recent efficient optimal temporal planners: we only compared CPT4 with its previous version, CPT3. All experiments are performed on an Intel Xeon X5670 running at 2.93GHz with 4GB of memory and a timeout of 30 minutes.

IPC	domain	#pbs	#solved					
			CPT3	CPT4	MAXPLAN	Mp	SASE	SATPLAN
1	grid	5	1 (2)	1 (2)	1 (2)	2 (1)	**3**	2 (1)
	gripper	20	4 (1)	4 (1)	3 (2)	3 (2)	**5**	3 (2)
	logistics	35	26 (1)	25 (2)	11 (16)	18 (9)	**27**	23 (4)
	mprime	35	19 (14)	24 (9)	28 (5)	**33**	**33**	25 (8)
	mystery	30	**29**	28 (1)	18 (11)	19 (10)	21 (8)	19 (10)
	total	125	79 (10)	82 (7)	61 (28)	75 (14)	**89**	72 (17)
	% solved		63.2%	65.6%	48.8%	60.0%	**71.2%**	57.6%
2	blocks	60	37 (1)	**38**	15 (23)	34 (4)	31 (7)	35 (3)
	elevator	150	43 (9)	44 (8)	20 (32)	33 (19)	46 (6)	**52**
	freecell	60	12 (19)	12 (19)	16 (15)	14 (17)	**31**	19 (12)
	logistics	198	38 (7)	42 (3)	29 (16)	27 (18)	44 (1)	**45**
	total	468	130 (22)	136 (16)	80 (72)	108 (44)	**152**	151 (1)
	% solved		27.8%	29.1%	17.1%	23.1%	**32.5%**	32.3%
3	depots	22	15 (1)	15 (1)	12 (4)	15 (1)	15 (1)	**16**
	driverlog	20	15 (2)	15 (2)	12 (5)	15 (2)	**17**	16 (1)
	freecell	20	3 (3)	3 (3)	5 (1)	4 (2)	**6**	4 (2)
	rovers	20	13 (5)	13 (5)	15 (3)	**18**	16 (2)	15 (3)
	zenotravel	20	15 (1)	**16**	12 (4)	14 (2)	**16**	15 (1)
	total	102	61 (9)	62 (8)	56 (14)	66 (4)	**70**	66 (4)
	% solved		59.8%	60.8%	54.9%	64.7%	**68.6%**	64.7%
4	pipesworld-notankage	50	16 (24)	16 (24)	26 (14)	19 (21)	**40**	37 (3)
	pipesworld-tankage	50	8 (18)	8 (18)	10 (16)	10 (16)	**26**	16 (10)
	psr-small	50	49 (1)	49 (1)	**50**	49 (1)	**50**	**50**
	total	150	73 (43)	73 (43)	86 (30)	78 (38)	**116**	103 (13)
	% solved		48.7%	48.7%	57.3%	52.0%	**77.3%**	68.7%
5	openstacks	30	0 (5)	0 (5)	0 (5)	0 (5)	**5**	**5**
	pathways	30	5 (4)	8 (1)	**9**	7 (2)	5 (4)	**9**
	tpp	30	17 (6)	**23**	20 (3)	14 (9)	16 (7)	20 (3)
	trucks	30	2 (8)	2 (8)	3 (7)	3 (7)	**10**	5 (5)
	total	120	24 (15)	33 (6)	32 (7)	24 (15)	36 (3)	**39**
	% solved		20.0%	27.5%	26.7%	20.0%	30.0%	**32.5%**
6	elevators	30	5 (9)	5 (9)	5 (9)	4 (10)	**14**	10 (4)
	openstacks	30	**3**	**3**	**3**	**3**	**3**	**3**
	parcprinter	30	27 (3)	28 (2)	28 (2)	25 (5)	**30**	29 (1)
	pegsol	30	10 (14)	10 (14)	11 (13)	18 (6)	**24**	19 (5)
	scanalyzer	30	14 (4)	15 (3)	12 (6)	17 (1)	**18**	14 (4)
	transport	30	9 (4)	9 (4)	6 (7)	6 (7)	**13**	11 (2)
	woodworking	30	**30**	**30**	**30**	**30**	5 (25)	**30**
	total	210	98 (18)	100 (16)	95 (21)	103 (13)	107 (9)	**116**
	% solved		46.7%	47.6%	45.2%	49.0%	51.0%	**55.2%**
total		1175	465 (105)	486 (84)	410 (160)	454 (116)	**570**	547 (23)
% solved			39.6%	41.4%	34.9%	38.6%	**48.5%**	46.6%

Table 1: Number and percentage of solved problems in selected parallel domains of the IPCs from 1998 to 2008. Numbers in bold indicate the best results and numbers in parenthesis indicate the number of unsolved problems with respect to the best result.

Parallel Planning

The first version of CPT had been ranked second in the 4th, after SATPLAN'04; since then, SAT-based planners have been greatly improved. The first reason is that SAT solvers are more and more efficient; and the second, because better encodings have been found. We compare six state-of-the-art planners on parallel planning problems: MAXPLAN (Xing, Chen, and Zhang 2006), Mp (Rintanen 2010), SASE (Huang, Chen, and Zhang 2010), SATPLAN'06 (Kautz, Selman, and Hoffmann 2006), and the last two versions of CPT. We took many domains from the 1st to the 6th IPC, for a total of 1175 planning problems. The domains that do not appear in these results (e.g. airport and sokoban) make the available version of SASE (v0.1) crash: to be fair, we have not included them. All planners optimize the parallel plan length.

As can be seen from Table 1, the recent SASE planner, which is based on a SAS+ encoding, is the most efficient one: it solves 570 problems (48.5%). SATPLAN'06 is the next best planner, with 547 problems solved (46.6%). Then come CPT4 and CPT3, which solve 486 problems (41.4%) and 465 problems (39.6%) respectively. Finally, comes Mp with 454 solved problems (38.6%) and MAXPLAN with 410 problems (34.9%). Although CPT4 is not the most efficient parallel planner, it is able to outperform some SAT-based planners on this set of benchmarks. However, the difference between CPT and the best SAT-based planner is perhaps higher than what it was a few days ago: CPT has not evolved a lot over the last few years.

Figure 1 shows the cumulated number of solved problems in function of the total running time. For each CPU time t on the x axis, the corresponding value on the y axis gives the number of problems solved in under t seconds. We can

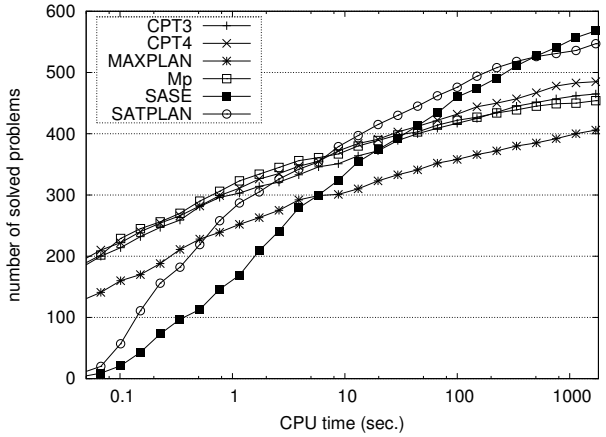

Figure 1: Cumulated number of solved problems in function of the search time for parallel planners.

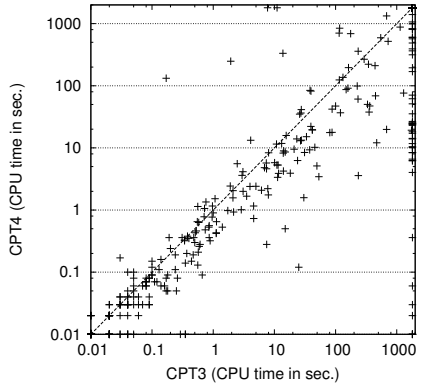

Figure 3: Comparison of the total running time between CPT3 and CPT4 on temporal planning problems.

see that CPT3, CPT4 and Mp solve more problems than the other planners with very small running times, but solve significantly fewer problems that require longer running time. Figure 2, which makes pairwise comparisons between CPT4 and the other planners, confirms this view.

Temporal Planning

CPT3 and CPT4 are then compared on all temporal problems without numerical fluents from all past IPCs, for a total of 664 planning problems. Table 2 shows the number of solved problems within the time limit. CPT4, which solves 316 problems (47.6%), clearly outperforms CPT3 which solves 271 problems (40.8%). Only one problem in the domain pipesworld-tankage is solved by CPT3 and not by CPT4. Figure 3, which compares the total running time of both planners, shows that CPT4 generally outperforms CPT3.

Acknowledgments

This work has been supported by the French National Research Agency (ANR) through COSINUS program (project DESCARWIN n°ANR-09-COSI-002).

IPC	domain	#pbs	#solved CPT3	#solved CPT4
3	depots	22	8 (2)	**10**
	driverlog	20	**11**	**11**
	rovers	20	5 (1)	**6**
	satellite	20	12 (1)	**13**
	zenotravel	20	**14**	**14**
	total	102	50 (4)	**54**
	% solved		49.0%	**52.9%**
4	airport	50	41 (3)	**44**
	airport-timewindows	50	34 (10)	**44**
	pipesworld-notankage-deadlines	30	14 (4)	**18**
	pipesworld-notankage	50	15 (2)	**17**
	pipesworld-tankage	50	**9**	8 (1)
	satellite-time	36	**17**	**17**
	satellite-time-timewindows	36	7 (5)	**12**
	total	302	137 (23)	**160**
	% solved		45.4%	**53.0%**
5	openstacks	20	**0**	**0**
	storage	30	**15**	**15**
	trucks	30	2 (5)	**7**
	total	80	17 (5)	**22**
	% solved		21.2%	**27.5%**
6	crewplanning	30	5 (10)	**15**
	elevators	30	**2**	**2**
	openstacks	30	**4**	**4**
	parcprinter	30	22 (3)	**25**
	pegsol	30	**29**	**29**
	sokoban	30	**5**	**5**
	total	180	67 (13)	**80**
	% solved		37.2%	**44.4%**
total		664	271 (45)	**316**
% solved			40.8%	**47.6%**

Table 2: Number and percentage of solved problems in all temporal domains of the IPCs from 1998 to 2008. Numbers in bold indicate the best results and numbers in parenthesis indicate the number of unsolved problems with respect to the best result.

References

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proc. ECAI*, 146–150.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proc. ECP*, 121–132.

Haslum, P. 2004. TP4 '04 and HSP^*_a. In *Booklet of the 4th IPC*.

Huang, R.; Chen, Y.; and Zhang, W. 2010. A novel transition based encoding scheme for planning as satisfiability. In *Proc. AAAI*.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. SATPLAN: Planning as satisfiability. In *Booklet of the 5th IPC*.

Lecoutre, C.; Saïs, L.; Tabary, S.; and Vidal, V. 2009. Reasoning from last conflict(s) in Constraint Programming. *Artificial Intelligence* 173(18):1592–1614.

Rintanen, J. 2007. Complexity of concurrent temporal planning. In *Proc. ICAPS*, 280–287.

Rintanen, J. 2010. Heuristic planning with SAT: Beyond

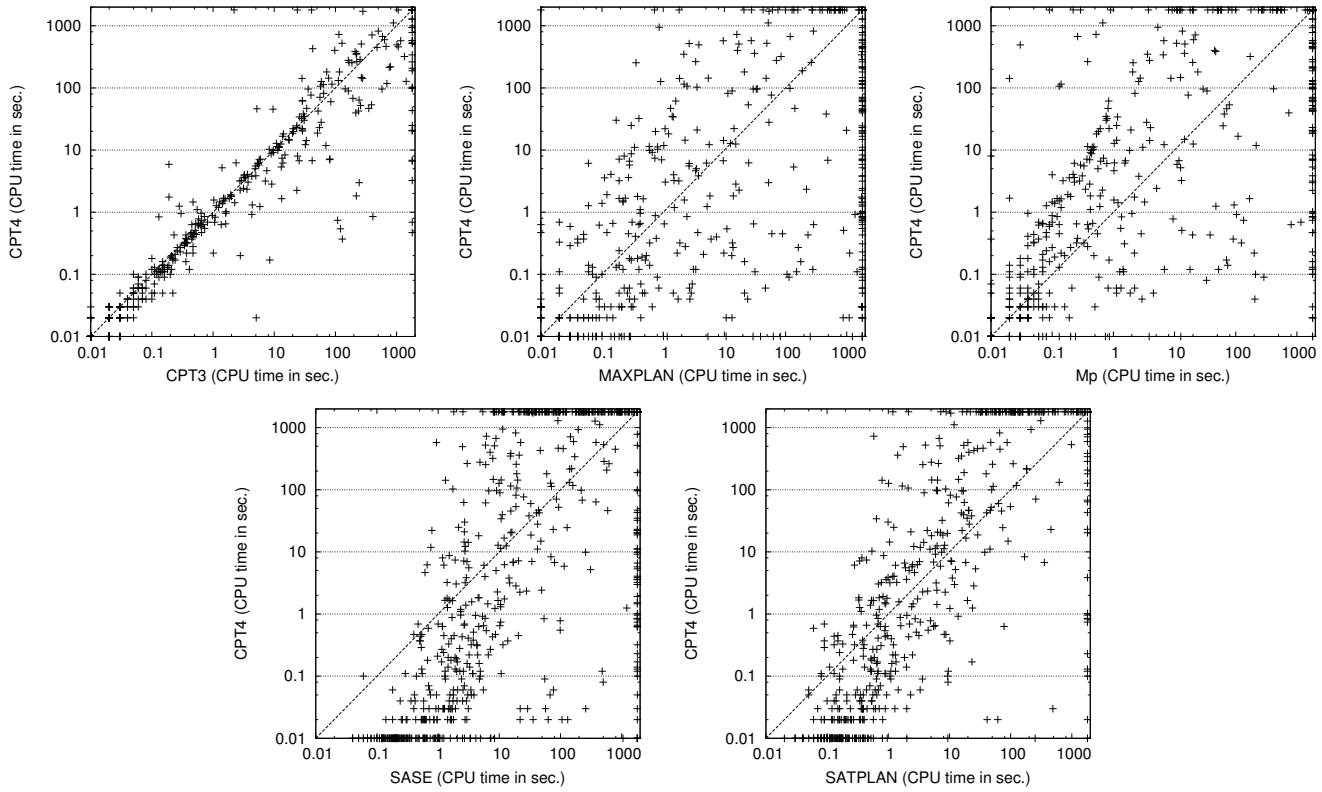

Figure 2: Comparison of the total running time between CPT4 and all other parallel planners.

uninformed depth-first search. In *Proc. Australasian Conf. on AI*, 415–424.

Smith, D. E., and Weld, D. S. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. IJCAI*, 326–337.

Vidal, V., and Geffner, H. 2005. Solving simple planning problems with more inference and no search. In *Proc. CP*, 682–696.

Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170(3):298–335.

Vidal, V., and Tabary, S. 2006. The new version of CPT, an optimal temporal POCL planner based on constraint programming. In *Booklet of the 5th IPC*.

Xing, Z.; Chen, Y.; and Zhang, W. 2006. Optimal strips planning by maximum satisfiability and accumulative learning. In *Proc. ICAPS*, 442–446.

Divide-and-Evolve: the Marriage of Descartes and Darwin

Johann Dréo **Pierre Savéant**
Thales Research & Technology
Palaiseau, France
first.last@thalesgroup.com

Marc Schoenauer
INRIA Saclay & LRI
Orsay, France
marc.schoenauer@inria.fr

Vincent Vidal
ONERA – DCSD
Toulouse, France
Vincent.Vidal@onera.fr

Abstract

DAE$_{\text{X}}$, the concrete implementation of the *Divide-and-Evolve* paradigm, is a domain-independent satisficing planning system based on Evolutionary Computation. The basic principle is to carry out a *Divide-and-Conquer* strategy driven by an evolutionary algorithm. The key components of DAE$_{\text{X}}$ are a state-based decomposition principle, an evolutionary algorithm to drive the optimization process, and an embedded planner X to solve the sub-problems. The release that has been submitted to the competition is DAE$_{\text{YAHSP}}$, the instantiation of DAE$_{\text{X}}$ with the heuristic forward search YAHSP planner. The marriage of DAE and YAHSP matches a clean role separation: YAHSP gets a few tries to find a solution quickly whereas DAE controls the optimization process.

Introduction

DAE$_{\text{X}}$, the concrete implementation of the *Divide-and-Evolve* paradigm, is a domain-independent satisficing planning system based on Evolutionary Computation (Schoenauer, Savéant, and Vidal 2006). The basic principle is to carry out a *Divide-and-Conquer* strategy driven by an evolutionary algorithm. The algorithm is detailed in (Bibaï et al. 2010a) and compared with state-of-the-art planners. In order to solve a planning task $\mathcal{P}_D(I, G)$, the basic idea of DAE$_{\text{X}}$ is to find a sequence of goals $S_1, \ldots, S_n$, and to rely on an embedded planner X to solve the series of planning tasks $\mathcal{P}_D(S_k, S_{k+1})$, for $k \in [0, n]$ (with $S_0 = I$ and $S_{n+1} = G$). A DAE$_{\text{X}}$ individual is thus a sequence of goals which define a sequence of subproblems to be solved (a *decomposition*). These subproblems are submitted successively to an embedded planner X and the global solution is obtained after the compression of these intermediate solutions. The overall optimization process is controlled by an evolutionary algorithm.

The decomposition principle of DAE$_{\text{X}}$ is very general and could be applied to any type of planning tasks. The scope of the planner is of course the one of the embedded planner X. The release that has been submitted to the competition is DAE$_{\text{YAHSP}}$, the instantiation of DAE$_{\text{X}}$ with the heuristic forward search YAHSP planner (Vidal 2004; 2011). The target is thus temporal satificing planning with conservative semantics, cost planning and classical STRIPS planning. The marriage of DAE and YAHSP matches a clean role separation: YAHSP gets a few tries to find a solution quickly whereas DAE controls the optimization process. In the current release we have introduced an initial estimation processing of the maximum number of tries allowed to YAHSP for all individual evaluations. This parameter is crucial for the time consumption of the algorithm.

The Evolutionary Engine

Figure 1 depicts the standard evolutionary loop which mimics a biological evolution. The fitness implements a gradient towards feasibility for unfeasible individuals and a gradient towards optimality for feasible individuals. Feasible individuals are always preferred to unfeasible ones. Population initialization as well as variation operators are driven by the critical path h^1 heuristic (Haslum and Geffner 2000) in order to discard inconsistent state orderings, and atom mutual exclusivity inference in order to discard inconsistent states. These two computations are done by YAHSP in an initial phase.

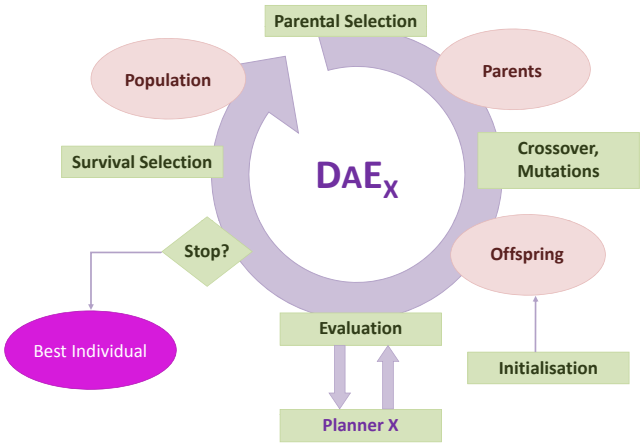

Figure 1: The standard evolutionary loop

Beside a standard one-point crossover for variable length representations, four mutations have been defined: addition

(resp. removal) of a goal in a sequence, addition (resp. removal) of an atom in a goal. Variation operators relax the strictly h^1 ordering of atoms within individuals, since it is only a heuristic estimate.

The selection is a comparison-based deterministic tournament of size 5.

For the sequential release, Darwinian-related parameters of DAE$_X$ have been fixed after some early experiments (Schoenauer, Savéant, and Vidal 2006) whereas parameters related to the variation operators have been tuned using the Racing method (Bibaï et al. 2010b). It should be noted that, due to the conditions of the competition, the parameter setting is global to all domains. In (Bibaï et al. 2010b) we showed that a specific tuning for an instance provides better results as expected and that what we would do for a real-life planning task.

We added two novelties to the version described in (Bibaï et al. 2010a). One important parameter is the maximum number of expanded nodes allowed to the YAHSP subsolver which defines empirically what is considered as an easy problem for YAHSP. As a matter of fact, the minimum number of required nodes varies from few nodes to thousands depending of the planning task. In the current release this number is estimated during the population initialization stage. An incremental loop is performed until the ratio of feasible individuals is over a given threshold or a maximum boundary has been reached. By default this number is doubled at each iteration until at least one feasible individual is produced or 100000 has been reached.

Furthermore we add the capability to perform restarts within a time contract in order to increase solution quality.

The fitness used for the competition differs from the one described in (Bibaï et al. 2010a). The fitness for bad individuals has been simplified by withdrawing the Hamming distance to the goal. The new fitness depends only on the "decomposition distance": the number of intermediate goals reached and more specifically the one that are "useful". A useful intermediate goal is a goal that require a non-empty plan to be reached.

Implementation

The implementation of DAE$_X$ has been made with the STL-based Evolving Objects framework[1] which provides an abstract control structure to develop any kind of evolutionary algorithm in C++. YAHSP is written in the C language.

In order to speed up search, a memozation mechanism has been introduced in YAHSP and carefully controlled to leave memory space for DAE. Indeed, most of the time during a run of YAHSP, and as a consequence during a run of DAE$_{YAHSP}$, is spent in computing the h^{add} heuristic for each encountered state (see (Vidal 2011) for more details about the algorithms of the new version of the YAHSP planner). During a single run of YAHSP, duplicate states are discarded; but during a run of DAE$_{YAHSP}$, the same state can be encoutered multiple times. We therefore keep track of the h^{add} costs of all atoms in the problem for each state, in order to avoid recomputing these values each time a duplicate state is reached. This generally leads to a speedup comprised between 2 and 4. When DAE$_{YAHSP}$ runs out of memory, which obviously happens much faster with the memoization strategy, all stored states and associated costs are flushed. More sophisticated strategies may be implemented, e.g. flushing the oldest or less often encountered states; but we found that the simplest solution of completely freeing the memoized information was efficient enough.

Several biases have been introduced in YAHSP, in order to help DAE$_{YAHSP}$ finding better solutions. The main one is that actions of lower duration are preferred to break ties between several actions of same h^{add} cost, when computing relaxed plans and performing the relaxed plan repair strategy. Another bias is that the cost incrementation made during h^{add}, which is usually equal to 1 for each applied action, is made equal to either the duration or the cost of the action. Although these biases do not change a lot the quality of the plans produced by YAHSP alone, we found that they are of better help to DAE$_{YAHSP}$. However, introducing such biases is not very satisfactorily; it would be better to exactly use the version described in (Vidal 2011). We still have to better investigate the relationships between the evolutionnary engine and the embedded planner, in order to determine how to manage such kind of biases and other tie-breaking strategies.

[1] `http://eodev.sf.net`

Acknowledgments

This work is being partially funded by the French National Research Agency (ANR) through the COSINUS programme, under the research contract DESCARWIN (ANR-09-COSI-002).

References

Bibaï, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010a. An Evolutionary Metaheuristic Based on State Decomposition for Domain-Independent Satisficing Planning. In 20^{th} *International Conference on Automated Planning and Scheduling (ICAPS-2010)*, 18–25. AAAI Press.

Bibaï, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010b. On the Generality of Parameter Tuning in Evolutionary Planning. In 20^{th} *Genetic and Evolutionary Computation Conference (GECCO'10)*, 241–248. ACM Press.

Haslum, P., and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *AIPS-2000*, 70–82.

Schoenauer, M.; Savéant, P.; and Vidal, V. 2006. Divide-and-Evolve: a New Memetic Scheme for Domain-Independent Temporal Planning. In Gottlieb, J., and Raidl, G., eds., 6^{th} *European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP'06)*. Springer Verlag.

Vidal, V. 2004. A Lookahead Strategy for Heuristic Search Planning. In 14^{th} *International Conference on Automated Planning and Scheduling (ICAPS-2004)*, 150–159. AAAI Press.

Vidal, V. 2011. YAHSP2: Keep It Simple, Stupid. In 7^{th} *International Planning Competition (IPC-2011), Deterministic Part*.

FD-Autotune: Automated Configuration of Fast Downward

Chris Fawcett
University of British Columbia
fawcettc@cs.ubc.ca

Malte Helmert
Albert-Ludwigs-Universität Freiburg
helmert@informatik.uni-freiburg.de

Holger Hoos
University of British Columbia
hoos@cs.ubc.ca

Erez Karpas
Technion
karpase@technion.ac.il

Gabriele Röger
Albert-Ludwigs-Universität Freiburg
roeger@informatik.uni-freiburg.de

Jendrik Seipp
Albert-Ludwigs-Universität Freiburg
seipp@informatik.uni-freiburg.de

Abstract

The FD-Autotune submissions for the IPC-2011 sequential tracks consist of three instantiations of the latest, highly parametric version of the Fast Downward Planning Framework. These instantiations have been automatically configured for performance on a wide range of planning domains, using the well-known ParamILS configurator. Two of the instantiations were entered into the sequential satisficing track and one into the sequential optimising track. We describe how the extremely large configuration space of Fast Downward was restricted to a subspace that, although still very large, can be managed by state-of-the-art automated configuration procedures, and how ParamILS was then used to obtain performance-optimised configurations.

Introduction

Developers of state-of-the-art, high-performance algorithms for combinatorial problems, such as planning, are frequently faced with many interdependent design choices. These choices can include the heuristics to use during search, options controlling the behaviour of these heuristics, as well as which search techniques to use and in what combination.

Recent work in other combinatorial problem domains such as satisfiability (SAT) and mixed-integer programming (MIP) suggests that by exposing these design choices as parameters, developers can leverage generic tools for automated algorithm configuration to find performance-optimising configurations of the resulting highly parameterised algorithm (Hutter et al. 2007; Hutter, Hoos, and Leyton-Brown 2010). In fact, the configurations resulting from this process often perform substantially better than those found manually through exploration by human experts.

These results suggest the following new approach to building planning algorithms. Given a highly-parametric, general purpose planner P, a representative set I of planning instances from one or more domains, and a performance metric m to be optimised, we can obtain a configuration of the parameters of P optimised for performance on I with respect to m using a generic automated algorithm configuration tool.

For this submission, we applied the above approach using a new, highly-parameterised version of the Fast Downward planning system (Helmert 2006) and the state-of-the-art automated algorithm configurator ParamILS (Hutter, Hoos, and Stützle 2007; Hutter et al. 2009) to obtain three configurations of Fast Downward: FD-Autotune-satisficing.1, FD-Autotune-satisficing.2, and FD-Autotune-optimizing. FD-Autotune-satisficing.1 was optimised using mean plan cost after a fixed runtime as the optimisation metric, while the FD-Autotune-optimizing configurations were obtained by using mean runtime to find an optimal plan. FD-Autotune-satisficing.2 is a hybrid planner obtained by inserting a phase at the beginning of search that has been configured to find satisficing plans as quickly as possible. Due to the highly structured and potentially infinite configuration space of Fast Downward, we carefully limited the number of parameters in order to comply with the requirements of ParamILS and to retain as many potential planner configurations as possible. All of our configuration and analysis experiments were performed using HAL, a recently developed software environment for work in empirical algorithmics (Nell et al. 2011).

The Fast Downward Planning Framework

In this section we describe the capabilities of the IPC-2011 version of the Fast Downward planning system. Since Fast Downward incorporates many different algorithms and approaches, which have each been published separately in peer-reviewed conferences and/or journals, we will simply list the available components with pointers to further information for the interested reader.

The Fast Downward planning system (Helmert 2006) is composed of three main parts: the translator, the preprocessor, and the search component, which are run sequentially in this order. The translator (Helmert 2009) is responsible for translating the given PDDL task into an equivalent one in SAS^+ representation. This is done by finding groups of propositions which are mutually exclusive and combining them into a single SAS^+ variable. The preprocessor performs a relevance analysis and precomputes some data structures that are used by the search and certain heuristics. The search component, whose capabilities we will describe in detail here, searches for a solution to the given SAS^+ task.

Search

The search component features three main types of search algorithms:

- Eager Best-First Search — the classic best-first search. The same search code is used for greedy best-first search, A^*, and weighted A^* by plugging in different f functions. The multi-path-dependent *LM-A** (Karpas and Domshlak 2009) is also implemented here.
- Lazy Best-First Search — this is best-first search with deferred evaluation (Richter and Helmert 2009). Here as well, the same search code is used for lazy greedy best-first search and lazy weighted A^* by using a different f function.
- Enforced Hill-Climbing (Hoffmann and Nebel 2001) — an incomplete local search technique. This has been slightly generalised from classic EHC to allow preferred operators from multiple heuristics, as well as enabling or disabling preferred operator pruning.

Each of these search algorithms can take several parameters and use one or more heuristics (heuristic combination methods will be discussed next). In addition, these algorithms can be run in an iterated fashion. This can be used, for example, to produce *RWA** (Richter, Thayer, and Ruml 2010), the search algorithm used in LAMA (Richter and Westphal 2010).

Heuristic Combination

As mentioned previously, the search algorithms described above can work with multiple heuristic evaluators. There are several heuristic combination methods available in the Fast Downward planning system, which are implemented as different kinds of *open lists*.

Some of these combination methods amount to simple arithmetic combinations of heuristic values and can use a standard ("regular") open list implementation, while others treat the different heuristic estimates $\langle h_1(s), \ldots, h_n(s)\rangle$ as a vector that is not reduced to a single scalar value (Röger and Helmert 2010).[1] As a result, some of these latter methods do not necessarily induce a total order on the set of open states. The following combination methods are available in Fast Downward, in addition to performing a regular search using a single heuristic:

- Max — the maximum of several heuristic estimates: $\max\{h_1(s), \ldots, h_n(s)\}$.
- Sum — the sum or weighted sum of several heuristic estimates: $w_1 h_1(s) + \cdots + w_n h_n(s)$.
- Selective Max (Domshlak, Karpas, and Markovitch 2010) — a learning-based method which chooses one heuristic to evaluate at each state: $h_i(s)$ where i is chosen on a per-state basis using a naive Bayes classifier trained on-line.
- Tie-breaking — considers the heuristics in fixed order: first, consider $h_1(s)$; if ties need to be broken, consider $h_2(s)$; and so on.
- Pareto-optimal — all states whose heuristic value vector is not Pareto-dominated by another heuristic value vector are candidates for expansion, with selection between multiple candidates performed randomly.
- Alternation (Dual Queue) — heuristics are used in round-robin fashion: the first expansion uses $h_1(s)$, the second uses $h_2(s)$, etc., up to $h_n(s)$, followed again by $h_1(s)$. Alternation can also be enhanced by *boosting* (Richter and Helmert 2009).

Each combination method can take several parameters. One important parameter is whether the open list contains only states which have been reached via preferred operators, or all states.

Moreover, wherever this makes sense, instead of using different *heuristics* as their components, these combination methods can also combine the results of different *open lists* which can themselves employ combination methods, and this nesting can even be performed recursively. For example, it is possible to use alternation over one regular heuristic, one Pareto-based open list, and one open list that uses tie-breaking over various weighted sums.

Such combinations allow us to build the 'classic' boosted dual queue of Fast Downward: use an alternation approach, which combines two standard open lists, one of which holds all states, and the other only preferred states, both of which are based on a single heuristic estimate. To use two heuristic estimates as in Fast Diagonally Downward (Helmert 2006) or LAMA (Richter and Westphal 2010), alternation over four open lists would be used (for each heuristic, one holding all states and one holding only preferred states).

Heuristics

So far, we have discussed the search algorithms and heuristic combination methods available in the Fast Downward planning system. We now turn our attention to the heuristics available in Fast Downward. Due to the number of heuristics, we simply list the available heuristics, with pointers to relevant literature.

Admissible Heuristics

- Blind — 0 for goal states, 1 (or cheapest action cost for non-unit-cost tasks) for non-goal states
- $h^{\max}$ (Bonet, Loerincs, and Geffner 1997; Bonet and Geffner 1999) — the relaxation-based maximum heuristic
- h^m (Haslum and Geffner 2000) — a very slow implementation of the h^m heuristic family
- $h^{\text{M\&S}}$ (Helmert, Haslum, and Hoffmann 2007; 2008) — the merge-and-shrink heuristic
- h^{LA} (Karpas and Domshlak 2009; Keyder, Richter, and Helmert 2010) — the admissible landmark heuristic
- $h^{\text{LM-cut}}$ (Helmert and Domshlak 2009) — the landmark-cut heuristic

[1] To simplify discussion, this description assumes that search algorithm behaviour only depends on heuristic values, but all these algorithms can also take into account path costs, as in A^* or weighted A^*.

Algorithm	Categorical	Numeric	Total	Configurations
FD-Autotune-satisficing	64	13	77	1.935×10^{26}
FD-Autotune-optimizing	20	6	26	6.99×10^{8}

Table 1: The number of categorical and numeric parameters in the reduced configuration space for both FD-Autotune-satisficing and FD-Autotune-optimizing, as well as the total number of distinct configurations for each.

Inadmissible Heuristics

- Goal Count — number of unachieved goals
- h^{add} (Bonet, Loerincs, and Geffner 1997; Bonet and Geffner 1999) — the relaxation-based additive heuristic
- h^{FF} (Hoffmann and Nebel 2001) — the relaxed plan heuristic
- h^{cg} (Helmert 2004) — the causal graph heuristic
- h^{cea} (Helmert and Geffner 2008) — the context-enhanced additive heuristic (a generalisation of h^{add} and h^{cg})
- h^{LM} (Richter, Helmert, and Westphal 2008; Richter and Westphal 2010) — the landmark heuristic

Apart from Goal Count, all heuristics listed above are cost-based versions (that is, they support non-unit cost actions). This also allows another option for these heuristics: action-cost adjustment. It is possible to instruct the heuristics (as well as the search code) to treat all actions as unit-cost (regardless of their true cost) or to add 1 to all action costs. This has been found to be helpful in tasks with 0-cost actions (Richter and Westphal 2010).

Configuration Space

The configuration space of Fast Downward poses a challenge in formulating the parameter space to be explored by an automated configurator: structured parameters. For example, it is possible to configure an 'alternation' open list that alternates between two internal alternation open lists, each of which alternates between their own internal alternation open lists, and so on. Since neither ParamILS (Hutter et al. 2007) nor any other configuration procedure we are aware of handles such structured parameters, we had to limit the configuration space somewhat.

The configuration spaces used for the competition (as shown in Tables 2, 3, and 4) contain a Boolean parameter for each heuristic (all heuristics for satisficing planning, only admissible heuristics for optimal planning), indicating whether that heuristic is in use or not. The other parameters of the heuristic (if any) are conditional on the heuristic being used.

For optimal planning, the search algorithm is predetermined (A^*), and so our only other choice is, when more than one heuristic is used, how the heuristics are combined (the relevant options are *max* and *selective max*). This is controlled by another parameter, which is conditional on more than one heuristic being chosen.

For satisficing planning, the theoretical configuration space is much more complex, since combination methods such as alternation and weighted sums introduce an infinite set of possibilities.

To keep this configuration space manageable, we only allow one layer of alternation, and its components must be standard open lists (sorted by scalar ranking values), one for each heuristic that was selected, and possibly more if preferred operators are used. In addition, we can combine search algorithms using iterated search as in *RWA**. Here, we limit the number of searches to a maximum of 5, in order to avoid an infinitely large structured configuration space. As shown in Table 1, FD-Autotune-optimizing and FD-Autotune-satisficing have many parameters, with 6.99×10^8 and 1.935×10^{26} distinct configurations, respectively. The configuration space of FD-Autotune-satisficing is one of the largest ever experimented with using automated algorithm configuration tools.

Automated Configuration

For the configuration task faced in the context of this work, we chose to use the FocusedILS variant of ParamILS (Hutter, Hoos, and Stützle 2007; Hutter et al. 2009), because it is the only procedure we are aware of that has been demonstrated to perform well on algorithm configuration problems as hard as the one encountered here. ParamILS is fundamentally based on Iterated Local Search (ILS), a well-known, general stochastic local search method that interleaves phases of simple local search – in particular, iterative improvement – with so-called perturbation phases that are designed to escape from local optima.

In the FocusedILS variant of ParamILS, ILS is used to search for high-performance configurations of a given target algorithm (here: Fast Downward) by evaluating promising configurations. To avoid wasting CPU time on poorly-performing configurations, FocusedILS carefully controls the number of target algorithm runs performed for candidate configurations; it also adaptively limits the amount of runtime allocated to each algorithm run using knowledge of the best-performing configuration found so far. Further information on ParamILS can be found in earlier work by Hutter, Hoos, and Stützle (2007) and Hutter et al. (2009), and interesting applications have been reported by Hutter et al. (2007), and Hutter, Hoos, and Leyton-Brown (2010).

Experiments and Configurations

For all experiments performed in this work, we took advantage of the features in HAL, a recently developed tool to support both the computer-aided design and the empirical analysis of high-performance algorithms (Nell et al. 2011). We used several meta-algorithmic procedures provided by HAL, primarily the ParamILS algorithm configurator and the plug-ins providing support for empirical analysis of one or two algorithms. We also leveraged the robust support in HAL for data management and run distribution on compute clusters. All experiments were performed using a cluster of identical linux-based machines, each with an Intel Xeon E5450 quad-core processor and 6GB of RAM.

For optimising planning, we used HAL to run ten independent runs of ParamILS on a set of 2000 training instances

sampled uniformly at random from the IPC-2008 optimisation track instances located in the Fast Downward benchmark set [2], using a maximum runtime cutoff of 600 CPU seconds for each run of Fast Downward and a total configuration time limit of four CPU days. The objective function used in ParamILS was mean runtime to find an optimal plan, with timed-out runs penalised at 10 times the runtime cutoff. In this case, we were also able to leverage support in ParamILS for adaptive runtime capping to reduce the runtime required for each run of Fast Downward. Out of the ten incumbent configurations produced by ParamILS, we selected the configuration with the best reported training quality to be FD-Autotune-optimizing. The exact parameter values for this configuration of Fast Downward are shown in Table 2.

For satisficing planning, we again performed ten independent runs of ParamILS on a set of 2000 training instances sampled from the entire Fast Downward benchmark set, with instances from the IPC-2008 satisficing track having twice the probability of being selected. We again used a runtime cutoff of 600 CPU seconds for each run of Fast Downward with a total configuration time limit of four CPU days. Here, the objective function used in ParamILS was mean plan cost, with runs that failed to find a satisficing solution assigned a (dummy) plan cost of $2^{31} - 1$ (the default value for solution quality in HAL). Again, we selected the configuration with the best reported training quality to be FD-Autotune-satisficing.1. The exact parameter values for this configuration are shown in Tables 3 and 4.

Finally, we performed an additional ten independent runs of ParamILS using the satisficing planning training set, on a reduced design space containing the parameters from Table 3 and only a single set of search parameters from Table 4. We again used a runtime cutoff of 600 CPU seconds for each run of Fast Downward with a total configuration time limit of four CPU days. The objective function used was mean runtime to find an initial satisficing plan, and Fast Downward was configured to terminate after this solution was found. The configuration with the best reported training quality was selected to form the basis of a hybrid version of the planner, where Fast Downward uses this configuration to find an initial satisficing plan, at which point the FD-Autotune-satisficing configuration with second-best training quality is used. We call this hybrid scheme FD-Autotune-satisficing.2. The parameter values for both phases of FD-Autotune-satisficing.2 are indicated in Tables 3 and 4.

Conclusions and Future Work

We believe that the generic approach underlying our work on FD-Autotune represents a promising direction for the future development of efficient planning systems. In particular, we suggest that it is worth including many different variants and a wide range of settings for the various components of a planning system, instead of committing at design time to particular choices and settings, and to use automated procedures for finding configurations of the resulting highly parameterised planning systems that perform well on the problems arising in a specific application domain, or set of domains, under consideration. We plan to further investigate ways in which automated algorithm configurators, such as ParamILS, can deal more effectively with the highly structured and potentially infinite space of Fast Downward.

We note that our approach naturally benefits from future improvements in planning systems (and in particular, from new heuristic ideas that can be integrated, in the form of parameterised components, into existing, flexible planning systems or frameworks) as well as from progress in automated algorithm configuration procedures. In principle, planning systems developed in this way can also be used in combination with techniques for automated algorithm selection, giving even greater performance than any single configuration alone (Xu et al. 2008; 2009; Xu, Hoos, and Leyton-Brown 2010). We also see much potential in testing new heuristics and algorithm components, based on measuring the performance improvements obtained by adding them to an existing highly-parameterised planner followed by automatic configuration. The results of such experiments should indicate to which extent new design elements are useful and also reveal under which circumstances they are most effective.

Acknowledgements The authors would like to thank WestGrid and Compute-Calcul Canada for providing access to the cluster hardware used in our experiments.

[2] This benchmark repository is part of the main Fast Downward repository, accessed from http://www.fast-downward.org

References

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *ECP*, 360–372.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *AAAI*, 714–719.

Domshlak, C.; Karpas, E.; and Markovitch, S. 2010. To max or not to max: Online learning for speeding up optimal planning. In *AAAI*, 1071–1076.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *AIPS*, 140–149.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 162–169.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *ICAPS*, 140–147.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2008. Explicit-state abstraction: A new method for generating heuristic functions. In *AAAI*, 1547–1550.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *ICAPS*, 161–170.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5–6):503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hutter, F.; Babic, D.; Hoos, H. H.; and Hu, A. J. 2007. Boosting verification by automatic tuning of decision procedures. *Formal Methods in Computer-Aided Design* 27–34.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2010. Automated configuration of mixed integer programming solvers. In Lodi, A.; Milano, M.; and Toth, P., eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*. Springer. 186–202.

Hutter, F.; Hoos, H. H.; and Stützle, T. 2007. Automatic algorithm configuration based on local search. In *AAAI*, 1152–1157.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *IJCAI*, 1728–1733.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In *ECAI*, 335–340.

Nell, C.; Fawcett, C.; Hoos, H. H.; and Leyton-Brown, K. 2011. HAL: A framework for the automated analysis and design of high-performance algorithms. In *LION-5*. To appear.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *ICAPS*, 273–280.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *AAAI*, 975–982.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In *ICAPS*, 137–144.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *ICAPS*, 246–249.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research* 32:565–606.

Xu, L.; Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2009. SATzilla2009: an automatic algorithm portfolio for SAT. Solver description, SAT competition 2009.

Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, 210–216.

Parameter name	Domain	Default	FD-Autotune-optimizing
blind_heuristic_enabled	$\{true, false\}$	$false$	$false$
hm_heuristic_enabled	$\{true, false\}$	$false$	$false$
hm_heuristic_m	$\{1, 2, 3\}$	2	–
hmax_heuristic_enabled	$\{true, false\}$	$false$	$true$
lm_heuristic_enabled	$\{true, false\}$	$true$	$false$
lm_heuristic_conjunctive_landmarks	$\{true, false\}$	$true$	–
lm_heuristic_disjunctive_landmarks	$\{true, false\}$	$true$	–
lm_heuristic_hm_m	$\{1, 2, 3\}$	1	–
lm_heuristic_no_orders	$\{true, false\}$	$false$	–
lm_heuristic_only_causal_landmarks	$\{true, false\}$	$false$	–
lm_heuristic_type	$\{lm\_rhw, lm\_zg, lm\_hm, lm\_exhaust, lm\_rhw\_hm1\}$	lm_rhw	–
lmcut_heuristic_enabled	$\{true, false\}$	$true$	$true$
mas_heuristic_enabled	$\{true, false\}$	$false$	$false$
mas_heuristic_max_states	{10 000, 50 000, 100 000, 150 000, 200 000}	50 000	–
mas_heuristic_merge_strategy	{5}	5	–
mas_heuristic_shrink_strategy	{4, 7, 6, 12}	4	–
combine_heuristics	$\{true, false\}$	$true$	$true$
combine_with_smax	$\{true, false\}$	$true$	$true$
smax_alpha	{0.0, 0.1, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 2.5, 3.0, 4.0, 5.0}	1.0	4.0
smax_classifier	{0, 1}	0	0
smax_conf_threshold	{0.51, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99}	0.9	0.85
smax_sample	{0, 2}	0	0
smax_training_set	{10, 50, 100, 500, 1 000, 5 000, 10 000}	1 000	10
smax_uniform	$\{true, false\}$	$false$	$true$
a_star_use_mpd	$\{true, false\}$	$true$	$false$
a_star_use_pathmax	$\{true, false\}$	$false$	$true$

Table 2: Configuration space for the optimising planner, comprising 26 parameters. Heuristic parameters are only active if the corresponding heuristic is enabled; $smax\_*$ are only active if $combine\_heuristics$ and $combine\_with\_smax$ are both $true$. "–" indicates that the given parameter is not active.

				FD-Autotune-satisficing.2	
Parameter name	Domain	Default	FD-Autotune-satisficing.1	$Phase1$	$Phase2$
add_heuristic_enabled	$\{true, false\}$	$false$	$false$	$false$	$true$
add_heuristic_cost_type	{0, 1, 2}	2	–	–	0
add_heuristic_pref_ops	$\{true, false\}$	$true$	–	–	$false$
blind_heuristic_enabled	$\{true, false\}$	$false$	$false$	$false$	$false$
cea_heuristic_enabled	$\{true, false\}$	$false$	$true$	$false$	$true$
cea_heuristic_cost_type	{0, 1, 2}	2	2	–	0
cea_heuristic_pref_ops	$\{true, false\}$	$true$	$true$	–	$true$
cg_heuristic_enabled	$\{true, false\}$	$false$	$true$	–	$true$
cg_heuristic_cost_type	{0, 1, 2}	2	1	–	2
cg_heuristic_pref_ops	$\{true, false\}$	$true$	$false$	–	$false$
ff_heuristic_enabled	$\{true, false\}$	$false$	$true$	$true$	$false$
ff_heuristic_cost_type	{0, 1, 2}	2	0	1	–
ff_heuristic_pref_ops	$\{true, false\}$	$true$	$false$	$true$	–
goalcount_heuristic_enabled	$\{true, false\}$	$false$	$true$	$false$	$true$
goalcount_heuristic_cost_type	{0, 1, 2}	2	2	–	0
goalcount_heuristic_pref_ops	$\{true, false\}$	$true$	$true$	–	$true$
hm_heuristic_enabled	$\{true, false\}$	$false$	$false$	$false$	$false$
hm_heuristic_m	{1, 2, 3}	2	–	–	–
hmax_heuristic_enabled	$\{true, false\}$	$false$	$false$	$false$	$false$
lm_ff_synergy	$\{true, false\}$	$true$	–	–	–
lm_heuristic_enabled	$\{true, false\}$	$true$	$false$	$false$	$false$
lm_heuristic_admissible	$\{true, false\}$	$false$	–	–	–
lm_heuristic_conjunctive_landmarks	$\{true, false\}$	$true$	–	–	–
lm_heuristic_cost_type	{0, 1, 2}	2	–	–	–
lm_heuristic_disjunctive_landmarks	$\{true, false\}$	$true$	–	–	–
lm_heuristic_hm_m	{1, 2, 3}	1	–	–	–
lm_heuristic_no_orders	$\{true, false\}$	$false$	–	–	–
lm_heuristic_only_causal_landmarks	$\{true, false\}$	$false$	–	–	–
lm_heuristic_pref_ops	$\{true, false\}$	$true$	–	–	–
lm_heuristic_reasonable_orders	$\{true, false\}$	$true$	–	–	–
lm_heuristic_type	$\{lm\_rhw, lm\_zg, lm\_hm, lm\_exhaust, lm\_rhw\_hm1\}$	lm_rhw	–	–	–
lmcut_heuristic_enabled	$\{true, false\}$	$true$	$false$	$false$	–
lmcut_heuristic_cost_type	{0, 1, 2}	0	$false$	–	–
mas_heuristic_enabled	$\{true, false\}$	$false$	$false$	$false$	–
mas_heuristic_max_states	{10 000, 50 000, 100 000, 150 000, 200 000}	50 000	–	–	–
mas_heuristic_merge_strategy	{5}	5	–	–	–
mas_heuristic_shrink_strategy	{4, 7, 6, 12}	4	–	–	–

Table 3: Parameters controlling heuristics in the configuration space for the satisficing planner, comprising 37 parameters. As with the optimising planner configuration space, the parameters for each heuristic are only active if the corresponding heuristic is enabled. "–" indicates that the given parameter is not active.

Parameter name	Domain	Default	FD-Autotune-satisficing.1	FD-Autotune-satisficing.2	
				$Phase1$	$Phase2$
search_0_cost_type	$\{0, 1\}$	0	0	1	1
search_0_eager_pathmax	$\{true, false\}$	$false$	$-$	$-$	$-$
search_0_ehc_preferred_usage	$\{0, 1\}$	0	0	$-$	$-$
search_0_search_boost	$\{0, 100, 200, 500, 1\,000, 2\,000, 5\,000\}$	0	$-$	2000	1000
search_0_search_open_list_tb	$\{true, false\}$	$false$	$-$	$false$	$false$
search_0_search_reopen	$\{true, false\}$	$false$	$-$	$false$	$false$
search_0_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, -1\}$	-1	$-$	10	7
search_0_type	$\{none, ehc, eager, lazy\}$	$lazy$	ehc	$lazy$	$lazy$
search_1_cost_type	$\{0, 1\}$	0	1	$-$	0
search_1_eager_pathmax	$\{true, false\}$	$false$	$-$	$-$	$-$
search_1_ehc_preferred_usage	$\{0, 1\}$	0	$-$	$-$	$-$
search_1_search_boost	$\{0, 100, 200, 500, 1\,000, 2\,000, 5\,000\}$	0	200	$-$	5000
search_1_search_open_list_tb	$\{true, false\}$	$false$	$false$	$-$	$true$
search_1_search_reopen	$\{true, false\}$	$false$	$false$	$-$	$false$
search_1_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, -1\}$	-1	1.5	$-$	3
search_1_type	$\{none, ehc, eager, lazy\}$	$lazy$	$lazy$	$-$	$lazy$
search_2_cost_type	$\{0, 1\}$	0	0	$-$	0
search_2_eager_pathmax	$\{true, false\}$	$false$	$-$	$-$	$true$
search_2_ehc_preferred_usage	$\{0, 1\}$	0	$-$	$-$	$-$
search_2_search_boost	$\{0, 100, 200, 500, 1\,000, 2\,000, 5\,000\}$	0	5000	$-$	500
search_2_search_open_list_tb	$\{true, false\}$	$false$	$false$	$-$	$true$
search_2_search_reopen	$\{true, false\}$	$false$	$true$	$-$	$true$
search_2_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, -1\}$	-1	5	$-$	10
search_2_type	$\{none, ehc, eager, lazy\}$	$lazy$	$lazy$	$-$	$eager$
search_3_cost_type	$\{0, 1\}$	0	$-$	$-$	$-$
search_3_eager_pathmax	$\{true, false\}$	$false$	$-$	$-$	$-$
search_3_ehc_preferred_usage	$\{0, 1\}$	0	$-$	$-$	$-$
search_3_search_boost	$\{0, 100, 200, 500, 1\,000, 2\,000, 5\,000\}$	0	$-$	$-$	$-$
search_3_search_open_list_tb	$\{true, false\}$	$false$	$-$	$-$	$-$
search_3_search_reopen	$\{true, false\}$	$false$	$-$	$-$	$-$
search_3_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, -1\}$	-1	$-$	$-$	$-$
search_3_type	$\{none, ehc, eager, lazy\}$	$lazy$	$none$	$-$	$none$
search_4_cost_type	$\{0, 1\}$	0	1	$-$	$-$
search_4_eager_pathmax	$\{true, false\}$	$false$	$-$	$-$	$-$
search_4_ehc_preferred_usage	$\{0, 1\}$	0	$-$	$-$	$-$
search_4_search_boost	$\{0, 100, 200, 500, 1\,000, 2\,000, 5\,000\}$	0	1000	$-$	$-$
search_4_search_open_list_tb	$\{true, false\}$	$false$	$false$	$-$	$-$
search_4_search_reopen	$\{true, false\}$	$false$	$true$	$-$	$-$
search_4_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, -1\}$	-1	2	$-$	$-$
search_4_type	$\{none, ehc, eager, lazy\}$	$lazy$	$lazy$	$-$	$none$

Table 4: Parameters controlling search in the configuration space for the satisficing planner, comprising 40 parameters. If $search\_i\_type$ is $none$ for some i, then that entry is left out of the iterated search in Fast Downward. "$-$" indicates that the given parameter is not active.

Fast Downward Stone Soup

Malte Helmert and **Gabriele Röger** and **Jendrik Seipp**
University of Freiburg, Germany
{helmert,roeger,seipp}@informatik.uni-freiburg.de
core team members

Erez Karpas
Technion, Israel
karpase@technion.ac.il
core team member

Jörg Hoffmann and **Emil Keyder**
INRIA, France
contributors

Raz Nissim
Ben-Gurion University, Israel
contributor

Silvia Richter
NICTA, Australia
contributor

Matthias Westphal
University of Freiburg, Germany
contributor

Abstract

Fast Downward Stone Soup is a sequential portfolio planner that uses various heuristics and search algorithms that have been implemented in the Fast Downward planning system.

We present a simple general method for concocting "planner soups", sequential portfolios of planning algorithms, and describe the actual recipes used for Fast Downward Stone Soup in the sequential optimization and sequential satisficing tracks of IPC 2011.

Before We Can Eat

Since the original implementation of the Fast Downward planner (Helmert 2006; 2009) for the 4th International Planning Competition (IPC 2004), various researchers have used it as a starting point and testbed for a large number of additional search algorithms, heuristics, and other capabilities (e. g., Helmert, Haslum, and Hoffmann 2007; Helmert and Geffner 2008; Richter, Helmert, and Westphal 2008; Helmert and Domshlak 2009; Richter and Helmert 2009; Röger and Helmert 2010; Keyder, Richter, and Helmert 2010).

Experiments with these different planning techniques have convinced us of two facts:

1. There is no single common search algorithm and heuristic that dominates all others for classical planning.
2. The coverage of a planning algorithm is often not diminished significantly when giving it less runtime, or put differently: if a planner does not solve a planning task quickly, it is likely not to solve it at all.

Fast Downward Stone Soup is a planning system that builds on these two observations by combining several components of Fast Downward into a *sequential portfolio*. In a sequential portfolio, several algorithms are run in sequence with short (compared to the 30 minutes allowed at the IPC) timeouts, in the hope that at least one of the component algorithms will find a solution in the time allotted to it.

There are two main versions of Fast Downward Stone Soup entered into the IPC: one for optimal planning, and one for satisficing planning. (Each version in turn has two variants, which differ from each other in smaller ways than the optimal planner differs from the satisficing one.)

The optimal portfolio planner exchanges no information at all between the component solvers that are run in sequence. The overall search ends as soon as one of the solvers finds a solution, since there would be no point in continuing after this.

The satisficing portfolio planner is an anytime system that can improve the quality of its generated solution over time. Here, the only information communicated between the component solvers is the quality of the best solution found so far, so that later solvers in the sequence can prune states whose "cost so far" (*g value*) is already as large as or larger than the cost of the best solution that was previously generated.

What is Stone Soup?

The name "Fast Downward Stone Soup" draws from a folk tale (for example told in Hunt and Thomas 2000, p. 7), in which hungry soldiers who are left without food take camp near a small village. They boil a pot of water over their campfire, and into the water they put three stones. This strange behaviour incites the curiosity of the villagers, to whom the soldiers explain that their "stone soup" is known as a true delicacy in the land where they come from, and that it would taste even better after adding some carrots. If the villagers could provide some carrots, they might participate in the feast. Hearing this, one of the villagers fetches the required ingredient, after which the soldiers explain that the recipe could be improved even further by adding potatoes, which another villager readily provides. Ingredient after ingredient is added in this fashion, until the soldiers are happy with the soup and finish its preparation with the final step in the recipe: removing the stones.

The stone soup tale is a story of collaboration. The final result, which benefits from the ingredients provided by a large number of villagers as well as the initiative of the soldiers, is more tasty and more satisfying than what any of the involved parties could have produced by themselves.

We consider the story a nice metaphor for the bits-and-pieces additions by many different parties that Fast Downward has seen in the last four or so years, which is part of the reason for calling the planner "Fast Downward Stone Soup". The second reason is that sequential portfolio algorithms in general can be seen as a "soup" of different algorithms that are stirred together to achieve a taste that hopefully exceeds that of the individual ingredients.

The idea to name a piece of software after the stone soup story is inspired by a similar case, the open-source computer game "Dungeon Crawl Stone Soup[1]", which incidentally would make for an excellent challenge of AI planning technology, similar to but much more complex than the venerable Rog-O-Matic (Mauldin et al. 1984).

Culinary Basics

Fast Downward Stone Soup is not a very sophisticated portfolio planner. Due to deadline pressures, our portfolio was chosen by a very simple selection algorithm, which had to be devised and implemented within a matter of a few hours, without any experimental evaluation, and based on limited and noisy training data. The algorithm does not aim to minimize the training data needed, does not use a separate training and validation set, and completely ignores the intricate time/cost trade-off in satisficing planning. Therefore, we do not recommend our approach as state of the art or even particularly good; rather, we describe it here to document what we did, and as a baseline for future, more sophisticated portfolio approaches.

In order to build a portfolio, we assume that the following information is available:

- A set of *planning algorithms* $\mathcal{A}$ to serve as component algorithms ("ingredients") of the portfolio. Our implementation assumes that this set is not too large; we used 11 ingredients for optimal planning and 38 ingredients for satisficing planning.
- A set of *training instances* $\mathcal{I}$, for which portfolio performance is optimized. We used the subset of IPC 1998–2008 instances that were supported by all planning algorithms we used as ingredients, a total of 1116 instances.[2]
- Complete *evaluation results* that include, for each algorithm $A \in \mathcal{A}$ and training instance $I \in \mathcal{I}$,
  - the *runtime* $t(A, I)$ of the given algorithm on the given training instance on our evaluation machines, in seconds, and
  - the *plan cost* $c(A, I)$ of the plan that was found. (For training instances from IPC 1998–2006, this is simply the plan length.)

  We used a timeout of 30 minutes and memory limit of 2 GB to generate this data. In cases where an instance could not be solved within these bounds, we set $t(A, I) = c(A, I) = \infty$.

  The plan cost is of course only relevant for the satisficing track, since in the optimization track, all component algorithms produce optimal plans. We did not consider anytime planners as possible ingredients. If we had, a single runtime value and plan cost value would of course not have been sufficient to describe algorithm performance on a given instance.

In the following, we represent a (sequential) *portfolio* as a mapping $P : \mathcal{A} \to \mathbb{R}_0^+$ which assigns a time limit to each component algorithm. Time limits can be 0, indicating that a given algorithm is not used in the portfolio. The *total time limit* of portfolio P is the sum of all component time limits, $\sum_{A \in \mathcal{A}} P(A)$.

Judging the Taste of a Soup

We say that portfolio P *solves* a given instance I if any of the component algorithms solves it within its assigned runtime, i. e., if there exists an algorithm A such that $t(A, I) \leq P(A)$. The *solution cost* achieved by portfolio P on instance I is the minimal cost over all component algorithms that solve the task in their allotted time, $c(P, I) := \min\{\, c(A, I) \mid A \in \mathcal{A}, t(A, I) \leq P(A) \,\}$. (If the portfolio does not solve I, we define the achieved solution cost as infinite.)

To evaluate the quality of a portfolio, we compute an instance score in the range 0–1 for each training instance and sum this quantity over all training instances to form a portfolio score. Higher scores correspond to better portfolios for the given benchmark set, either because they solve more instances, or because they find better plans.

In detail, training instances not solved by the portfolio are assigned a score of 0. The score of a solved instance I is computed as the lowest solution cost of any algorithm in algorithm set $\mathcal{A}$ on I, $\min_{A \in \mathcal{A}} c(A, I)$, divided by the cost achieved by the portfolio, $c(P, I)$. Note that this ratio always falls into the range 0–1 since the cost achieved by the portfolio cannot be lower than the cost achieved by the best component algorithm. (We assume that optimal costs are never 0, so that division by 0 is avoided.)

This scoring function is almost identical to the one used for IPC 2008 and IPC 2011 except that we use the best solution quality *among our algorithms* as the reference quality, rather than an objective "best known" solution as mandated by the actual IPC scoring functions. This difference is simply due to lack of time in preparing the portfolios; we did not have a set of readily usable reference results.

In the case of optimal planning, only optimal planning algorithms can be used as ingredients. In this case, the scoring function simplifies to 0 for unsolved and 1 for solved tasks, since all solutions for a given instance have the same cost.

Preparing a Planner Soup

We now describe the generic algorithm for building a planner portfolio, and then detail the specific ingredients used for IPC 2011.

[1] `http://crawl.develz.org`

[2] Fine print: we included IPC 2008 instances which require action cost support, even though three of our ingredients for optimal planning did not support costs. These planners automatically failed on all IPC 2008 instances. IPC 2008 used different instance sets for satisficing and optimal planning, and we followed this separation in our training. For technical reasons to do with hard disk space usage on our experimentation platform, we omitted the cyber security domain from IPC 2008 from the satisficing benchmark suite.

build-portfolio(*algorithms*, *results*, *granularity*, *timeout*):
  portfolio := $\{ A \mapsto 0 \mid A \in \textit{algorithms} \}$
  repeat $\lfloor \textit{timeout}/\textit{granularity} \rfloor$ **times**:
    candidates := successors(*portfolio*, *granularity*)
    portfolio := $\arg\max_{C \in \textit{candidates}}$ score(C, *results*)
  portfolio := reduce(*portfolio*, *results*)
  return *portfolio*

Figure 1: Algorithm for building a portfolio.

We use a simple hill-climbing search in the space of portfolios, shown in Figure 1. In addition to the set of ingredients (*algorithms*) and evaluation results (*results*) as described above, it takes two further arguments: the step size with which we add time slices to the current portfolio (*granularity*) and an upper bound on the total time limit for the portfolio to be generated (*timeout*). Both parameters are measured in seconds. In all cases, we set the total time limit to 1800, the time limit of the IPC.

Portfolio generation starts from an initial portfolio which assigns a runtime of 0 to each ingredient (i. e., does nothing and solves nothing). We then perform hill-climbing: in each step, we generate a set of possible *successors* to the current portfolio, which are like the current portfolio except that each successor increases the time limit of one particular algorithm by *granularity*. (Hence, the number of successors equals the number of algorithms.) We then commit to the best successor among these candidates and continue, for a total of $\lfloor \textit{timeout}/\textit{granularity} \rfloor$ iterations. (If we continued further after this point, the total time limit of the generated portfolio would exceed the given timeout.)

Of course there may be ties in determining the best successor, for example if none of the successors improves the current portfolio. Such ties are broken in favour of successors that increase the timeout of the component algorithm that occurs earliest in some arbitrary total order that we fix initially. We did not experiment with more sophisticated tie-breaking strategies or other search neighbourhoods.

After hill-climbing, a post-processing step reduces the time limit applied to each ingredient by considering the different ingredients in order (the same arbitrary order used for breaking ties between successors in the hill-climbing phase) and setting the time limit of each ingredient to the lowest (whole) number that would still lead to the same portfolio score. For example, if algorithm A is assigned a time limit of 720 seconds after hill-climbing but reducing this time limit to 681 seconds would not affect the portfolio score, its time limit is reduced to 681 (or less, if that still does not affect the score).

Optimizing IPC 2011 Soups

For the sequential optimization track of IPC 2011, we used the following ingredients in the portfolio building algorithm:

- *blind:* A* with a "blind" heuristic that assigns 0 to goal states and the lowest action cost among all actions of the given instance to all non-goal states. Apart from bug fixes and other minor changes, this is the baseline planner used in the sequential optimization track of IPC 2008. This algorithm was contributed by Silvia Richter.
- $h^{\max}$: A* with the $h^{\max}$ heuristic introduced by Bonet and Geffner (2001). This was implemented by Malte Helmert with contributions by Silvia Richter.
- *LM-cut:* A* with the landmark-cut heuristic (Helmert and Domshlak 2009). This was implemented by Malte Helmert. The LM-cut planner was also entered into IPC 2011 as a separate competitor.
- *RHW landmarks*, h^1 *landmarks* and *BJOLP:* LM-A* with the admissible landmark heuristic (Karpas and Domshlak 2009) using "RHW landmarks" (Richter, Helmert, and Westphal 2008), h^1-based landmarks (Keyder, Richter, and Helmert 2010) and, in the case of the "big joint optimal landmarks planner (BJOLP)", the combination of both, respectively.

  The landmark synthesis algorithms were implemented by Silvia Richter and Matthias Westphal (RHW landmarks) and Emil Keyder (h^1-based landmarks), the admissible landmark heuristic by Erez Karpas with some improvements by Malte Helmert based on earlier code by Silvia Richter and Matthias Westphal, and the LM-A* algorithm by Erez Karpas.

  BJOLP was also entered into IPC 2011 as a separate competitor.
- *M&S-LFPA*: A* with a merge-and-shrink heuristic (Helmert, Haslum, and Hoffmann 2007), using the original abstraction strategies suggested by Helmert et al. ("linear f-preserving abstractions"). We use three different abstraction size limits: 10000, 50000, and 100000. This was implemented by Malte Helmert.
- *M&S-bisim 1* and *M&S-bisim 2*: A* with two different merge-and-shrink heuristics, using the original merging strategies of Helmert et al. and two novel shrinking strategies based on the notion of bisimulation. The new shrinking strategies were implemented by Raz Nissim.

  A sequential portfolio of these two planners was entered into IPC 2011 as a separate competitor called "Merge-and-Shrink".

After some unprincipled initial experimentation, we set the granularity parameter for the portfolio building algorithm to 120 seconds. The resulting portfolio is shown in Table 1, which also shows the score (number of solved tasks) of the portfolio and of its ingredients on the training set.[3]

We see that the portfolio makes use of four of the eleven possible ingredients: LM-cut, BJOLP, and the two new merge-and-shrink variants.

With 654 solved instances, the portfolio significantly outperforms BJOLP, the best individual configuration, which solves 605 instances. Moreover, the portfolio does not fall far short of the holy grail of portfolio algorithms (sequential

[3]The performance of the M&S-LFPA algorithms appears to be very bad because we did not manage to implement action-cost support for these algorithms in time, so that they failed on all IPC 2008 tasks. Hence, the numbers reported are not indicative of the true potential of these heuristics.

Algorithm	Score	Time	Marginal
BJOLP	605	455	46
RHW landmarks	597	0	—
LM-cut	593	569	26
h^1 landmarks	588	0	—
M&S-bisim 1	447	175	8
$h^{\max}$	427	0	—
M&S-bisim 2	426	432	20
blind	393	0	—
M&S-LFPA 10000	316	0	—
M&S-LFPA 50000	299	0	—
M&S-LFPA 100000	286	0	—
Portfolio	654	1631	
"Holy Grail"	673		

Table 1: Variant 1 of Fast Downward Stone Soup (sequential optimization). For each algorithm A, the table shows the score (number of solved instances) achieved by A on the training set when given the full 1800 seconds, next to the time that A is assigned by the portfolio. The last column shows the marginal contribution of A, i. e., the number of instances that are no longer solved when removing A from the portfolio.

or otherwise), which is to solve the union of all instances solved by any of the possible ingredients. In our training set, there are 673 instances solved by any of the component algorithms, only 19 more than solved by the portfolio.

The portfolio in Table 1 is not globally optimal in the sense that no other fixed sequential portfolio could achieve a higher score. Indeed, after the planner submission deadline, and with substantial manual effort, we managed to find a slightly better portfolio that solves one more training instance while respecting the 1800 second limit. However, while our portfolio is not optimal on this training set, it is certainly close. We conclude that for this data set, a more sophisticated algorithm for searching the space of portfolios would not increase the number of solved instances substantially. However, a more sophisticated algorithm might guard against overfitting, and hence achieve better performance on unseen instances.

We entered the portfolio shown in Figure 1 into the sequential optimization track of IPC 2011 as variant 1 of Fast Downward Stone Soup. To partially guard against the dangers of overfitting to our training set, we also entered a second portfolio as variant 2, which included equal portions of blind search, LM-cut, BJOLP, and the two M&S-bisim variants.

Satisficing IPC 2011 Soups

Computing a good portfolio for satisficing planning is more difficult than in the case of optimal planning for various reasons. One major difficulty in the case of Fast Downward is that there is a vastly larger range of candidate algorithms to consider.

Initial experiments showed that in some cases greedy best-first search was preferable to weighted A*; in other cases the opposite was true, with no weight uniformly better than others. Sometimes, deferred evaluation is the algorithm of choice, sometimes eager evaluation is better (Richter and Helmert 2009). The choice between lazy and eager search is not clear, either (Richter and Helmert 2009). And last not least, combining different heuristics is very often, but far from always, beneficial (Röger and Helmert 2010).

Since generating experimental data on all training instances takes a significant amount of time, we had to limit our set of ingredients to a subset of all promising candidates. Specifically, we only considered planning algorithms with the following ingredients:

- *search algorithm:* Of the various search algorithms implemented in Fast Downward, we only experimented with greedy best-first search and with weighted A* with a weight of 3. (This weight was chosen very arbitrarily with no experimental justification at all.)
- *eager vs. lazy:* We considered both "eager" (textbook) and "lazy" (deferred evaluation) variants of both search algorithms. This is backed by the study of Richter and Helmert (2009), in which these two variants appear to be roughly equally strong, with somewhat different strengths and weaknesses.
- *preferred operators:* We only considered search algorithms that made use of preferred operators. For eager search, we only used the "dual-queue" method of exploiting preferred operators, for lazy search only the "boosted dual-queue" method, using the default (and rather arbitrary) boost value of 1000. These choices are backed by the results of Richter and Helmert (2009).
- *heuristics:* Somewhat arbitrarily, we restricted attention to four heuristics: additive heuristic h^{add} (Bonet and Geffner 2001), FF/additive heuristic h^{FF} (Hoffmann and Nebel 2001; Keyder and Geffner 2008), causal graph heuristic h^{CG} (Helmert 2004), and context-enhanced additive heuristic h^{cea} (Helmert and Geffner 2008).

  These are the four heuristics that in past experiments have produced best performance when used in isolation. We did not include the landmark heuristic used in LAMA (Richter and Westphal 2010), even though it has been shown to produce very good performance when combined with some of the other heuristics (see, e. g., Richter, Helmert, and Westphal 2008).

  Since Fast Downward supports combinations of multiple heuristics and these are very often beneficial to performance (Röger and Helmert 2010), we considered planner configurations for each of the 15 non-empty subsets of the four heuristics. Backed by the results of Röger and Helmert (2010), we only considered the "alternation" method of combining multiple heuristics.
- *action costs:* We only considered configurations of the planner that treat all actions as if they were unit-cost in the computation of heuristic values and (for weighted A*) g values. This was more due to a mistake in setting up the experiments to generate the training data than due to a conscious decision, but as Richter and Westphal (2010) have shown, this is not necessarily a bad way of handling

Search	Evaluation	Heuristics	Performance	Time	Marg. Contribution
Greedy best-first	Eager	h^{FF}	926.13 / 1021	88	1.82 / 0
Weighted A* ($w = 3$)	Lazy	h^{FF}	921.71 / 1023	340	10.02 / 5
Greedy best-first	Eager	h^{FF}, h^{CG}	919.24 / 1023	76	1.15 / 0
Greedy best-first	Eager	h^{add}, h^{FF}, h^{CG}	909.75 / 1021	0	—
Greedy best-first	Eager	h^{FF}, h^{CG}, h^{cea}	907.52 / 1010	73	1.25 / 0
Greedy best-first	Eager	h^{FF}, h^{cea}	906.92 / 1008	0	—
Greedy best-first	Eager	h^{add}, h^{FF}, h^{CG}, h^{cea}	903.57 / 1012	0	—
Greedy best-first	Eager	h^{add}, h^{FF}	900.52 / 1015	90	1.51 / 1
Greedy best-first	Eager	h^{add}, h^{CG}, h^{cea}	892.08 / 1012	0	—
Greedy best-first	Eager	h^{add}, h^{FF}, h^{cea}	890.96 / 1002	0	—
Greedy best-first	Eager	h^{CG}, h^{cea}	889.93 / 1009	0	—
Greedy best-first	Eager	h^{add}, h^{CG}	888.64 / 1014	0	—
Greedy best-first	Lazy	h^{FF}	880.12 / 1042	171	7.24 / 9
Greedy best-first	Eager	h^{cea}	878.58 / 990	84	3.45 / 2
Greedy best-first	Eager	h^{add}, h^{cea}	877.41 / 999	0	—
Greedy best-first	Lazy	h^{FF}, h^{CG}, h^{cea}	874.64 / 1035	0	—
Weighted A* ($w = 3$)	Eager	h^{FF}	874.18 / 920	87	2.75 / 0
Greedy best-first	Eager	h^{add}	872.74 / 1006	0	—
Greedy best-first	Lazy	h^{FF}, h^{cea}	872.48 / 1037	0	—
Greedy best-first	Lazy	h^{FF}, h^{CG}	871.77 / 1045	49	1.93 / 2
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{CG}, h^{cea}	861.06 / 1032	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{cea}	860.64 / 1031	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{CG}	860.04 / 1042	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}	859.72 / 1046	0	—
Weighted A* ($w = 3$)	Lazy	h^{cea}	849.66 / 1001	0	—
Weighted A* ($w = 3$)	Eager	h^{cea}	844.67 / 938	0	—
Greedy best-first	Lazy	h^{CG}, h^{cea}	841.78 / 1026	27	1.25 / 0
Greedy best-first	Lazy	h^{add}, h^{cea}	839.60 / 1020	0	—
Greedy best-first	Lazy	h^{add}, h^{CG}, h^{cea}	835.33 / 1019	0	—
Greedy best-first	Lazy	h^{add}, h^{CG}	831.28 / 1030	0	—
Weighted A* ($w = 3$)	Lazy	h^{add}	830.39 / 1006	50	0.90 / 0
Weighted A* ($w = 3$)	Eager	h^{add}	828.76 / 936	166	3.35 / 3
Greedy best-first	Lazy	h^{cea}	827.57 / 1014	56	2.04 / 2
Weighted A* ($w = 3$)	Eager	h^{CG}	822.46 / 906	89	2.30 / 1
Greedy best-first	Lazy	h^{add}	808.80 / 1019	0	—
Greedy best-first	Eager	h^{CG}	802.47 / 920	0	—
Weighted A* ($w = 3$)	Lazy	h^{CG}	782.14 / 908	73	2.57 / 1
Greedy best-first	Lazy	h^{CG}	755.43 / 924	0	—
Portfolio			1057.57 / 1071	1519	
"Holy Grail"			1078.00 / 1078		

Table 2: Variant 1 of Fast Downward Stone Soup (sequential satisficing). The performance column shows the score/coverage of the configuration over all training instances. The portfolio uses 15 of the 38 possible configurations, running them between 27 and 340 seconds. The last column shows the decrease of score and number of solved instances when removing only this configuration from the portfolio.

Search	Evaluation	Heuristics	Performance	Time	Marg. Contribution
Greedy best-first	Eager	h^{FF}	960.77 / 1021	330	26.12 / 4
Greedy best-first	Lazy	h^{FF}	914.58 / 1042	411	22.32 / 14
Greedy best-first	Eager	h^{cea}	909.07 / 990	213	9.93 / 5
Greedy best-first	Eager	h^{add}	904.49 / 1006	204	4.56 / 3
Greedy best-first	Lazy	h^{cea}	856.91 / 1014	57	6.17 / 4
Greedy best-first	Lazy	h^{add}	840.94 / 1019	63	1.64 / 0
Greedy best-first	Eager	h^{CG}	829.34 / 920	208	3.48 / 0
Greedy best-first	Lazy	h^{CG}	781.27 / 924	109	3.17 / 1
Portfolio			1064.23 / 1069	1595	
"Holy Grail"			1073.00 / 1073		

Table 3: Variant 2 of Fast Downward Stone Soup (sequential satisficing). Columns as in Table 2.

action costs in the IPC 2008 benchmark suite, and all previous IPC benchmarks are unit-cost anyway.

The implementations of these various planner components are due to Malte Helmert (original implementation of lazy greedy best-first search; implementation of all heuristics except FF/additive), Silvia Richter (implementation of all other search algorithms and of FF/additive heuristic), with further contributions by Gabriele Röger (search algorithms, preferred operator handling mechanisms, heuristic combination handling mechanisms) and by Erez Karpas (search algorithms).

We should emphasize that many potentially good search algorithms were not included in our portfolio, such as the combination of FF/additive heuristic and landmark heuristic used by LAMA (Richter and Westphal 2010). Also, the evaluation data we used for our analysis was partially noisy since some runs were performed before and others after major bug fixes, and machines with different hardware configurations were used for different experiments, introducing additional noise. Finally, there is good reason to believe that our simple hill-climbing algorithm for building portfolios is not good enough to find the strongest possible portfolios according to our scoring criterion.

For variant 1 of Fast Downward Stone Soup in the sequential satisficing track, we considered all possible ingredient combinations for greedy best-first search but due to limited time only included results for weighted A* using single-heuristic algorithms.

With all the caveats mentioned above, the portfolio found by the hill-climbing procedure, shown in Table 2, does indeed achieve a substantially better score than any of the ingredient algorithms. (After significant experimentation, we set the granularity parameter of the algorithm to 90 seconds.) The total score for the best ingredient, eager greedy search with the FF/additive heuristic, is 926.13, while the portfolio scores 1057.57, which is a very substantial gap. The difference between the portfolio and the "holy grail" score of 1078 (achieved by a portfolio which runs each candidate algorithm for 1800 seconds, which of course hugely exceeds the IPC time limit) is much smaller, but nevertheless substantial, so we suspect that better sequential portfolios than the one we generated exist.

For variant 2 we used only greedy best-first search with a single heuristic. The hill-climbing procedure (this time using a granularity of 110 seconds) found the portfolio shown in Table 3. Note that the performance scores are not comparable to the ones of variant 1 because they are computed for a different algorithm set $\mathcal{A}$. The best single algorithm is again eager greedy search with the FF/additive heuristic with a score of 960.77. The total score of the portfolio is 1064.23 which likewise is a huge improvement over the best single algorithm. The gap to the "holy grail" score of 1073 is narrower than for variant 1.

Serving the Soup

We have finished our description of how we computed the portfolio that entered the IPC. We now describe how exactly a run of the portfolio planner proceeds. The simplified view of a portfolio run is that the different ingredients are run in turn, each with their specified time limit, on the input planning task. However, there are some subtleties that make the picture more complicated:

- The Fast Downward planner that underlies all our ingredients consists of three components: translation, knowledge compilation, and search (Helmert 2006). The translation and knowledge compilation steps are identical for all ingredients, so we only run them once, rather than once for each ingredient. (To reflect that this computation is common to all algorithms, the training data we use for selecting portfolios is also based on search time only, not total planning time.)

  While translation and knowledge compilation are usually fast, there are cases where they can take substantial amounts of time, which means that by the time the actual portfolio run begins, we are no longer left with the complete 1800 second IPC time limit.
- The overall time budget can also change in unexpected ways during execution of the portfolio when an ingredient finishes prematurely. In addition to planner bugs, there are three reasons why an algorithm might finish before reaching its time limit: running out of memory, terminating cleanly without solving the instance[4], or finding a

[4]Most of our ingredients are complete algorithms which will

plan. In cases where the full allotted time is not used up by a portfolio ingredient, we would like to do something useful with the time that is saved.

- If a solution is found, we need to consider how to proceed. For optimal planning, the only sensible behaviour is of course to stop and return the optimal solution, but for satisficing search it is advisable to use the remaining time to search for cheaper solutions.

The first and second points imply that we need to adapt to changing time limits in some way. The second and third points imply that the *order* in which algorithms are run can be important. For example, we might want to first run algorithms that tend to fail or succeed quickly. For the first optimization portfolio, we addressed this ordering issue by beginning with those algorithms that use up memory especially quickly. For the first satisficing portfolio, we sorted algorithms by decreasing order of coverage, hence beginning with algorithms likely to *succeed* quickly. For the other portfolios, we used more arbitrary orderings.

To address changing time budgets, we treat per-algorithm time limits defined by the portfolio as *relative*, rather than absolute numbers. For example, consider a situation where after translation, knowledge compilation and running some algorithms in the portfolio, there are still 930 seconds of computation time left. Further, assume that the *remaining* algorithms in the portfolio have a total assigned runtime of 900 seconds, of which 300 seconds belong to the *next* algorithm to run. Then we assign 310 seconds, which is $300/900 = 1/3$ of the remaining time, to the next algorithm. Note that this implies that once the last algorithm in the portfolio is reached, it automatically receives all remaining computation time.[5]

The final point we need to discuss is how to take care of the anytime aspect of satisficing planning. We do this in a rather ad-hoc fashion, by modifying the portfolio behaviour after the first solution is found. First of all, the best solution found so far is always used for pruning based on g values: only paths in the state space that are cheaper than the best solution found so far are pursued.[6]

In both satisficing portfolios, all search algorithms initially ignore action costs (as in our training), since this can be expected to lead to the best coverage (Richter and Westphal 2010). However, unless all actions of task to solve are unit-cost, once a solution has been found we re-run the successful ingredient in a way that takes action costs into account, since this can be expected to produce solutions of higher quality (again, see Richter and Westphal 2010). This is done in the same way as in the LAMA planner, by treating all actions of cost c with cost $c + 1$ in the heuristics, to avoid the issues with zero-cost actions noted by Richter and Westphal (2010). All remaining ingredients of the portfolio are modified in the same way for the current portfolio run.

In the second sequential portfolio, for which we specifically limited consideration to greedy best-first search (which tends to have good coverage, but poor solution quality), we make an additional, more drastic modification once a solution has been found. Namely, we *discard* all further ingredients mentioned in the portfolio, based on the intuition that the current ingredient managed to solve the instance and therefore appears to be a good algorithm for the given instance. Hence, we use the remaining time to perform an anytime search based on the same heuristic and search type (lazy vs. greedy) as the successful algorithm, using the RWA* algorithm (Richter, Thayer, and Ruml 2010) with the weight schedule $\langle 5, 3, 2, 1 \rangle$.

Towards Better Recipes

We close our planner description by briefly mentioning a number of shortcomings of the approach we pursued for Fast Downward Stone Soup, as well as some steps towards improvements.

First, we used a very naive local search procedure. The need to tune the granularity parameter in the portfolio building algorithm highlights a significant problem with our local search neighbourhood. With a low granularity, it can easily happen that no single step in the search neighbourhood improves the current portfolio, causing the local search to act blindly. On the other hand, with a high granularity, we must always increase the algorithm time limits by large amounts even though a much smaller increase might be sufficient to achieve the same effect. A more adaptive neighbourhood would be preferable, for example along the lines of greedy algorithms for the knapsack problem that prefer packing items that maximize the value/weight ratio.

Second, our approach needed complete experimental data for each ingredient of the portfolio. This is a huge limitation because it means that we cannot experiment with nearly as many different algorithm variations as we would like to (as hinted in the description of the satisficing case, where we omitted many promising possibilities). A more sophisticated approach that generates additional experimental data (only) when needed and aims at making decisions with limited experimental data, as in the FocusedILS parameter tuning algorithm (Hutter et al. 2009) could mitigate this problem.

Third, we had to choose all possible ingredients for the portfolio a priori. We believe that there is significant potential in growing a portfolio piecemeal, adding one ingredient at a time, and then specifically *searching* for a new ingredient that complements what is already there, similar to the Hydra algorithm that has been very successfully applied to SAT solving (Xu, Hoos, and Leyton-Brown 2010).

Fourth, unlike systems like Hydra or ISAC (Kadioglu et al. 2010) that learn a classifier to determine on-line which algorithm from a given portfolio to apply to a given instance, we only use *sequential* portfolios, i. e., apply each selected ingredient to each input instance when running the portfolio

not terminate without finding a solution on solvable inputs, but a few exceptions exist. Namely, those algorithms that are based on h^{CG} and/or h^{cea} are not complete because these heuristics can assign infinite heuristic estimates to solvable states, hence unsafely pruning the search space.

[5] If the *last* algorithm in the sequence terminates prematurely, we have leftover time with nothing left to do. Our portfolio runner contains special-purpose code for this situation. We omit details as this seems to be an uncommon corner case.

[6] We do not prune based on h values since the heuristics we use are not admissible.

planner. We believe that this is actually not such a serious problem in planning due to the "solve quickly or not at all" property of many current planning algorithms. Indeed, it may be prudent not to commit to a single algorithm selected by an imperfect classifier.

Finally, the largest challenge we see is in building a portfolio that addresses the anytime nature of satisficing planning in a principled fashion, ideally exploiting information from previous successful searches to bias the selection of the next algorithm to run in order to find an improved solution. As far as we know, this is a wide open research area, and we believe that it holds many interesting theoretical questions as well as potential for significant practical performance gains.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 140–147. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 161–170. AAAI Press.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hunt, A., and Thomas, D. 2000. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Kadioglu, S.; Malitsky, Y.; Sellmann, M.; and Tierney, K. 2010. ISAC – instance-specific algorithm configuration. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 751–756. IOS Press.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1728–1733.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 588–592.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 335–340. IOS Press.

Mauldin, M. L.; Jacobson, G.; Appel, A.; and Hamey, L. 1984. ROG-O-MATIC: A belligerent expert system. In *Proceedings of the Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 273–280. AAAI Press.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 975–982. AAAI Press.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In Brafman, R.; Geffner, H.; Hoffmann, J.; and Kautz, H., eds., *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 137–144. AAAI Press.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In Brafman, R.; Geffner, H.; Hoffmann, J.; and Kautz, H., eds., *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 246–249. AAAI Press.

Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2010)*, 210–216. AAAI Press.

Planning with Implicit Abstraction Heuristics

Michael Katz and **Carmel Domshlak**
Faculty of Industrial Engineering & Management
Technion, Israel

Abstract

We present four planning systems that are based on so-called Implicit Abstraction Heuristics. *ForkInit* and *IForkInit* planning systems are sequentially cost-optimal planners that invoke A^* heuristic search, *LMFork* planning system is a sequentially cost-optimal planner that invoke *LM-*A^* heuristic search, and *ForkUniform* planning system is a sequentially satisficing planner that iteratively invoke weigthed A^* heuristic search. All these different search procedures employ admissible implicit abstraction heuristics. These heuristics are based on two fragments of tractable cost-optimal planning, binary forks and constant-bounded inverted forks, respectively. The heuristics are implemented on top of the Fast Downward planning framework. We describe here the parameters chosen for the participation in the International Planning Competition.

Introduction

Heuristic search, either through progression in the space of world states or through regression in the space of subgoals, is a common and successful approach to classical planning. It is probably the most popular approach to *cost-optimal planning*, that is, finding a plan with a minimal total cost of its actions. The difference between various heuristic-search algorithms for optimal planning is mainly in the admissible heuristic functions they employ. In state-space search, such a heuristic estimates the cost of achieving the goal from a given state and guarantees not to overestimate that cost.

A useful heuristic function must be accurate as well as efficiently computable. Improving the accuracy of a heuristic function without substantially worsening the time complexity of computing it usually translates into faster search for optimal solutions. During the last decade, numerous computational ideas evolved into new admissible heuristics for classical planning; these include the delete-relaxing max heuristic $h_{\max}$ (Bonet and Geffner 2001), critical path heuristics h^m (Haslum and Geffner 2000), landmark heuristics h^{L}, h^{LA} (Karpas and Domshlak 2009) and $h_{\text{LM-CUT}}$ (Helmert and Domshlak 2009), and abstraction heuristics such as pattern database heuristics (Edelkamp 2001), merge-and-shrink heuristics (Helmert, Haslum, and Hoffmann 2007), and implicit abstraction heuristics (Katz and Domshlak 2010b). Our focus in this work is on the *abstraction heuristics*.

Generally speaking, an abstraction of a planning task is given by a mapping $\alpha : S \to S^\alpha$ from the states of the planning task's transition system to the states of some "abstract transition system" such that, for all states $s, s' \in S$, the cost from $\alpha(s)$ to $\alpha(s')$ is upper-bounded by the cost from s to s'. The abstraction heuristic value $h^\alpha(s)$ is then the cost from $\alpha(s)$ to the closest goal state of the abstract transition system. Perhaps the most well-known abstraction heuristics are pattern database (PDB) heuristics, which are based on projecting the planning task onto a subset of its state variables and then explicitly searching for optimal plans in the abstract space. Over the years, PDB heuristics have been shown to be very effective in several hard search problems, including cost-optimal planning (Culberson and Schaeffer 1998; Edelkamp 2001; Felner, Korf, and Hanan 2004; Haslum et al. 2007). The conceptual limitation of these heuristics, however, is that the size of the abstract space and its dimensionality must be fixed.[1] The more recent merge-and-shrink abstractions generalize PDB heuristics to overcome the latter limitation (Helmert, Haslum, and Hoffmann 2007). Instead of perfectly reflecting just a few state variables, merge-and-shrink abstractions allow for imperfectly reflecting all variables. As demonstrated by the formal and empirical analysis of Helmert, Haslum, and Hoffmann (2007), this flexibility often makes the merge-and-shrink abstractions much more effective than PDBs. However, the merge-and-shrink abstract spaces are still searched explicitly, and thus they still have to be of fixed size. While quality heuristics estimates can still be obtained for many problems, this limitation is a critical obstacle for many others.

In attempt to push the envelope of abstraction heuristics beyond explicit abstractions, Katz and Domshlak (2010b) introduced a principled way to obtain abstraction heuristics that limit neither the dimensionality nor the size of the abstract spaces. The basic idea behind so called *implicit abstractions* is simple and intuitive: instead of relying on abstract problems that are easy to solve because they are small, we can rely on abstract problems belonging to provably tractable fragments of optimal planning. The key point is that, at least theoretically, moving to implicit abstractions

[1] This does not necessarily apply to *symbolic* PDBs which, on some tasks, may exponentially reduce the PDB's representation (Edelkamp 2002).

removes the requirement on the abstractions size to be small. Implicit abstractions are, however, far from being of theoretical interest only. Specifically, Katz and Domshlak introduced a concrete family of such abstractions, called *fork decompositions*, that are based on two novel fragments of tractable cost-optimal planning. Likewise, Katz and Domshlak showed that an equivalent of the PDB and merge-and-shrink's (and very important for empirical speed-up) notion of "database" exists for the fork-decomposition abstractions as well, despite the exponential-size abstract spaces of the latter. These *databased implicit abstractions* are based on a proper partitioning of the heuristic computation into parts that can be shared between search states and parts that must be computed online per state.

Of course, as planning is known to be NP-hard even for conservative planning formalisms (Bylander 1994), no heuristic should be expected to work well in all planning tasks. Moreover, even for a fixed planning task, no tractable heuristic will home in on all the "combinatorics" of the task at hand. The promise, however, is that different heuristics could target different sources of the planning complexity, and composing a set of heuristics to exploit their individual strengths could allow a larger range of planning tasks to be solved as well as each individual task to be solved more efficiently.

One of the well-known and heavily-used properties of admissible heuristics is that taking the maximum of their values maximizes informativeness while preserving admissibility. A more recent, alternative approach to composing a set of admissible heuristics corresponds to carefully separating the information used by the different heuristics in the set so that their values could be summed instead of maximized over. This direction was first exploited in devising domain-specific heuristics, and more recently in works on additive pattern database (PDB) heuristics (Edelkamp 2001; Felner, Korf, and Hanan 2004; Haslum et al. 2007) and constrained PDBs and m-reachability heuristics (Haslum, Bonet, and Geffner 2005).

The basic idea underlying all these *additive heuristic ensembles* is elegantly simple: for each planning task's action a, if it can possibly be counted by more than one heuristic in the ensemble, then one should ensure that the cumulative counting of the cost of a does not exceed its true cost in the original task.

Such *action cost partitioning* was originally achieved by accounting for the whole cost of each action in computing a single heuristic in the ensemble, while ignoring the cost of that action in computing all the other heuristics in the ensemble (Edelkamp 2001; Felner, Korf, and Hanan 2004; Haslum, Bonet, and Geffner 2005). Recently, this "all-in-one/nothing-in-rest" action-cost partitioning has been generalized to *arbitrary* partitioning of the action cost among the heuristics in the ensemble (Katz and Domshlak 2007; 2008; Yang, Culberson, and Holte 2007; Yang et al. 2008).

The great flexibility of arbitrary cost partitioning, however, is a mixed blessing. The question of how such a cost partitioning can be found, and which cost partitioning is best arise naturally. These questions are answered, at least for abstraction based heuristics, by defining a method of finding an *optimal* cost partitioning, as well as a cheaper ad-hoc, uniform, cost partitioning (Katz and Domshlak 2010a). The problem of finding an optimal cost partitioning is formulated as a linear program (LP). Although solving a linear program is polynomial (in the size of the LP, which is polynomial in the size of the planning task description), in practice this takes a very long time, and generally, is not feasible to apply in every search node. On the other hand, using an ad-hoc uniform cost partitioning scheme seems to lead to much better results with both of these heuristics, due to much lower computation time per state. In order to try and get the best of both worlds, Katz and Domshlak (2010a) proposed using an optimal cost-partitioning for the initial state to evaluate all other states. This approach was later generalized to optimal cost-partitionings of a set of states (Karpas, Katz, and Markovitch 2011).

Details

In this section we describe the different parts of our planning system in detail, the algorithms, data structures, and parameters.

Search Algorithms

As was previously mentioned, our planning systems are based on heuristic search procedures. These procedures are implemented within the Fast Downward planning frameworks as following.

- A^* is implemented by the Eager Best-First Search algorithm – the classic best-first search. The multi-path-dependent *LM-A** (Karpas and Domshlak 2009) implementation is based on the same code.
- Lazy greedy best-first search and lazy weighted A^* are implemented by the Lazy Best-First Search algorithm – best-first search with deferred evaluation (Richter and Helmert 2009). Both these algorithms also allow for the use of preferred operators.

All these algorithms are expanding states in the order of their $f = g + h$ values, breaking ties by the h value,

Forks and Inverted Forks

The heuristic function h is computed as an admissible additive combination of the true goal distances in the abstract tasks (Katz and Domshlak 2010b). Each such task corresponds to a fragment of tractable cost-optimal planning. Two such fragments employed by the implicit abstraction heuristics are

- fork structured planning task with a binary-domain root variable, and
- inverted fork structured planning task with constant bounded root domain.

A construction of abstract tasks that correspond to such fragments is done as follows. First, for each multi-valued variable, a maximal in term of variables (inverted) fork structured subgraph of the task's causal graph is identified. Then, an acyclic causal-graph decomposition is performed for each such subgraph, resulting in a collection of fork and inverted

fork structured fragments. Finally, several domain abstractions are preformed on these fragments, each resulting in either a binary fork or constant bounded inverted fork.

Binary domain abstractions are performed as follows. For each fork structured fragment, one domain abstraction is performed for each domain value of the root variable, separating it from the rest. Such domain abstraction is referred to as *leave-one-out*. The resulting collection of tractable abstract tasks is henceforth referred as *forks-only*.

Constant bounded domain abstractions, used for inverted forks, are made according to the distance of the values from the goal value. The bound that was selected is 3. That is, the first abstraction distinguishes between the goal, all values that are one step from the goal, and all values that are at least two steps from the goal. The second between those that are at most two steps from the goal, exactly three steps, and at least four steps. The abstractions are created as long as they distinguish between different values. The resulting collection of tractable abstract tasks is henceforth referred to as *inverted forks-only*, while the collection of tractable abstract tasks of both types is henceforth referred to as *both forks and inverted forks*.

For a detailed description of the heuristic construction process both for fork and inverted fork based heuristics we refer the reader to Katz and Domshlak (2010b).

Action Cost Partitioning

As was briefly mentioned above, implicit abstraction heuristics are based on additive composition of abstraction based heuristics. The admissibility criterion for such a composition is referred to as *action-cost partitioning*. Such a partitioning can be performed in several ways. We refer here to two such ways, namely *uniform cost partitioning*, where each action representative in an abstraction gets the same portion of the original action cost, and *optimal for a given state cost partitioning*, where the cost portions are divided in a way that maximizes the heuristic value. A procedure that obtains such a cost partitioning was presented by Katz and Domshlak (2010a).

The procedure, that is based on solving large scale LPs, is implemented within Fast Downward Planning Framework, and makes use of the MOSEK LP solver (MOSEK 2009). Although its time and memory complexities are polynomial, in practice it often takes considerable amount of time and memory, or even fails altogether. Thus, our planning systems attempt at obtaining an optimal for initial state cost partition within the time bound of 5 minutes and memory bound slightly less than 6 GB. If succeeded, an optimal for initial state cost partition is used for all evaluated states along the search, otherwise the uniform cost partition is used. Henceforth, we refer to such an action-cost partitioning scheme as *optimal for initial state* scheme.

All our planning systems employ the *databased* version of implicit abstraction heuristics, that allows for faster per-node computation, as long as the same action-cost partitioning scheme is adopted for all states during the search. For details on *databased implicit abstractions* we refer the reader to Katz and Domshlak (2010b).

Planning Systems

In what follows we describe each of the four planning systems participating in the International Planning Competition 2011 (IPC-11), according to the different tracks.

Sequentially Optimal Track

- *ForkInit* is a cost-optimal planner, based on A^* heuristic search with *forks-only* implicit abstraction heuristic. The action-cost partitioning scheme employed is optimal for initial state, as described above.
- *IForkInit* is a cost-optimal planner, based on A^* heuristic search with *inverted forks-only* implicit abstraction heuristic. The action-cost partitioning scheme employed is optimal for initial state, as described above.
- *LMFork* is a cost-optimal planner, based on *LM-*A^* heuristic search with *forks-only* implicit abstraction heuristic evaluation of the landmarks-enriched planning task (Domshlak, Katz, and Lefler 2010). The action-cost partitioning scheme employed is optimal for initial state, as described above.

Sequentially Satisificing Track *ForkUniform* is a sequentially satisficing planner. It is based on an iterative implication of heuristic search procedures, setting a bound on a solution cost for the next iteration from a previously found solution. All iterations except the first one are using *both forks and inverted forks* implicit abstraction heuristic with uniform action-cost partitioning scheme. In addition, helpful actions (preferred operators) from the FF heuristic (Hoffmann and Nebel 2001) are used to boost the search. The first iteration performs lazy greedy best-first search with FF heuristic and helpful actions, run with a time bound of 5 minutes. Then, lazy weighted A^* is run in each next iteration, decreasing the bounds as 100, 10, 5, 3, 2, and 1, and gradually improving the plan quality. In order to overcome the problem of 0-cost actions, for the purpose of heuristic evaluation all action costs are increased by 1.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1-2):165–204.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(4):318–334.

Domshlak, C.; Katz, M.; and Lefler, S. 2010. When abstractions met landmarks. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, 50–56.

Edelkamp, S. 2001. Planning with pattern databases. In *Proceedings of the European Conference on Planning (ECP)*, 13–34.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Proceedings of the International Conference on AI Planning and Scheduling (AIPS)*, 274–293.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (ICAPS)*, 140–149.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI)*, 1007–1012.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI)*, 1163–1168.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 162–169.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, 176–183.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-09)*, 1728–1733.

Karpas, E.; Katz, M.; and Markovitch, S. 2011. When optimal is just not good enough: Fast near-optimal action cost-partitioning. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*.

Katz, M., and Domshlak, C. 2007. Structural patterns heuristics. In *ICAPS-07 Workshop on Heuristics for Domain-independent Planning: Progress, Ideas, Limitations, Challenges*.

Katz, M., and Domshlak, C. 2008. Structural patterns heuristics via fork decomposition. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, 182–189.

Katz, M., and Domshlak, C. 2010a. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence* 174:767–798.

Katz, M., and Domshlak, C. 2010b. Implicit abstraction heuristics. *Journal of Artificial Intelligence Research* 39:51 – 126.

MOSEK. 2009. The MOSEK Optimization Tools Version 6.0 (revision 61). [Online]. `http://www.mosek.com`.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satiscing planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 273–280.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research* 32:631–662.

Yang, F.; Culberson, J.; and Holte, R. 2007. A general additive search abstraction. Technical Report TR07-06, University of Alberta.

LAMA 2008 and 2011

Silvia Richter
NICTA
silvia.richter@nicta.com.au

Matthias Westphal
Albert-Ludwigs-Universität Freiburg
westpham@informatik.uni-freiburg.de

Malte Helmert
Albert-Ludwigs-Universität Freiburg
helmert@informatik.uni-freiburg.de

Abstract

LAMA is a propositional planning system based on heuristic search with landmarks. This paper describes two versions of LAMA that were entered into the 2011 International Planning Competition: the original LAMA as developed for the 2008 competition and a new re-implementation of LAMA that uses the latest version of the Fast Downward Planning Framework.

Landmarks are propositions that must be true in every solution of a planning task. LAMA uses a heuristic derived from landmarks in conjunction with the well-known FF heuristic. LAMA builds on the Fast Downward Planning System using non-binary (but finite domain) state variables and multi-heuristic search. A weighted A* search is used with iteratively decreasing weights, so that the planner continues to search for plans of better quality until the search is terminated. LAMA combines cost-to-goal and distance-to-goal estimates with the aim of finding good solutions using reasonable runtime.

Introduction

LAMA is a planning system based on heuristic state space search, in the spirit of FF (Hoffmann and Nebel 2001) and Fast Downward (Helmert 2006). It won the sequential satisficing track of the International Planning Competition in 2008 and is entered into this year's competition (2011) as a reference point.

A detailed description of LAMA can be found in a recent JAIR article by Richter and Westphal (2010). This paper gives a brief overview of LAMA, focusing mainly on its architectural structure, landmarks, and the differences between the 2008 and 2011 versions. This paper overlaps significantly with the IPC 2008 planner description of LAMA (Richter and Westphal 2008). Readers familiar with that paper may skip straight to Section *LAMA 2008 versus LAMA 2011*, where we describe LAMA 2011.

LAMA builds on the Fast Downward System, inheriting the general structure of Fast Downward, the translation of PDDL tasks with binary state variables to representations with finite-domain variables, and a search architecture that is able to exploit several heuristics simultaneously. One core feature of LAMA is the use of *landmarks* as a heuristic and for generating preferred operators. The landmark heuristic was introduced in a AAAI 2008 article (Richter, Helmert, and Westphal 2008). Other core features of LAMA are an iterated search using restarts (Richter, Thayer, and Ruml 2010), and the combination of cost and distance estimates.

Structure of the Planner

LAMA consists of three separate programs:

1. the *translator* (written in Python),
2. the *knowledge compilation* module (written in C++), and
3. the *search engine* (also written in C++).

To solve a planning task, the three programs are called in sequence; they communicate via text files.

Translator

The purpose of the *translator* is to transform the planner input, specified in the propositional fragment of PDDL (including ADL features and derived predicates, but not the preferences and constraints introduced for IPC-5), into a finite-domain state representation similar to the SAS$^+$ formalism (Bäckström and Nebel 1995).

The main components of the translator are an efficient grounding algorithm for instantiating schematic operators and axioms and an invariant synthesis algorithm for determining groups of mutually exclusive facts. Such fact groups are consequently replaced by a single finite-domain state variable encoding *which* fact (if any) from the group is satisfied in a given world state.

The groups of mutually exclusive facts found during translation serve an important purpose for determining orders between landmarks. For more details on the translator component, see an article by Helmert (2009). We have modified this component only in some minor ways, including the extraction of all mutually exclusive facts (as mentioned above), the handling of action costs for IPC 2008, and some small enhancements.

Knowledge Compilation

Using the finite-domain task representation generated by the translator, the *knowledge compilation* module is responsible for building a number of data structures which play a central role in the subsequent landmark generation and search.

For example, *domain transition graphs* are produced which encode for each state variable the ways in which it

may change its value through operator applications. Furthermore, the knowledge compilation module constructs *successor generators* and *axiom evaluators*, data structures for efficiently determining the set of applicable actions in a given state of the planning task and for evaluating the values of derived state variables. We refer to Helmert (2006) for more detail on the knowledge compilation component.

Search Engine

Using the data structures generated by the knowledge compilation module, the *search engine* attempts to find a plan using heuristic search with some enhancements, such as the use of *preferred operators* (similar to helpful actions in FF) and *deferred heuristic evaluation*, which mitigates the impact of large branching factors in planning tasks with fairly accurate heuristic estimates (Richter and Helmert 2009). Deferred heuristic evaluation means that states are not evaluated upon generation, but upon expansion. States are thus not selected for expansion according to their own heuristic value, but according to that of their parent. If many more states are generated than expanded, this leads to a substantial reduction in the number of heuristic estimates computed, if at a loss of heuristic accuracy.

The rules of the 6th International Planning Competition (IPC 2008), for which LAMA was designed, suggest a type of search that takes plan quality into account. LAMA first runs a greedy best-first search, aimed at finding a solution as quickly as possible. Once a plan is found, it searches for progressively better solutions by running a series of weighted A* searches with decreasing weight. The cost of the best known solution is used for pruning the search, while decreasing the weight makes the search progressively less greedy, trading speed for solution quality (Richter, Thayer, and Ruml 2010).

The search engine is configured to use several heuristic estimators (namely, the FF heuristic and the landmark heuristic) within an approach called *multi-heuristic search* (Helmert 2006; Röger and Helmert 2010). This technique attempts to exploit strengths of the utilised heuristics in different parts of the search space in an orthogonal way. To this end, it uses separate open lists for each of the different heuristics as well as separate open lists for the preferred operators of each heuristic. Newly generated states are evaluated with respect to all heuristics, and added to all open lists (with the value estimate corresponding to the heuristic of that open list). When choosing which state to expand next, the search engine alternates between the different heuristics, and expands states from preferred-operator queues with higher priority than states from other queues.

Landmarks

Landmarks are variable assignments that must occur at some point in every solution plan. They were first introduced by Porteous, Sebastia and Hoffmann (2001) and later studied in more depth by the same authors (Hoffmann, Porteous, and Sebastia 2004). Consider the logistics example task in Fig. 1, where the goal is to transport the packet from location B to location F. In order to achieve the goal, the packet

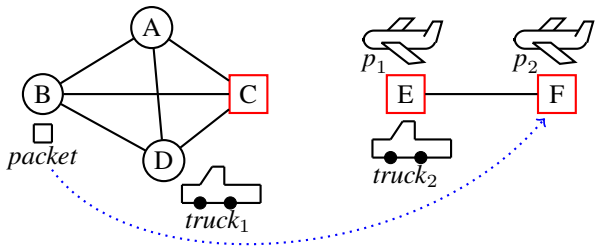

Figure 1: A logistics task: transport packet x from B to F.

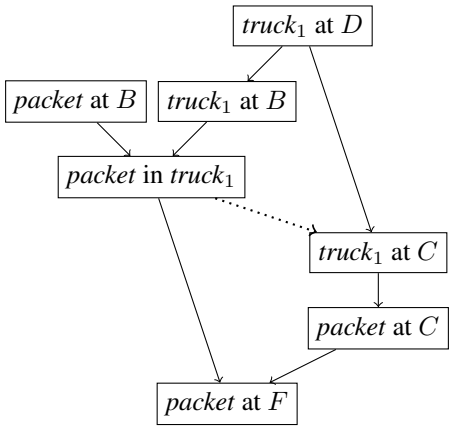

Figure 2: Partial landmark graph for the example task in Fig. 1, showing simple landmarks.

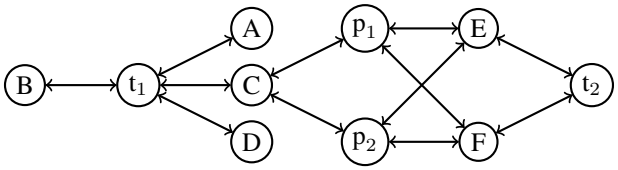

Figure 3: Domain transition graph for the packet from Fig. 1.

must be loaded onto *truck*$_1$ and unloaded at the airport C, in order to then be loaded into one of the planes p_1 or p_2. Hence, the facts "*packet* is on *truck*$_1$", and "*packet* is at C" are landmarks for this task. It is also possible to infer orders between landmarks, e. g. in this case, that "*packet* is on *truck*$_1$" must be true *before* "*packet* is at C". The landmarks and orders can be stored in a directed graph called the landmark graph. For our example, a partial landmark graph is depicted in Fig. 2.

Our algorithm for finding landmarks and orderings between them is similar to the one by Porteous and Cresswell (2002), which is in turn based on the one by Hoffmann et al. Like Porteous and Cresswell we admit *disjunctive* landmarks (sets of propositions of which one needs to be true at some point), but we adapted the algorithm to the SAS$^+$ setting and use domain transition graphs to derive further landmarks.

We find landmarks by backchaining from already known landmarks, starting with the goals (which are landmarks by definition, as they have to be true in every solution plan). For any given landmark L that is not true in the initial state, we consider the *shared preconditions* of its *possible first achievers*. Possible first achievers are those operators that a) have L as an effect, and b) can be possibly applied at the end of a partial plan (starting in the initial state) which has never made L true. Their shared preconditions are those propositions (if any) that are a precondition for each of the operators. Every such shared precondition must be true in order to reach L and is thus a landmark, which can be ordered before L.

Since it is PSPACE-hard to determine the set of *actual* first achievers of a landmark L, we use an over-approximation containing every operators that can *possibly* be a first achiever. By intersecting over the preconditions of more operators we do not loose correctness, though we may of course miss out on some landmarks. The approximation of first achievers of L is done with the help of a relaxed planning graph (RPG) (Hoffmann and Nebel 2001). During construction of the RPG we leave out any operator that would add L. When the relaxed planning graph levels out, its last set of facts is an over-approximation of the set of facts that can be achieved *before* L in the planning task; we denote it by *pb*(L) (for *possibly before*). Any operator that is *applicable* given *pb*(L) and achieves L is a possible first achiever of L.

We also create disjunctive sets of facts from the first achievers' shared preconditions, such that a set contains one precondition fact from each first achiever. Hence, these sets form disjunctive landmarks. All facts in a disjunctive landmark must stem from the same predicate symbol, and we discard any sets of size greater than 4 in order to limit the number of possible sets.

We find further landmarks by exploiting the domain transition graphs (DTGs) generated by the knowledge compilation module. For each variable, a corresponding DTG has a node for each value that can be assigned to the variable, and arcs for possible transitions between them (where a transition can be achieved through operator application). For example, assume that the location of the packet in our example is encoded with a state variable v. The DTG of v is depicted in Fig. 3. From its initial value B, the location of the packet can change to "in *truck*$_1$" (denoted as t_1 for short), from there to any of the locations A, B, C and D and so on.

Given a *simple* (i. e., non-disjunctive) landmark $L = \{v \mapsto l\}$ that is not part of the initial state, we consider the DTG of the landmark's variable v. If there is a node that occurs on *every* path from the *initial state value* $s_0(v)$ of the variable to the *landmark value* l, then that node corresponds to a landmark value l' of the variable: We know that every plan achieving L requires that v take on the value l', hence the fact $L' = \{v \mapsto l'\}$ can be introduced as a new landmark and ordered before L. To find these kinds of landmarks, we iteratively remove one node from the DTG and test with a simple graph algorithm whether $s_0(v)$ and l are still connected – if not, the removed node corresponds to a landmark. Nodes corresponding to assignments of v which are not in *pb*(L) are removed from the DTG prior to this test, as they can only occur *after* B and do not have to be tested. However, we remember these nodes and if such a node is later found to be a landmark (e. g. by the backchaining procedure), we can introduce an ordering between B and the node.

Consider again the landmark graph of our example in Fig. 2. Most of the landmarks and orders in it can be found by the backchaining procedure even when restricting it to simple, i. e., non-disjunctive, landmarks, because the propositions are direct preconditions of their successor nodes in the graph. There are two exceptions: "*packet* in *truck*$_1$" and "*packet* at C". These two landmarks are however found with the DTG method. The DTG in Fig. 3 shows immediately, that the package location must be both t_1 and C on any path from the initial state (where it has value B) to the goal F.

If we introduced another truck in the left city, the fact "*packet* in *truck*$_1$" would no longer be a landmark. However, using disjunctive landmarks we would get a landmark for the packet being inside one of the two trucks.

Inconsistencies found in the translating phase are exploited to determine further orders between the landmarks, using the definition of *reasonable orders* and the conditions proposed by Hoffmann et al. (2004). For example, the order depicted by a dotted line in Fig. 2 is such a reasonable order. For more details on how landmarks and their orders are derived, see the JAIR 2010 article (Richter and Westphal 2010).

The Landmark Heuristic

The LAMA planning system uses landmarks as a pseudo-heuristic. We estimate the goal distance of a state s by the number of landmarks l that still need to be achieved from s onwards. We estimate this number as $\hat{l} := n - m + k$, where n is the total number of landmarks, m is the number of landmarks that are *accepted*, and k is the number of accepted landmarks that are *required again*. A landmark B is accepted in a state s if it is true in that state and all landmarks ordered before B are accepted in the predecessor state from which s was generated. An accepted landmark remains accepted in all successor states. An accepted landmark is *required again* if it is not true in s and it is a direct precondition

of some landmark which is not accepted. Note that $\hat{l}$ is not a proper state heuristic in the usual sense, as its definition depends on the way s was reached during search. Nevertheless, it can be used like a heuristic.

We also generate preferred operators along with the landmark heuristic. An operator is preferred in a state if applying it achieves an *acceptable* landmark in the next step, i. e., a landmark whose predecessors have already been accepted. If no acceptable landmark can be achieved within one step, the preferred operators are those which occur in a relaxed plan to the nearest acceptable landmark.

Action Costs

The landmark heuristic as outlined above estimates the goal distance of states, i. e., the number of operator applications needed to reach the goal state from a given state. Due to the inclusion of action costs in IPC 2008, however, we are interested in generating least-cost plans rather than short plans. Hence, the heuristics used during search should also estimate the *cost* of reaching the goal from a state, not just its goal distance.

The FF heuristic that is also used in our framework can easily be adapted to action costs, as we can use action costs in the underlying *additive heuristic* (Bonet and Geffner 2001). When generating a relaxed plan from the additive heuristic estimates, we simply use the cost rather than the length of that relaxed plan as our estimate for the cost-to-go of a given state. See Keyder and Geffner (2008) for a detailed description of this cost-sensitive version of the FF heuristic which they call FF(h_a).

For the landmark heuristic this method is not directly applicable, since no actual plan is formed by the heuristic. Instead, we *weight* landmarks with an estimate on their minimum cost. Rather than counting the number of landmarks that still need to be achieved from a state, the heuristic value is then the sum of all minimum costs of those landmarks. The cost estimate for each landmark is the minimum cost that is required to make the landmark true for the first time, i. e., the minimum of all action costs of its first achievers.

Zero-cost actions can lead to problems in a standard cost-sensitive search like weighted A*. Since zero-cost actions can always be added to a search path "for free", i. e., without negative side effects, the search may explore very long search paths without getting closer to a goal. In the worst case, this can prevent it from finding a solution within the given time limit. To avoid this problem, LAMA combines cost and distance estimates in a simple fashion, by counting for each action its cost *plus 1 for its distance*. This method has largely overcome the problems of zero-cost actions in our experiments, and offers a simple trade-off between the aim of finding cheap solutions and the need to find solutions within reasonable time. Of course this means that a plan consisting of five originally zero-cost actions is deemed worse by LAMA than a plan consisting of two actions that originally cost one, whereas the opposite is true. A smaller value for the additive constant would lessen the problem, though not solve it completely.

LAMA 2008 versus LAMA 2011

We have entered two versions of LAMA into the 2011 planning competition. LAMA 2008 is largely the same system as the one that won the sequential satisficing track in 2008. However, bug fixes have been incorporated since the 2008 competition, in particular to the translator component, and some improvements to the invariant synthesis in the translator have taken place. The translator component in LAMA 2008 is identical to the translator used in LAMA 2011 and other recent Fast Downward offshoots.

LAMA 2011 is a reimplementation of LAMA in the latest version of the Fast Downward planning framework. In the past two years, the Fast Downward framework has undergone significant changes aimed at back-integrating various offshoots of the original Fast Downward code and making the framework more modular. LAMA 2011 is entered into the competition in particular for comparison with LAMA 2008 and for comparison with other planners based on the latest Fast Downward code.

Compared to the 2008 version, LAMA 2011 uses a more space-efficient way of storing landmark information and a more efficient implementation of the FF heuristic in cases where the values of this heuristic are large. LAMA 2011 furthermore uses a different configuration sequence for its search algorithms. In line with our observations that focus on solution quality may harm coverage (Richter and Westphal 2010), LAMA 2011 runs one iteration of greedy best-first search *ignoring costs* before it starts the sequence of search iterations used by LAMA 2008 (cost-sensitive greedy best-first search followed by weighted A* searches).

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Computational Intelligence* 11(4):625–655.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AIJ* 129(1):5–33.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ* 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *JAIR* 22:215–278.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proc. ECAI 2008*, 588–592.

Porteous, J., and Cresswell, S. 2002. Extending landmarks analysis to reason about resources and repetition. In *Proceedings of the 21st Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG '02)*, 45–54.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *Proc. ECP 2001*, 37–48.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS 2009*, 273–280.

Richter, S., and Westphal, M. 2008. The LAMA planner — Using landmark counting in heuristic search. IPC 2008 short papers, `http://ipc.informatik.uni-freiburg.de/Planners`.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:127–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proc. AAAI 2008*, 975–982.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In *Proc. ICAPS 2010*, 137–144.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proc. ICAPS 2010*, 246–249.

Randward and Lamar: Randomizing the FF Heuristic

Alan Olsen and **Daniel Bryce**
alan@olsen.org, daniel.bryce@usu.edu
Utah State University
Logan, UT

Abstract

We present two planners, Randward and Lamar, which are respectively built upon the Downward and LAMA planners. The technique developed in these planners is to randomize heuristic construction so that heuristic plateaus are less frequent and the bias introduced by poor, arbitrary nondeterministic choices in heuristic construction is removed. Specifically, we randomize the FF heuristic, naming the resulting planner Randward, and with the addition of the landmark count heuristic, name it Lamar. Within the FF heuristic, we randomize planning graph construction, which is akin to a random-walk in the relaxed planning space. From this randomized planning graph, we extract a relaxed plan heuristic (exactly as would FF).

Introduction

Randomization in planning has shown to be effective at overcoming bias in both search (Gerevini, Saetti, and Serina 2003; Nakhost and 0003 2009) and heuristics (Bryce, Kambhampati, and Smith 2008). We present a unique point of randomization in the FF heuristic (Hoffmann 2001), which is then used in two planners, Randward and Lamar. Both planners are built upon the Fast Downward (Helmert 2006) and LAMA (Richter and Westphal 2008) planners, and differ in that Randward does not use the landmark count heuristic, but Lamar does.

The FF heuristic involves constructing a planning graph to solve the relaxed planning problem, and the resulting relaxed plan is used to compute a search heuristic as well as identify preferred operators. The planning graph construction is deterministic in the FF heuristic, and resembles the forward chaining algorithm for Horn clause inference. The algorithm associates with each action a count of the number of unsatisfied preconditions (initially all of its preconditions), and initializes a list of propositions to process. Processing a proposition involves decrementing the number of unsatisfied preconditions for each action using it as a proposition. Once an action's count reaches zero, its effects are added to the proposition list. Processing the propositions as a FIFO queue resembles the traditional planning graph that inserts all actions at the same level, and then moves the next level in a breadth-first manner.

Randomizing the order in which propositions are processed will lead to a random walk in the relaxed planning space. While we do not extract relaxed plans by randomization, we extract a relaxed plan as soon as one exists in the planning graph. Random walks can thus randomize the heuristic and preferred operators. As we have seen in our prior work (Bryce, Kambhampati, and Smith 2008), randomization is useful for overcoming poor decisions (such as proposition order) in a heuristic because, for example, a parent node may have an uncharacteristically low heuristic value, and its child may suffer from the same bias. However by randomizing the heuristic, a child's heuristic is unlikely to suffer from the same bias as the parent, and the probability that an entire path suffers from the same bias is very low.

In the following, we describe the approach taken in Randward and Lamar through an example, discuss preliminary results on the IPC-2008 domains, and outline directions for future work.

Relaxed Planning Graph Expansion Order

Traditionally, relaxed planning graph generation has been explained in a breadth-first fashion, as in (Bryce and Kambhampati 2007). Each proposition in the initial layer is considered towards adding applicable relaxed-actions to the graph at that layer. The add-effects of those actions, as well as all propositions currently true, are added to the next layer. This new set of propositions is used to determine which relaxed-actions are applicable in that layer. This continues until all goal propositions are true, and then a relaxed plan is generated by tracing back through the graph. Figure 1 gives one example of a relaxed search space. Figure 2 shows the breadth-first expansion of propositions in layer P_0 yielding the actions in layer A_0 and ultimately reaching the goal g in layer P_1.

Alternatively, a relaxed planning graph can be generated in a random order. The only difference is which proposition to consider next. As soon as an action is added to the graph, its add-effects immediately become candidates for expansion. When expanding randomly, any random proposition is considered next towards applying actions in its layer. Figure 3 and Figure 4 show two random expansions of the same relaxed search space from the previous scenario. Notice how randomizing removes bias in preferred operators (i.e. which actions are first in the relaxed plan). Also, while the order in which propositions are listed can bias which plans are gen-

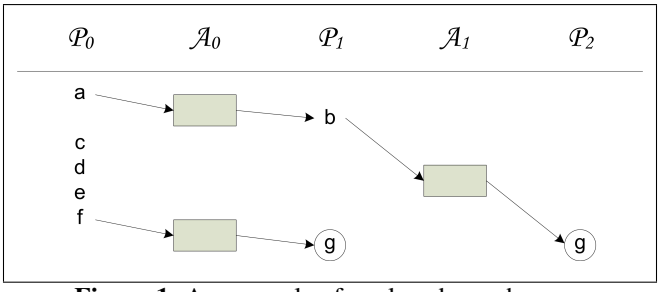

Figure 1: An example of a relaxed search space

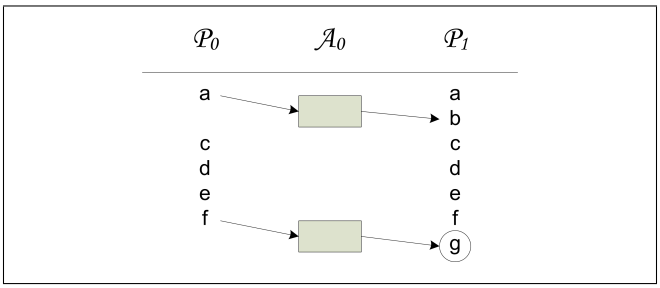

Figure 2: Breadth-first expansion of a relaxed planning graph

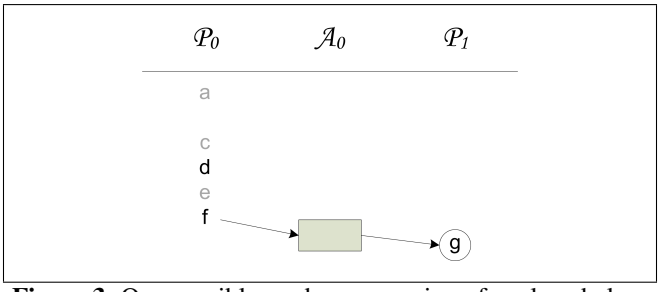

Figure 3: One possible random expansion of a relaxed planning graph (unexpanded propositions in gray)

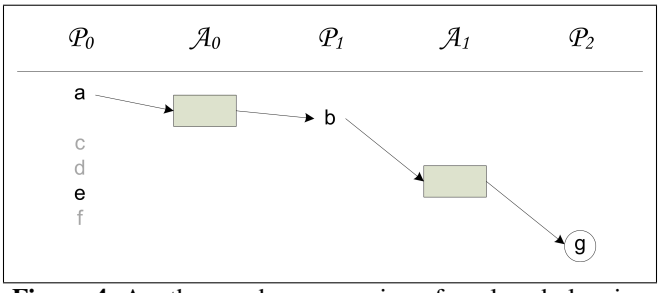

Figure 4: Another random expansion of a relaxed planning graph (unexpanded propositions in gray)

erated by breadth-first expansion, it does not bias those generated through random expansion.

Order of expansion does not effect reachability of goal propositions but it does effect the length of the relaxed plan. Because delete-effects are ignored, actions can only progress the graph towards the goal. Therefore, reaching the goal depends only on applying the right actions, not the order in which they are applied. However, some actions help progress the graph towards the goal sooner than others. Applying those actions first will generate a shorter relaxed plan. Breadth-first is guaranteed to find the step-optimal relaxed plan however random expansion is not guaranteed to. And neither approach is guaranteed to always return the most accurate approximate distance to the goal.

It would be ideal to know which proposition should be considered next to produce the relaxed plan that most accurately approximates the distance to the goal. However, without that knowledge, randomly picking the next proposition avoids the bias inherent in breadth-first expansion.

Results

Each of the IPC-2008 problems were run on a Linux machine with one dedicated 1.95 GHz processor, 2 GB memory, and a time limit of 30 minutes. LAMA and Lamar were also tested without the landmark count heuristic - in which case they are called FF and Randward, respectively. Randomized algorithms were tested on 5 different random seeds.

Table 1 shows that Randward is able to complete more problems than FF, particularly for the elevators and transport domains. This means that simply randomizing the expansion order of the FF heuristic is an improvement.

When landmarks are introduced, however, we see that Lamar completes less problems than LAMA. To expand propositions randomly, LAMA's list of propositions had to be replaced with a priority queue in Lamar. When Lamar's implementation is given breadth-first priorities for each proposition (listed as "LM-BFS" in the table) it performs worse than Lamar. Once again, random expansion is an improvement over breadth-first expansion. Therefore, it is likely that the difference between LAMA and Lamar is implementation related.

While LAMA performs the best, over all, it can still be seen between FF and Randward, and Lamar and LM-BFS, that randomly expanding the relaxed planning graph improves the FF heuristic.

Future Work

Each randomly generated relaxed plan is only one sample from the set of all valid relaxed plans in the relaxed search space and may not be sufficient to get an accurate heuristic. Aggregating over several samples, by taking the average or the minimum, may prove to give a better heuristic value.

Also, if a simple heuristic for relaxed plan expansion exists, then it could be used to perform an A* search over the relaxed planning space. This wouldn't solve the problem of finding the most accurate relaxed plan, however it would be able to return the step-optimal solution quicker than breadth-first.

Domain	FF	Randward	LAMA	Lamar	LM-BFS
cybersec	29	29 (1)	**30**	29.4 (0.5)	**30**
elevators	16	**26.6 (1.3)**	25	26.4 (1.1)	20
openstacks	30	30	30	30	30
openstacks-adl	30	30	30	30	30
parcprinter	12	14.6 (0.9)	**18**	16.2 (0.4)	16
pegsol	**30**	29 (0)	**30**	29.2 (1.1)	28
scanalyzer	28	27.4 (0.5)	**30**	**30**	**30**
sokoban	24	22.8 (1.1)	**25**	22.6 (0.5)	21
transport	21	25.4 (0.5)	**30**	27 (1)	29
woodworking	29	**30**	29	**30**	**30**
Total	249	264.8 (2.3)	**277**	270.8 (2.4)	264

Table 1: Number of IPC-2008 problems solved (30 instances per domain, totaling 300), each given a 30-minute time limit. Randomized algorithms are averaged over 5 seeds with standard deviation in parentheses.

Conclusion

A relaxed planning graph can be expanded in many different ways - two of which have been explained here. Traditionally only a breadth-first approach has been used, which finds the step-optimal relaxed plan but maintains a bias towards certain relaxed plans. A random approach does not assure step-optimality but has shown to produce a better heuristic than breadth-first alone.

Acknowledgments

This work was supported by DARPA contract HR001-07-C-0060.

References

Bryce, D., and Kambhampati, S. 2007. A tutorial on planning graph based reachability heuristics. *AI Magazine* 28(1):47.

Bryce, D.; Kambhampati, S.; and Smith, D. 2008. Sequential monte carlo in probabilistic planning reachability heuristics. *AIJ* 172(6-7):685–715.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs in lpg. *J. Artif. Intell. Res. (JAIR)* 20:239–290.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

Hoffmann, J. 2001. FF: The fast-forward planning system. *AI magazine* 22(3):57.

Nakhost, H., and 0003, M. M. 2009. Monte-carlo exploration for deterministic planning. In Boutilier, C., ed., *IJCAI*, 1766–1771.

Richter, S., and Westphal, M. 2008. The lama planner using landmark counting in heuristic search. *Proceedings of the IPC*.

LPRPG: A Planner for Metric Resources

Amanda Coles, Andrew Coles, Maria Fox and Derek Long
Department of Computer and Information Sciences,
University of Strathclyde, Glasgow, G1 1XH, UK
firstname.lastname@cis.strath.ac.uk

Please direct all citations for this planner to the following paper:
Coles, Fox, Long and Smith *'A Hybrid Relaxed Planning Graph-LP Heuristic for Numeric Planning Domains'*, ICAPS 2008

Abstract

LPRPG is a a planner that is designed to solve problems that use metric resources according to linear constraints. It uses a linear program model of the resources and their use, under the control of actions, to produce tight approximations of the reachable ranges of values for variables during reachability analysis, improving on the approximations produced by Metric-FF (Hoffmann 2003). This improvement has a direct effect on the accuracy of the relaxed plan heuristic and allows LPRPG to solve problems that other metric planners cannot.

In this paper we briefly review the structure of LPRPG.

1 Introduction

Although there have been many recent developments in planning with finite domain state variables, the state of the art for planning with number-valued state variables (both integer and real-valued) has not progressed as fast. It is, of course, the case that planning with metric variables is, in general, undecidable (see, for example, (Helmert 2002)). However, planning with metric resources is a vital capability for handling a vast array of potential application problems. LPRPG (Coles et al. 2008) is a planner that offers one approach to handling certain kinds of metric variable behaviours and can generate solutions to some problems that other planners currently cannot solve.

In this short paper we briefly review approaches to planning with metric resources and the way that LPRPG fits within this landscape.

2 Planning with Metric Variables

One of the most successful strategies in classical planning has proved to be forward state-space search, guided by informed heuristics based on relaxations of the problem. These heuristics typically begin with a reachability analysis for the domain from the current state and this process can be seen as defining unary constraints on the state variables at each time step during throughout the plan. One of the most important contributions to general domain-independent planning with metric variables was made by Hoffmann (2003), who proposed that the domains for the metric variables can be approximated by bounded intervals at each time step. Applicability of actions is tested by confirming that there is *some* assignment of values within their constrained domains at a given time step that makes the precondition true (regardless of whether this assignment is actually achievable for all these variables simultaneously). In fact, this test can be still further relaxed, so that the check confirms that there is some assignment of values that makes each conjunct of the precondition true: the same assignment need not make all conjuncts true. In practice, this second relaxation is not important, since most action preconditions are sufficiently simple that the two relaxations are equivalent.

As actions are judged applicable, their effects are used to update the ranges of the state variables at the next time step. In the case of metric variables, Metric-FF updates the ranges by increasing upper bounds and decreasing lower bounds according to the minimum and maximum effects that the applicable actions can (collectively) achieve on each variable.

This process leads to relatively rapid divergence of the bounds on the reachable range of values for metric variables and these quickly become very coarse approximations of the actual bounds on the reachable range of values. It is perhaps not immediately apparent that this should be so, but consider the actions shown in figure 1. If there is one item available at a location occupied by an empty vehicle, then the bounds on the available and loaded values will start at $[1, 1]$ and $[0, 0]$ respectively, and then change to $[0, 1]$ and $[0, 1]$ after the first step. However, on the following step the application of the effects of both actions (which remain applicable) to the ranges will lead to $[-1, 2]$ for both. The upper and lower bounds will continue to diverge in this way at each subsequent step. We refer to this particular problem as Cyclical Resource Transfer.

The problems this creates are two-fold. Firstly, actions can appear to be applicable much earlier than they will be in reality, leading to distortions in the structures of relaxed plans that undermine their value in heuristic guidance. Secondly, the actions that appear to contribute to solving a problem (the helpful actions) can be entirely flawed because of

```
(:action load
 :parameters (?v - vehicle ?l - location)
 :precondition (>= (available ?l) 1)
 :effect (and (increase (onboard ?v) 1)
              (decrease (available ?l) 1)))

(:action unload
 :parameters (?v - vehicle ?l - location)
 :precondition (>= (onboard ?v) 1)
 :effect (and (increase (available ?l) 1)
              (decrease (onboard ?v) 1)))
```

Figure 1: Simple actions manipulating metric variables.

the ways that they appear to contribute to resource accumulation under the relaxation. We refer to this problem as Helpful Action Distortion.

3 Linear Resource Constraints and LPRPG

In many planning problems, the effects of actions on metric variables are simple increase or decrease effects, using constant increments or decrements. Where all the actions have this form, coupled with preconditions that compare linear combinations of metric variables with constants, the domains are called *linear*. This is because the entire structure of the planning problem, for a finite horizon, can be encoded as a collection of boolean constraints and linear constraints.

LPRPG is designed to work with linear domains. It is structured very similarly to Metric-FF, but it improves the bounds on metric variables generated by Metric-FF by exploiting the fact that the domains are linear. It does this by constructing a linear program (LP) to capture the relationship between actions, their effects and the values of individual metric variables. By solving the LP for a maximum and minimum value of each variable at each time step, it is possible to generate a tighter approximation of the reachable range of values than is achieved by Metric-FF. The reason for this is that the LP encodes the *flow* of metric resources between different metric variables. So, for example, the LP captures the fact that loading or unloading an item using the actions in figure 1 leaves invariant the total number of items, so the bounds on the number of items in the vehicle and at the location remain at $[0, 1]$ throughout the reachability analysis.

Although LPs can be solved very efficiently, in large planning problems a very large number of LPs can be generated. LPRPG is carefully designed to restrict the number of problems that have to be solved and to use the warm-start facility of many LP solvers to minimise the work involved in solving multiple problems that are very similar to one another. The astute reader will have noticed that the effects of actions must be captured as discrete changes in metric variables, which implies that the LP should in fact be an Integer Linear Program (ILP). This is true, but the LP relaxation gives us effective performance in the approximation of the bounds. The LP can also help in identifying appropriate action choices during construction of relaxed plans. In this case, because helpful actions are either to be applied or not, it is difficult to avoid ensuring that some of the action variables should be integral. For this reason, during relaxed plan construction, LPRPG uses an ILP relaxation in which the action variables at the first step (only) are constrained to be integral. Further details can be found in (Coles et al. 2008).

Since the original publication of work on LPRPG (Coles et al. 2008), the planner has been improved or extended in various ways. Haizan (Radzi 2011) has explored an extension to make it more sensitive to plan metrics and to allow it to make different tradeoffs between makespan and the metric value of plans. The competition variant also supports preferences, as described in (Coles and Coles 2011); but as the preferences track was withdrawn, these have not been tested in the competition.

4 The Impact of Tighter Variable Bounds

Although LPRPG can generate tighter bounds on variables and, as we have already observed, this can, in principle, reduce the problems associated with over-approximation of the ranges, the extent to which the improvement affects planner performance depends on the problems. This is because metric variables can be intertwined with finite-domain variables in actions in a wide variety of ways. Metric variables can form a relatively self-contained element of the problem alongside a complex causal structure in the rest of the problem, or they can form a complex and intricate pattern of flows alongside a relatively simple causal problem, or they can interact so that complex causal chains are required to enable particular metric effects and vice versa. LPRGP is most effective when the metric problem is not too tightly intertwined with the manipulation of the finite-domain variables in the problem. This is because the LP encodes the interactions between metric effects of actions, but does not directly reflect the effects of the actions on finite-domain variables. Thus, solutions to the LP ignore the ways in which finite-domain variables might constrain what is actually possible. Thus, a finite-domain variable representing a counted resource would be ignored in the LP, despite it possibly having a fundamental impact on the reachable range of values of metric variables.

We have experimented with ways to encode aspects of the finite-domain variable behaviours in the LP, but it remains an unresolved challenge to find a good balance that achieves efficient performance across a wide range of problems.

5 Conclusion

LPRPG represents an interesting treatment of metric variables in linear domains, offering significant improvements in performance over Metric-FF in many problems. The tradeoff it exploits between the additional effort in finding tighter bounds on the range of reachable values for metric variables and the extra information this generates within the reachability analysis and relaxed plan construction is a difficult one to control and this remains an open challenge in achieving really effective general performance in complex metric and causally-structured domains.

References

Coles, A. J., and Coles, A. I. 2011. LPRPG-P: Relaxed Plan Heuristics for Planning with Preferences. In *Proceedings of the Twenty First International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. A Hybrid Relaxed Planning Graph–LP Heuristic for Numeric Planning Domains. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-08)*.

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *Proc. AI Plan. and Sched. (AIPS)*.

Hoffmann, J. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research* 20:291–341.

Radzi, N. H. M. 2011. *Multi-Objective Planning using Linear Programming*. Ph.D. Dissertation, University of Strathclyde, UK.

Madagascar: Efficient Planning with SAT

Jussi Rintanen
The Australian National University
Canberra, Australia

Abstract

Planning with SAT has long been viewed as a main approach to AI planning, but suffering from a poor scalability to large problems. In this paper we explain how the scalability has been dramatically improved by better implementation technology, and how this, with a shift of understanding about SAT-based planning during the past ten years, enables planners that radically differ from those from the late 1990s. We discuss a SAT-based planning system that implements modern versions of virtually all components of first planners that used SAT.

Introduction

During the last decade, SAT, the prototypical NP-complete problem of testing the satisfiability of the formulas in the classical propositional logic (Cook 1971), has emerged, due to dramatically improved SAT solvers, as a *practical* language for representing hard combinatorial search problems and solving them, in areas as diverse as Model-Checking (Biere et al. 1999), FPGA routing (Wood and Rutenbar 1998), test pattern generation (Larrabee 1992), and diagnosis (Smith et al. 2005). Planning as Satisfiability, which enjoyed a lot of attention in the late 1990s after the works by Kautz and Selman (1996), has remained more in the periphery of planning for the last ten years. This is somewhat surprising, when one considers the great successes of SAT in applications other than planning. In this paper, we show that this situation can be traced back to properties of early SAT-based planning systems, which have made for example heuristic state-space search (Bonet and Geffner 2001) seem a more attractive alternative. We will first give a brief description of the Planning and Satisfiability approach, discuss the issues critical to its time and space complexity in practice, and explain the factors that separate the state-of-the-art now and in the late 1990s. We show that what have been widely assumed to be *inherent* restrictions of the SAT approach to planning, are in fact not. Eliminating the issues leads to planning systems that are competitive with other existing search methods. Specifically, we discuss two critical issues of SAT-based planning: potentially high memory requirements, and the necessity and utility of guaranteeing that plans have the shortest possible horizon length (parallel optimality). We show that the perceived disadvantages of SAT with respect to state-space search have largely disappeared.

The planning system Madagascar (also called Mp or M depending on the heuristics used) implements several of the innovations in planning with SAT, inlcuding compact and efficient encodings (Rintanen, Heljanko, and Niemelä 2006), parallelized/interleaved search strategies (Rintanen 2004; Rintanen, Heljanko, and Niemelä 2006), and SAT heuristics specialized for planning (Rintanen a2010; b2010).

Background

A classical planning problem is defined by a set F of facts (or state variables) the valuations of which correspond to states, one initial state, a set A of actions (that represent the different possibilities of changing the current state), and a goal which expresses the possible goal states in terms of the facts F. A solution to the planning problem is a sequence of actions that transform the initial state step by step to one of the goal states.

The classical planning problem can be translated into a SAT problem of the following form.

$$\Phi_t = I \wedge \mathcal{T}(0,1) \wedge \mathcal{T}(1,2) \wedge \cdots \wedge \mathcal{T}(t-1,t) \wedge G$$

Here I represents the unique initial state, expressed in terms of propositional variables $f@0$ where $f \in F$ is a fact, and G represents the goal states, expressed in terms of propositional variables $f@t, f \in F$. The formulas $\mathcal{T}(i,i+1)$ represent the possibilities of taking actions between time points i and $i+1$. These formulas are expressed in terms of propositional variables $f@i$ and $f@(i+1)$ for $f \in F$ and $a@i$ for actions $a \in A$.

The formula Φ_t is satisfiable if and only if a plan with t time points exists. Planning therefore reduces to performing satisfiability tests for different t. The effectiveness of the planner based on this idea is determined by the following.

1. The form of the formulas $\mathcal{T}(i,i+1)$.
2. The way the values of t are chosen.
3. The way the SAT instances Φ_t are solved.

Below we will discuss the first two issues, and their interplay. The third, SAT solving, can be performed with generic SAT solvers (Moskewicz et al. 2001) or SAT

solvers specialized for planning (Rintanen a2010; b2010; 1998).

Encodings of $\mathcal{T}(i, i+1)$

The encoding of transitions from i to $i+1$ as the formulas $\mathcal{T}(i, i+1)$ determines how effectively the satisfiability tests of the formulae Φ_t can be performed. The leading encodings are the factored encoding of Robinson et al. (2009), and the $\exists$-step encoding of Rintanen et al. (2006). Both of them use the notion of *parallel* plans, which allow several actions at each time point and hence time horizons much shorter than the number of actions in a plan. The encoding by Robinson et al. is often more compact than that by Rintanen et al., but the latter allows more actions in parallel. Both of these encodings are often more than an order of magnitude smaller than earlier encodings such as those of Kautz and Selman (1996; 1999), and also substantially more efficient (Rintanen, Heljanko, and Niemelä 2006; Sideris and Dimopoulos 2010). This is due to the very large quadratic representation of action exclusion in the Graphplan encodings. Rintanen et al. (2006) and Sideris and Dimopoulos (2010) show that eliminating logically redundant mutexes or improving the quadratic representation to linear dramatically reduces the size of the formulas.

It is interesting to now have a new look at the issues faced by the SAT-based planners from late 1990s, more than 12 years later. Kautz and Selman (1999) conclude from a comparison of BlackBox to the MEDIC planner (Ernst, Millstein, and Weld 1997) that "use of an intermediate graph representation appears to improve the quality of automatic SAT encodings of STRIPS problems". However, empirical or other demonstration of the utility of planning graphs as a basis of encodings has not later emerged. The useful information in planning graphs is the "persistent" fact mutexes, equivalent to *invariants* (Rintanen 2008), which can be easily, and more efficiently, computed without constructing the planning graphs. The currently best encodings of planning in SAT encode the planning problem much more compactly than the planning graph encodings allow. The main problem with planning graphs is the quadratic number of action mutexes, which leads to impractically large CNF encodings, without providing performance advantages over other types of encodings (Rintanen, Heljanko, and Niemelä 2006). The performance advantage observed by Kautz and Selman's comparison to MEDIC is most likely have been the more explicit representation of some of the reachability information in planning graphs, which for current generation of SAT solvers (Moskewicz et al. 2001) – radically different from the ones from 1990s – do not make a difference. Kautz and Selman do conclude: *"SAT encodings become problematically large in sequential domains with many operators, although refinements to the encoding scheme can delay the onset of the combinatorial explosion."* In retrospect, the "problematically large" encoding can be seen to have been caused by the quadratic encoding of action mutexes, which has later been reduced to linear (Rintanen, Heljanko, and Niemelä 2006), eliminating the problem. A further disadvantage of planning graphs is that they are incompatible with more efficient forms of parallel plans such as the $\exists$-step plans (Rintanen, Heljanko, and Niemelä 2006).

Scheduling the Solution of the SAT Instances

Kautz and Selman (1996) proposed testing the satisfiability of Φ_t for different values of $t = 0, 1, 2, \ldots$ sequentially, until a satisfiable formula is found. This strategy is asymptotically optimal if the t parameter corresponds to the plan quality measure to be minimized, as it would with sequential plan encodings that allow at most one action at a time. However, BlackBox uses Graphplan-style parallel plans for which the t parameter is meaningless because Graphplan's parallelism notion does not correspond to the actual physical possibility of taking actions is parallel. For STRIPS, Graphplan-style parallelism exactly matches the possibility of totally ordering the actions to a sequential plan (Rintanen, Heljanko, and Niemelä 2006). Hence the parallelism can be viewed as a form of partial order reduction (Godefroid 1991), the purpose of which is to avoid considering all $n!$ different ordering of n independent actions, as a way of reducing the state-space explosion problem. In this context the t parameter often only provides a weak lower bound on the sequential plan length. So if the minimality of t does not have a practical meaning, why minimize it? The proof that t is minimal is the most expensive part of a run of BlackBox and similar planners.

More complex algorithms for scheduling the SAT tests for different t have been proposed and shown both theoretically and in practice to lead to dramatically more efficient planning, often by several orders of magnitude (Rintanen 2004; Streeter and Smith 2007). These algorithms avoid the expensive proofs of minimality of the parallel plan length, and in practice still lead to plans of comparable quality to those with the minimal parallel length. The most effective implementations of these algorithms solve several SAT problems (for different horizon lengths) in parallel. This is not feasible with the 1990s encodings of planning as SAT, as even solving a SAT problem for one horizon length was often impractical due to the sizes of the formulas. Hence the two issues, the size of the encodings and the strategies of considering different horizon lengths, are closely intertwined: without compact encodings and clause representations, efficient plan search, with multiple SAT instance solved simultaneously, is infeasible. The very large size of the early encodings may be the best explanation why the use of the sequential strategy has been popular still quite recently.

Figure 1 depicts the gap between the longest horizon length with a completed unsatisfiability test and the horizon length for the found plan for the Mp planner and for all the instances considered by Rintanen (a2010). The dots concentrate in the area below 50 steps, but outside this area there are typically an area of 30 to 50 horizon lengths for which the SAT test was not completed, in the vast majority of cases because their difficulty well exceeded the capabilities of current SAT solvers. This explains why the use of the parallel strategies which avoid the expensive (but unnecessary) parallel optimality proofs are essential for efficient planning.

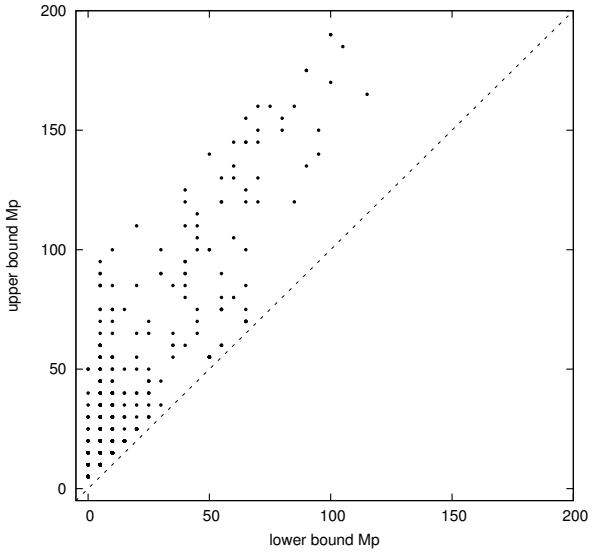

Figure 1: Lower and upper bounds of plan lengths

Progress in Planning as SAT

We have tested a number of planners with 998 STRIPS instances from the planning competitions from 1998 until 2008. Figure 2 shows the performance improvements starting from BlackBox's successor SATPLAN06 with Graphplan-style parallel plans (∀-exists step plans (Rintanen, Heljanko, and Niemelä 2006)), parallel optimality and standard SAT solvers (SATPLAN06), the same with our planner that uses a more compact ∀-step encoding (A-opt), the same with the more efficient ∃-step encoding (E-opt), the same without parallel optimality (E, also known as the planner M (Rintanen a2010)), and finally, the same with a planning-specific heuristic (Rintanen a2010) for SAT solving (Mp). The curves depict the number of instances solved in a given time. To compare these improvements to progress in planning with heuristic state-space search, we have also curves depicting the performance of the HSP (Bonet and Geffner 2001) and LAMA (Richter, Helmert, and Westphal 2008) planners respectively from 1998 and 2008.

One major change in the SAT area has been the dramatic and continuous improvements of SAT algorithms, and it is therefore interesting to make a direct comparison to runtimes given in papers over 10 years ago. The SAT algorithm improvements alone, together with faster CPUs and cheap memory, has lifted SAT based planning to a completely different level. As an illustration of this, consider Kautz and Selman's (1999) Table 2, in which they list the solution times for rocket.$\{a, b\}$ and logistics.$\{a, b, c, d\}$ when finding minimal length Graphplan-style plans. The M planner solves the same problems (including the proof of the parallel optimality) with speed-ups from 393 to 3300 over BlackBox. The same instances without the parallel optimality proofs and by using encodings that allow more parallelism are all solved by Mp in 0.02 seconds or less, with speed-ups of several thousands.

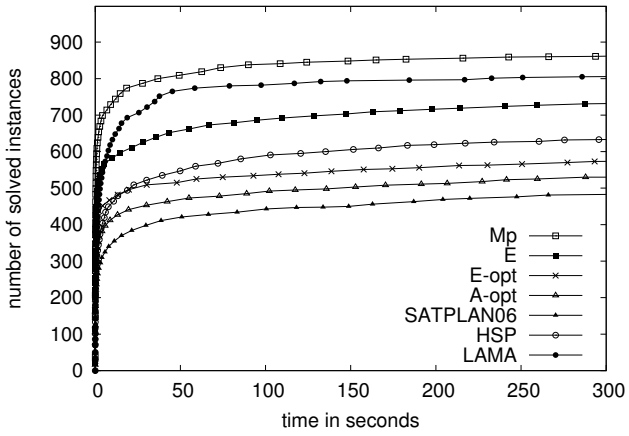

Figure 2: Numbers of instances solved in a given time

	rocket a b		logistics a b c d			
BlackBox	57	60	66	126	180	156
M par. opt.	0.02	0.02	0.02	0.32	0.23	0.16
Mp	0.01	0.01	0.02	0.01	0.01	0.02

Conclusions

We have discussed the main components of a SAT-based planner that avoids the main problems with earlier SAT-based planners, which are excessive memory consumption due to very large quadratic size encodings, top-level strategies that are forced to unnecessarily establish a "parallel optimality" property, and the lack of planning specific heuristics to drive the search.

We argued that the high memory consumption of early SAT-based planners was a main obstacle for the adoption of parallelized planning strategies, which prevent SAT from being competitive with other search methods, such as state-space search. Reduction of the memory consumption by better encodings and implementation technology is essential for achieving truly efficient SAT-based planners.

References

Biere, A.; Cimatti, A.; Clarke, E. M.; and Zhu, Y. 1999. Symbolic model checking without BDDs. In Cleaveland., ed., *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, 193–207. Springer-Verlag.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Cook, S. A. 1971. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 151–158.

Ernst, M.; Millstein, T.; and Weld, D. S. 1997. Automatic SAT-compilation of planning problems. In Pollack, M., ed., *Proceedings of the 15th International Joint Conference on*

Artificial Intelligence, 1169–1176. Morgan Kaufmann Publishers.

Godefroid, P. 1991. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531 in Lecture Notes in Computer Science, 176–185. Springer-Verlag.

Kautz, H., and Selman, B. 1996. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference*, 1194–1201. AAAI Press.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In Dean, T., ed., *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 318–325. Morgan Kaufmann Publishers.

Larrabee, T. 1992. Test pattern generation using Boolean satisfiability. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11(1):4–15.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01)*, 530–535. ACM Press.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, 975–982. AAAI Press.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170(12-13):1031–1080.

Rintanen, J. 1998. A planning algorithm not based on directional search. In Cohn, A. G.; Schubert, L. K.; and Shapiro, S. C., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98)*, 617–624. Morgan Kaufmann Publishers.

Rintanen, J. 2004. Evaluation strategies for planning as satisfiability. In López de Mántaras, R., and Saitta, L., eds., *ECAI 2004. Proceedings of the 16th European Conference on Artificial Intelligence*, 682–687. IOS Press.

Rintanen, J. 2008. Regression for classical and nondeterministic planning. In Ghallab, M.; Spyropoulos, C. D.; and Fakotakis, N., eds., *ECAI 2008. Proceedings of the 18th European Conference on Artificial Intelligence*, 568–571. IOS Press.

Rintanen, J. 2010. Heuristics for planning with SAT. In Cohen, D., ed., *Principles and Practice of Constraint Programming - CP 2010, 16th International Conference, CP 2010, St. Andrews, Scotland, September 2010, Proceedings.*, number 6308 in Lecture Notes in Computer Science, 414–428. Springer-Verlag.

Rintanen, J. 2010. Heuristic planning with SAT: beyond uninformed depth-first search. In Li, J., ed., *AI 2010 : Advances in Artificial Intelligence: 23rd Australasian Joint Conference on Artificial Intelligence, Adelaide, South Australia, December 7-10, 2010, Proceedings*, number 6464 in Lecture Notes in Computer Science, 415–424. Springer-Verlag.

Robinson, N.; Gretton, C.; Pham, D.-N.; and Sattar, A. 2009. SAT-based parallel planning using a split representation of actions. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *ICAPS 2009. Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 281–288. AAAI Press.

Sideris, A., and Dimopoulos, Y. 2010. Constraint propagation in propositional planning. In *ICAPS 2010. Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, 153–160. AAAI Press.

Smith, A.; Veneris, A.; Fahim Ali, M.; and Viglas, A. 2005. Fault diagnosis and logic debugging using Boolean satisfiability. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24(10).

Streeter, M., and Smith, S. F. 2007. Using decision procedures efficiently for optimization. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *ICAPS 2007. Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, 312–319.

Wood, R. G., and Rutenbar, R. A. 1998. FPGA routing and routability estimation via Boolean satisfiability. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6(2).

POPF2: a Forward-Chaining Partial Order Planner

Amanda Coles, Andrew Coles, Maria Fox and Derek Long
Department of Computer and Information Sciences,
University of Strathclyde, Glasgow, G1 1XH, UK
firstname.lastname@cis.strath.ac.uk

Please direct all citations for this planner to one of the following two papers:

(i) To refer to the search approach of POPF2 (or POPF), cite:
Coles, Coles, Fox and Long, *'Forward-Chaining Partial-Order Planning'*, ICAPS 2010

(ii) To refer to the cost-optimisation approach used in POPF2 (or Stochastic-POPF), cite:
Coles, Coles, Clark and Gilmore, *'Cost-Sensitive Concurrent Planning under Duration Uncertainty for Service Level Agreements'*, ICAPS 2011

Abstract

This paper gives an overview of the planner POPF2, as competing in the 2011 International Planning Competition. POPF employs forward-chaining search, expanding a partial-order rather than the conventional total-order: steps added to the plan are ordered after a subset of those in the plan so far, rather than after every step in the plan so far. POPF2 adopts this search approach, extending it in two ways. First, it has a number of changes to allow it to make fewer commitments to ordering constraints, and hence find more makespan-efficient plans. Second, it borrows the cost-optimisation approach of Stochastic-POPF, where modifications to the temporal relaxed planning graph heuristic, and the use of anytime-search, allow it to improve upon the first plan found.

This paper describes the planner POPF2, the latest revision of POPF (Coles *et al.* 2010). The aim of POPF is to preserve the benefits of partial-order plan construction, in terms of producing makespan-efficient, flexible plans; whilst avoiding explicit conflict resolution by always expanding the plan in a forwards direction.

We begin by giving an overview of how POPF works, before describing a number of modifications that further reduce the number of ordering constraints introduced during search in certain situations. We then briefly describe how anytime search is used to find plans of successive quality, through direct application of techniques used in Stochastic-POPF (Coles *et al.* 2011).

1 Background

POPF (Coles *et al.* 2010) is a temporal planner, working under the semantics of PDDL 2.1 (Fox & Long 2003). Each durative action A has:

- duration constraints, placing lower and/or upper bounds on duration of the action.
- instantaneous conditions that must hold either at the start or end of its execution;
- instantaneous propositional and numeric effects that occur at the start and/or end of its execution;
- conditions that must hold continuously `over all` its execution (in the interval between the start and the end);
- numeric effects that occur continuously throughout its execution. In POPF, we insist that such numeric effects are linear, i.e. increase or decrease a numeric variable by a constant amount per unit time.

Under these semantics, each durative action A can be thought of as two instantaneous snap-actions: $A_\vdash$, representing the start of the action, with the associated instantaneous conditions and effects; and $A_\dashv$, similarly representing the end of the action. Applying $A_\vdash$ then begins an interval during which the `over all` conditions of A must be respected, and its continuous numeric effects occur; this interval can then be terminated by applying $A_\dashv$. With this representation, one additional requirement needs to be introduced: for a state to be a goal state, no actions can be executing.

In the general case, states in a temporal planning problem can be characterised by a tuple $\langle F, V, Q, P, C\rangle$, where:

- F is the set of propositions that hold in the state.
- V is the vector of values of the task numeric variables.
- Q is the event queue: is a list of actions whose execution has started but not yet finished. For each $(a, i) \in Q$, a identifies a ground action, and i the step at which it began.
- P is the plan to reach the current state.
- C is a list of temporal constraints over the steps in P.

The temporal constraints in C arise from one of two sources. First, clearly, the duration constraints of actions must be obeyed: the amount of time between $A_\vdash$ and $A_\dashv$ must respect the duration constraint of A. Second, as in non-temporal planning, ordering constraints are needed to ensure the plan is causally sound. In the absence of continuous numeric effects (or instantaneous effects that depend on the duration of actions), these temporal constraints take the form of a Simple Temporal Problem (Dechter, Meiri, & Pearl 1991) (STP), and POPF employs an incremental STP solver (Cesta & Oddi 1996) to check that the temporal constraints at each state are satisfied. In the presence of either or both of these,

then if the continuous numeric change is linear, a Linear Programming (LP) encoding is used, where additional constraints encode the relationship between the times at which actions are scheduled and the values of numeric variables at each point in the plan, with the LP constrained to ensure preconditions are met.

2 POPF Forwards Partial-Order Expansion

The search in POPF is based around the idea of expanding a partial-order plan in a forwards direction. Simply, when applying a snap-action in a given state S, it is ordered only after a subset of the actions in the plan to reach S: those with which there is some sort of causal interaction in terms of facts or numeric variables. This is in contrast to its predecessor, COLIN (Coles *et al.* 2009a), where each new snap-action would be ordered after *all* the actions in the plan to reach S.

To support partial-order expansion, the general case definition of a state given in Section 1 is augmented with additional information, recording which steps in the plan interact with a given fact p or numeric variable v. For full details, including how POPF supports continuous numeric effects, we refer the reader to (Coles *et al.* 2010). In the case where all effects are instantaneous, as is the case in the competition, the state annotations can be summarised as follows:

- $F^+(p)$ ($F^-(p)$) records the step in the plan that most recently had an add effect (resp. delete effect) on the fact p;
- $FP(p)$, a set of pairs, each $\langle i, d\rangle$, records preconditions involving p. For each pair, i is a step index in the plan, and d is 0 or ϵ:
  - If $d = 0$, then p can be deleted in parallel to step i. This arises where step i is the end of an action with an `over all` condition on p: under the PDDL 2.1 semantics, deleting p at this point would not be mutually exclusive with ending the action requiring p.
  - In all other cases (start or end conditions of temporal actions, or the preconditions of non-temporal actions), $d = \epsilon$, and hence p can only be deleted epsilon after step i.
- $V^{\mathit{eff}}(v)$ records the step in the plan that most recently had a numeric effect upon the variable v;
- $VP(v)$ is a set, recording which steps in the plan depend on v. A step i depends on v in one of three cases:
  1. i has a precondition referring to v
  2. i has an effect whose outcome depends on the current value of v (e.g. assigning $w = v$);
  3. i is the start of an action whose duration depends on v.

When an action is applied, as step i of a plan, these annotations are first used to ensure the preconditions of the action are met:

- To satisfy a propositional precondition p, we add the ordering constraint $t(i) - t(F^+(p)) \geq t$. If step i is the start of an action, and p is only needed as an `over all` condition of the action, then $t = 0$. In all other cases, $t = \epsilon$: the effect must complete strictly before step i.
- To satisfy a numeric precondition referring to the variable v, we add the ordering constraint $t(i) - t(V^{eff}(v)) \geq t$ (where, again, the value of t depends on the nature of the precondition).
- If i is the start of an action, with a duration constraint referring to v, then $t(i) - t(V^{eff}(v)) \geq \epsilon$.

Following this, the annotations are used and updated to reflect the effects of the action:

- For a propositional delete effect on p, prior to setting $F^-(p) = i$, we add the ordering constraints:

$$t(i) - t(F^+(p)) \geq \epsilon$$
$$\forall_{\langle j,d\rangle \in FP(p)}\, t(i) - t(j) \geq d$$

- For a propositional add effect on p, we add the ordering constraint $t(i) - t(F^-(p)) \geq \epsilon$, and then set $F^+(p) = i$. (As a special case, if $F^-(p) = i$ then no ordering constraint is added, as the action is adding a fact it has just deleted.)
- For a numeric effect referring to v, $t(i) - t(V^{\mathit{eff}}(v)) \geq \epsilon$;
- For a numeric effect acting on v, prior to setting $V^{\mathit{eff}}(v) = i$ and $VP(v) = \emptyset$, we add the ordering constraints:

$$t(i) - t(V^{\mathit{eff}}(v)) \geq \epsilon$$
$$\forall_{j \in VP(v)}\, t(i) - t(j) \geq \epsilon$$

POPF exploits the approach introduced in (Coles *et al.* 2009b) where if an action A is 'compression-safe', $A_\dashv$ is immediately added to the plan whenever $A_\vdash$ is applied. An action is considered compression safe in this setting if its end preconditions are subsumed by the action's `over all` conditions, and the only end effects are to add propositions (i.e. no numeric effects or delete effects). The state annotations ensure immediately adding $A_\dashv$ does not have a catastrophic effect on makespan: actions will only be ordered after $A_\dashv$ if they require one of its effects, or violate one of the action's `over all` conditions, which are both circumstances under which they would previously have had to follow $A_\dashv$.

3 Introducing Fewer Ordering Constraints

The treatment of numeric variables in the search approach taken in POPF, as described in Section 2, is quite limited. In the interests of generality, effects on numeric variables are totally ordered, and steps requiring a value of v (but not changing its value) are scheduled to occur after all the steps prior to them that modified v, and before all the steps following that modify v. The rationale behind this is that the ordering constraints ensure the value of v is known at every relevant point, and the interleaving of actions with effects and/or preconditions on v cannot be changed in such a way that renders the plan invalid.

To seek to reduce the number of ordering constraints introduced through the use of numeric preconditions and effects, POPF2 performs static analyses on the problem structure to identify patterns of numeric behaviour that can be handled more favourably. The remainder of this section will go through these cases.

3.1 Metric-Tracking Variables with Order-Independent Effects

In various domains, the metric cost function to seek to minimise when planning comprises a number of 'metric-tracking variables'. These are only ever modified by the effects of actions, and the correctness of the plan does not depend on their values: they never appear in preconditions and duration constraints, nor are their values used as a basis for numeric effects. (Indeed, if these variables did not appear in the metric function, they would be irrelevant and could be disregarded entirely.) One example of such a metric-tracking variable can be found in the Time formulation of the ZenoTravel domain from the 2002 International Planning Competition (Long & Fox 2003). Here, the variable `total-fuel-used` is updated by the `fly` and `zoom` actions to record how much fuel has been used so far when constructing the plan. In the problem file set for this domain, the metric is then to minimise a weighted sum of `total-fuel-used` and plan makespan.

If the final value of a metric-tracking variable v does not depend on the order in which the effects upon it are applied, then there is no need to totally order actions affecting v: it suffices that applying all the relevant effects will yield a value of v, which can then be used when determining the quality of the plan found. In other words, the effects on v are 'order independent'. Order-independence can be guaranteed if all effects on v can be written in the form:

$$v\text{+=}c + w_0.v_0 + w_1.v_1 + ... + w_n.v_n$$

...where $c, w_0..w_n \in \Re$, and each $v_i \in [v_0..v_n]$ denotes a state numeric variable. Through the definition of a metric-tracking variable (specifically, that the value of v cannot be used as the basis of a numeric effect), we know that $v \notin [v_0..v_n]$. So long as appropriate ordering constraints are added for each v_i (as discussed in Section 2), each effect on v will then have a known value at the point of being introduced, and the sum of these effect values gives the net effect on v by the plan.

In POPF2, once static analysis has identified a metric-tracking variable v, effects on v will update its value without adding ordering constraints; and when a goal state has been found, the value of v is then available for use in calculating the plan metric. Thus, returning to ZenoTravel, the `total-fuel-used` is increased by the constant-valued effect of each `fly`/`zoom` action, without insisting that these effects are totally ordered.

3.2 As-Needed Ordering after Beneficial Effects

For a given numeric variable v, larger (smaller) values of v may always be preferable, in terms of how the actions in the domain interact with v. Larger values of v are always preferable if:

1. all preconditions (or goals) on v are of the form $v\{\geq, >\}\mathbf{w}.\mathbf{v} + c$, i.e. a larger value of v is more likely to meet the condition;
2. no action has a duration constraint depending on v;
3. the value of v is never used as the basis for a numeric effect.

The first of these is key: for meeting preconditions or goals, a larger v is better. The latter two ensure there are no circumstances in which this might not be the case, and are introduced for simplicity: more sophisticated analyses may be able to relax these, but we leave this to future work. To identify where smaller values of v are preferable, conditions 1 is altered such that all preconditions and goals on v must use a $\leq$ or $<$ operator (rather than $\geq$ or $>$). For the remainder of this subsection, in the interests of clarity, we will discuss only the case where larger is preferable.

If larger values of v are preferable, then we can conclude that increase effects on v are beneficial, and decrease effects on v are not. Returning to the treatment of numeric effects in POPF, the effects on such a variable v are totally ordered, with the most recent effect on v in a given state denoted $V^{\mathit{eff}}(v)$. An action at step i with a precondition on v is then ordered after $V^{\mathit{eff}}(v)$, i.e. after all previous effects on v. In the absence of any assignment effects on v, we can do a little better than this: rather than ordering step i after all previous increase effects on v and all previous decrease effects on v, we order it after all previous decrease effects, and *some* increase effects — enough to satisfy the precondition[1].

In POPF2 the state annotations in POPF are extended to support this. For a variable v where larger values are preferable, the state now also contains $V^{inc}(v)$: a queue of step–effect pairs, each $\langle j, c\rangle$. These correspond to beneficial effects on v: that step j has a (calculated) increase effect v+=c. Preconditions and effects on v interact with $V^{inc}(v)$, and the existing annotations $V^{\mathit{eff}}(v)$ and $VP(v)$, as follows:

- For a new step i with a decrease effect on v , calculated as v-=c , the effect is handled as before: i is ordered after each of $VP(v)$ and $V^{\mathit{eff}}(v)$, $V^{\mathit{eff}}(v) = i$, and the recorded value of v is decreased by c.
- For a new step i with an increase effect on v, calculated as v+=c (based on the values of the variable in the state), a pair $\langle i, c\rangle$ is added to $V^{inc}(v)$, and the recorded value of v is increased by c.
- For a new step i with precondition $v\{\geq, >\}k$, the ordering constraints are determined according to Algorithm 1, ordering the step after all decrease effects, and some of the increase effects (avoiding ordering i after those later in $V^{inc}(v)$) .

For this approach to be reasonable, the effects in $V^{inc}(v)$ must occur in chronological order. Otherwise, woefully inefficient ordering constraints could be introduced, using far later actions to satisfy the precondition than was necessary. As such, we only apply this special-case reasoning to the case where once a given plan step i has been added to the plan, and its minimum timestamp $t(i)$ found, step i will occur at $t(i)$ in all states subsequently reached by extending

[1] It could, in theory, be possible to satisfy a precondition on v by ordering step i before some of the existing decrease effects, but this would contradict the ethos of POPF: the partial-order is only ever expanded in a forwards direction, ordering new steps after existing ones, to avoid the issues of conflict resolution.

Algorithm 1: Ordering after beneficial increase effects on an as-needed basis

Data: a step index i; a numeric precondition $v \geq k$; the recorded value value of v in S, $S[v]$; annotations $V^{\mathit{eff}}(v)$, $VP(v)$ and $V^{inc}(v)$

$C \leftarrow \{t(i) - t(V^{\mathit{eff}}(v)) \geq \epsilon\}$;
$residual \leftarrow S[v]$;
$remaining \leftarrow V^{inc}(v)$;
while $residual \geq k \wedge remaining \neq \emptyset$ **do**
  $\langle j, c\rangle \leftarrow$ the back element of $remaining$;
  $residual \leftarrow residual - c$;
  if $residual \geq k$ **then**
    remove back element of $remaining$;
for *each* $\langle j, c\rangle \in remaining$ **do**
  $C \leftarrow C \cup \{t(i) - t(j) \geq \epsilon\}$;
$VP(v) \leftarrow VP(v) \cup \{i\}$;
return *additional temporal constraints C, and the updated annotations*

this plan. Then, it suffices to order the elements in $V^{inc}(v)$, each $\langle j, c\rangle$, according to $t(j)$, in ascending order from smallest $t(j)$ to largest $t(j)$. The necessary guarantee about $t(i)$ being fixed once the step is added to the plan can only be made if the domain does not require the starts and ends of actions to be coordinated, i.e. if the domain does not contain required concurrency (Cushing *et al.* 2007). For our purposes, we detect that a domain has no required concurrency by observing that all its actions are compression-safe.

4 Special Cases of Compression-Safe Action Detection

POPF inherited the basic notation of compression-safety introduced in (Coles *et al.* 2009b). As noted earlier in this paper, a durative action is 'basically' compression safe if:

- Its end effects only add propositions, i.e. it has no end numeric effects or end propositional delete effects;
- Its `at end` preconditions are a subset of its `over all` conditions, and hence ending the action does not require facts that are not already true through virtue of it executing.

This analysis is somewhat basic: scrutiny of domains will reveal actions that are not considered to be compression-safe according to this definition, but are compression-safe *with respect to the current problem*. Two such cases that we determine analytically in POPF2 are detailed below.

4.1 Compression Safety of Some End Numeric Effects

The intuition behind the general-case definition of compression safety used in POPFis to isolate actions where the end effects are only ever a good idea. As POPF does not support negative preconditions, adding a proposition is only ever beneficial: it does not preclude any actions from taking place. For numeric effects, though, it is not always clear whether a numeric effect is beneficial, so in the general case, an action cannot be compression safe if it has an end numeric effect.

In Section 3.1 we discussed the case where order-independent effects on metric-tracking variables need not be explicitly ordered. We can exploit this to relax the definition of compression safety. Simply, if an action has an end numeric effect v+=$c, c \in \Re$ on a metric-tracking variable upon which all effects are order independent then we can move that effect can be moved to the start of the action. The effect was inevitably going to occur, once the action had started; and the order in which it occurs (with respect to other effects on v) is irrelevant. This is a prime example of where actions can violate the basic definition of compression safety, but are compression safe in the current problem due to the nature of the variables upon which their end numeric effects act.

In Section 3.2, we discussed the case where certain numeric effects can be identified as being beneficial; specifically, those increasing (decreasing) a variable v, where larger (smaller) values of v are definitely preferable. We can also use this here to relax the constraints that determine whether an action is compression safe, by allowing compression-safe actions to have end numeric effects which are definitely beneficial. There is an additional consideration we must make, however: there is a risk that by deeming an action to be compression safe, and hence adding its end to the plan as soon as its start is added, we preclude concurrent activity that was previously possible. For instance, the action A, with start effect v-=c and end effect v+=c, can be applied concurrently alongside itself in a plan ordered:

$$[A_{\vdash}, A_{\vdash}, A_{\dashv}, A_{\dashv}]$$

The total order arises due to each having an effect on v, leading to each updating $V^{\mathit{eff}}(v)$. The action A is typical of the actions involving catalysts in the Pathways domain (Gerevini *et al.* 2009), where the amount of available catalyst is decreased at the start of the action, but then increased at the end. Allowing A to occur in parallel to itself allows the resulting compounds to be obtained sooner, subject to sufficient catalyst being available.

If larger values of v are preferable, A is in theory compression-safe. Exploiting this as in the propositional case, we would then add $A_{\dashv}$ to the plan immediately, as step $i + 1$, whenever $A_{\vdash}$ is added as step i. In POPF, this would result in $V^{\mathit{eff}}(v) = (i + 1)$, forcing the second copy of A to start after $A_{\dashv}$ rather than being able to start after $A_{\vdash}$, as in the total-order fragment above.

To address this potential issue, we only mark actions with end numeric effects as being compression safe if the domain does not contain required concurrency (Cushing *et al.* 2007), and we then exploit this in combination with the 'as-needed' ordering constraint approach described in Section 3.2. Again, for our purposes, we detect that a domain has no required concurrency if all its actions are compression-safe, though there is, ostensibly, a circular argument here: an end numeric effect is compression-safe if all actions are compression-safe. To address this, we first loop over the actions, marking them as compression safe (according to the basic definition of POPF) or hypothetically compression safe (subject to all other actions being compression

safe). Then, if there is one non-compression-safe action, the hypothetically compression-safe actions are marked as being non-compression-safe. Otherwise, they are considered to be compression-safe, as the increase-effect-queue described in Section 3.2 is sufficient to preserve opportunities for concurrency.

4.2 The Over-All, End-Delete-Effect Idiom

In general, in the absence of negative preconditions (as in POPF) delete effects are never beneficial: deleting a fact can only preclude actions from being applied. As such, in the basic definition of compression safety, end delete effects are prohibited. But, consider two durative actions A and B, each with condition `(over all (p))` and effect `(at end (not (p)))`, and an instantaneous action C with precondition `(over all (p))` and effect `(at end (not (p)))`. It is clear that:

- $A_{\dashv}$, $B_{\dashv}$ and C are mutually exclusive (all deleting the fact `p`), so cannot occur at the same time;
- Neither $A_{\dashv}$ nor C can fall within the execution of B, or as it would violate the `over all` condition. (Similarly, neither $B_{\dashv}$ nor C can fall within the execution of A.)

In common between all three of these actions is the notion that `p` is required for some amount of time (instantaneously, in the case of C) but then inevitably destroyed. If this idiom covers all the uses of `p` in the problem, then we can allow end delete effects on `p` to be considered to be compression safe. In effect, deleting `p` can be moved to the start of the actions such as A or B. There is no point maintaining `p` throughout actions such as A, as no action referring to `p` can be applied. Such an action would either:

- follow the pattern of C, immediately deleting `p` and hence violating the active `over all` condition of A;
- follow the pattern of B, thereby leading to a guaranteed future conflict between either the `over all` condition of A and the effect of $B_{\dashv}$, or the `over all` condition of B and the effect of $A_{\dashv}$.

5 Anytime Search

In its original form, POPF terminated after the first plan found. A derivative of POPF, Stochastic-POPF (Coles *et al.* 2011), has recently extended this to both search in domains where action durations are uncertain, but also to seek to minimise plan cost. The techniques of Stochastic-POPFcan be applied in here, in a deterministic setting, unaltered. For full details, we refer the reader to the paper on Stochastic-POPF, but we will sketch the approach here. The search algorithm can be summarised as follows:

- Search begins with an upper-bound on acceptable plan quality of ∞
- Attempts to find a plan using enforced hill-climbing (EHC). If a plan is found, it is stored, and the upper-bound on acceptable plan quality is then set to the quality of this plan;
- Irrespective of whether a plan is found by EHC, then search using WA* (where $W = 5$, $g(n)$ is the plan length to node n and $h(n)$ is its heuristic value). If the variables used to record plan cost are monotonically worsening, then nodes in the search space are discarded if the cost of the plan to reach that state equals or exceeds the acceptable upper-bound on plan quality. This is similar to MIPS-XXL (Edelkamp, Jabbar, & Nazih 2006), where each time a new best solution plan is found, an additional goal is added to ensure the next plan is of better quality; but the planner does not start search from the initial state each time a new best plan is found.

Stochastic-POPF also contains an updated temporal relaxed planning graph (RPG) heuristic, where admissible estimates on the cost of reaching each fact are maintained. The approach taken is based on the costed RPG of Sapa (Do & Kambhampati 2003), extended to handle the case where certain costs are only relevant to achieving facts that only appear as goals. This further supports state pruning: if the cost of reaching the goals from a given state would definitely lead to a plan being found that is worse than the incumbent, the state can be pruned. extended to perform pruning (rather than preferring

Acknowledgements

The authors would like to thank Allan Clark and Stephen Gilmore for motivating the development of Stochastic-POPF, and David Smith for helpful discussions regarding a problem-specific notion of compression-safe actions. This work was supported by SICSA, and EPSRC fellowship EP/H029001/1.

References

Cesta, A., and Oddi, A. 1996. Gaining Efficiency and Flexibility in the Simple Temporal Problem. In *Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME-96)*.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2009a. Temporal planning in domains with linear processes. In *IJCAI '09*.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2009b. Extending the Use of Inference in Temporal Planning as Forwards Search. In *Proc. ICAPS*.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2010. Forward-chaining partial-order planning. In *ICAPS*.

Coles, A. J.; Coles, A. I.; Clark, A.; and Gilmore, S. T. 2011. Cost-sensitive concurrent planning under duration uncertainty for service level agreements. In *ICAPS*.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. 2007. When is temporal planning *really* temporal planning? In *Proc. of Int. Joint Conf. on AI (IJCAI)*, 1852–1859.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49:61–95.

Do, M. B., and Kambhampati, S. 2003. Sapa: Multi-objective Heuristic Metric Temporal Planner. *JAIR* 20:155–194.

Edelkamp, S.; Jabbar, S.; and Nazih, M. 2006. Large-Scale Optimal PDDL3 Planning with MIPS-XXL. In *IPC5 booklet, ICAPS*.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension of PDDL for Expressing Temporal Planning Domains. *JAIR* 20:61–124.

Gerevini, A. E.; Long, D.; Haslum, P.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic Planning in the Fifth International Planning Competition: PDDL3 and Experimental Evaluation of the Planners. *AIJ*.

Long, D., and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *J. of Art. Int. Res.* 20:1–59.

Searching with Probes: The Classical Planner PROBE

Nir Lipovetzky
DTIC Universitat Pompeu Fabra
Barcelona, SPAIN
nir.lipovetzky@upf.edu

Hector Geffner
ICREA & Universitat Pompeu Fabra
Barcelona, SPAIN
hector.geffner@upf.edu

Background

Heuristic search has been the mainstream approach in planning for more than a decade, with planners such as FF, FD, and LAMA being able to solve problems with hundreds of actions and variables in a few seconds (Hoffmann and Nebel 2001; Helmert 2006; Richter and Westphal 2010). The basic idea behind these planners is to search for plans using a search algorithm guided by heuristic estimators derived automatically from the problem (McDermott 1996; Bonet and Geffner 2001). State-of-the-art planners, however, go well beyond this idea, adding a number of techniques that are specific to planning. These techniques, such as helpful actions and landmarks (Hoffmann and Nebel 2001; Hoffmann, Porteous, and Sebastia 2004; Richter, Helmert, and Westphal 2008), are designed to exploit the *propositional structure* of planning problems; a structure that is absent in traditional heuristic search where states and heuristic evaluations are used as *black boxes*. Moreover, new search algorithms have been devised to make use of these techniques. FF, for example, triggers a best-first search when an incomplete but effective greedy search (enforced hill climbing) that uses helpful actions only, fails. In FD and LAMA, the use of helpful or preferred operators is not restricted to the first phase of the search, but to one of the open lists maintained in a multi-queue search algorithm. In both cases, dual search architectures that appeal either to two successive searches or to a single search with multiple open lists, are aimed at solving fast, large problems that are simple, without giving up completeness on problems that are not.

PROBE

The planner PROBE implements a new dual search architecture for planning that is based on the idea of *probes*: single action sequences computed without search from a given state that can quickly go deep into the state space, terminating either in the goal or in failure.

PROBE is a complete, standard greedy best first search (GBFS) STRIPS planner using the standard additive heuristic (Bonet and Geffner 2001), with just *one change*: when a state is selected for expansion, it first launches a *probe* from the state to the goal. If the probe reaches the goal, the problem is solved and the solution is returned. Otherwise, the states expanded by probe are added to the open list, and control returns to the GBFS loop. The crucial and only novel part in the planning algorithm is the definition and computation of the probes

The main contribution in PROBE is the design of these probes. A probe is an action sequence computed greedily from a seed state for achieving a *serialization of the problem subgoals* that is computed dynamically along with the probe. The next subgoal to achieve in a probe is chosen among the first unachieved landmarks that are *consistent*. Roughly, a subgoal that must remain true until another subgoal is achieved, is consistent, if once it is made true, it does not have to be undone in order to make the second subgoal achievable. The action sequence to achieve the next subgoal uses standard heuristics and helpful actions, while maintaining and enforcing the reasons for which the previous actions have been selected in the form of *commitments* akin to causal links.

Probes are described in (Lipovetzky and Geffner 2011). As shown in that work, a single probe from the initial state manages to solve by itself 683 out of 980 problems from previous IPCs, a number that compares well with the 627 problems solved by FF in EHC mode, with similar times and plan lengths. Moreover, when a probe is launched from each expanded state in a standard greedy best first search informed by the additive heuristic, the number of problems solved jumps to 900 (92%), which compares well with 827 problems solved by FF (84%), and the 879 problems solved by LAMA (89%). In this note we focus on the changes made to PROBE to adapt it to the Int. Planning Competition 2011.

Improving the First Solution: Anytime Search

In the IPC, planners are given a time window and are rewarded when they compute good quality solutions. Since the time window is often much larger than the time required to find a solution, the IPC version of PROBE follows LAMA in trying to compute a plan fast, and then using the rest of the time to iteratively improve the best plan found so far. The first part is achieved by using PROBE as described in (Lipovetzky and Geffner 2011), i.e. by performing a Greedy Best First Search with probes. The second part in turn is achieved by iteratively triggering a Weighted A* search

without probes, with a reduced weight W on the heuristic term, and by keeping the cost of the best solution as a bound so that plan prefixes whose cost does not improve the bound are pruned. The heuristic used by the WA* phase in PROBE is given by the size of the 'cost sensitive relaxed plan heuristic', which is given by the size of the relaxed plan as produced by the additive heuristic (Keyder 2010),

Dealing with Non-Uniform Costs

The direct approach of replacing length-based heuristics by cost-based heuristics when plan cost is different than plan length is known to run into a problem: if length estimates are ignored, the coverage over many domains is reduced (Richter, Helmert, and Westphal 2008; Keyder 2010). In order to avoid this problem, PROBE treats costs in two ways: in the first stage, for finding the first solution (GBFS with probes), action costs are ignored (i.e., they are all taken to be 1), while in the second stage (WA*), they are taken into account. We found that some problems could be solved in this manner that could not be solved if the real action costs were used in both stages.

An important issue appears with the presence of zero cost actions that can lead to heuristic plateaus in which the application of such operators does not decrease the cost to the goal. In order to avoid these situations, we added a base cost of 0.01 to all zero cost actions (Keyder 2010).

When Probes Fail

In most classical benchmarks a single probe suffices to find a solution, suggesting that most problems admit good landmark serializations On the other hand, in problems with no perfect serializations, such as Sokoban or the 8-puzzle, too many probes turn out to be needed to find a solution, something which rather than boosting the performance of the GBFS loop, slows it down. In order to avoid triggering probes that are not likely to help, we measure the progress that the probes do in solving the problem. Basically, we assume that a probe is useless if the end state of the probe is no better than the first state in the probe, where the notion of better is given as in FF by the heuristic: the final state of the probe is better than the seed state of the probe if its heuristic value is smaller. When a probe is found to be useless in this sense, i.e. it doesn't improve the value of the seed state, then a parameter R, called the *probe ratio* is increased by 1. The meaning of this parameter is that the planner launches a probe every R expanded nodes. The parameter is initially set to 1, and then when probes fail without doing 'useful work', it is increased, so that probes end up being triggered less and less often, thus reducing their overhead.

Experimental Results

We compare PROBE with FF and LAMA over the domains of the last IPC.[1] PROBE is written in C++ and uses Metric-FF as an *ADL* to *Propositional STRIPS* compiler (Hoffmann 2003). LAMA and PROBE are executed without the plan improvement option, reporting the first plan that they find. All experiments were conducted on a dual-processor Xeon 'Woodcrest' running at 2.33 GHz and 8 GB of RAM. Processes time or memory out after 30 minutes or 2 GB. As the first solution of PROBE ignores costs, all action costs are assumed to be 1 so that plan cost is plan length also for LAMA and FF. Table 1 compares PROBE with FF and LAMA over 270 instances from last IPC. In terms of coverage, PROBE solves 7 more problems than LAMA and 37 more than FF.

Domain	I	FF	LAMA	PROBE
Cyber	30	4	30	30
Elevator	30	30	30	30
Openstacks	30	30	30	30
Parc-Printer	30	30	25	30
Pegsol	30	30	30	30
Scanalyzer	30	30	30	28
Sokoban	30	27	25	24
Transport	30	29	30	30
Woodworking	30	17	28	30
Total	270	227	257	264
Percentage		84%	95%	98%

Table 1: Coverage of PROBE vs. FF and LAMA over instances of the last IPC where *I* is the number of instances per domain

[1] FF is FF2.3, while LAMA is the 2008 IPC version. with a more recently fixed parser.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *JAIR* 22:215–278.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *JAIR* 20:291–341.

Keyder, E. 2010. *New Heuristics For Planning With Actioin Costs*. Ph.D. Dissertation, Universitat Pompeu Fabra.

Lipovetzky, N., and Geffner, H. 2011. Searching for plans with carefully designed probes. In *Proc. ICAPS-11*.

McDermott, D. 1996. A heuristic estimator for means-ends analysis in planning. In *Proc. AIPS-96*.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:122–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proc. AAAI-08*.

The Roamer Planner
Random-Walk Assisted Best-First Search

Qiang Lu
University of Science and Technology of China
Hefei, Anhui, China
rczx@mail.ustc.edu.cn

You Xu, Ruoyun Huang, Yixin Chen
Washington University in St. Louis
St. Louis, MO, USA
{yx2,ruoyun.huang,chen}@cse.wustl.edu

Abstract

Best-first search is one of the most fundamental techniques for planning. A heuristic function is used in best-first search to guide the search. A well-observed phenomenon on best-first search for planning is that for most of the time during search, it explores a large number of states without reducing the heuristic function value. This phenomenon, called "plateau exploration", has been extensively studied for heuristic search algorithms for satisfiability (SAT) and constraint satisfaction problems (CSP).

In planning, plateau exploration consists of most of the search time in state-of-the-art best-first search planners. Therefore, their performance can be improved if we can reduce the plateau exploration time by finding an exit state (a state with better heuristic value than the best one found so far). In this paper, we present a random-walk assisted best-first search algorithm for planning, which invokes a random walk procedure to find exits when the best-first search is stuck on a plateau. The resulting planner, Roamer, building on the LAMA and Fast-Downward planning system, uses a best-first search in first iteration to find a plan and a weighted A* search to iteratively decreasing weights of plans. Roamer is an anytime planner which continues to search for plans with better quality until exhausting the whole state space or being terminated because of time limits.

Introduction

Roamer is a best-first state space search planner based on Fast-Downward and LAMA planners (Helmert 2006; Richter and Westphal 2010). The core feature of Roamer is the use of Monte-Carlo Random Walks assisted heuristic search to escape from plateaus. The Monte-Carlo random exploration method for deterministic planning is introduced in (Nakhost and Msller 2009).

One of the most successful approaches to planning is best-first search. Best-first search typically employs a heuristic function that maps any state to a real number that estimates the distance to goal. The number of states expanded by best-first search depends largely on the quality of the heuristic function. Best-first search with a perfect heuristic function only needs to expand $O(|L|)$ states where L is the solution path from the initial state to a goal state (Russell and Norvig 2003). On the other hand, best-first search for planning with almost perfect heuristic may still explore an exponential number of states before finding a goal (Helmert and Röger 2008). In practice, since the length of the solution path L is much smaller than the number of expanded states, it is easy to see that most of the states explored by a best-first search are not on the solution path.

During the best-first search, for any state s explored, we define the incumbent heuristic value $h^*(s)$ as the smallest heuristic function value of all states explored so far till s. Evidently, h^* decreases monotonically during search and finally reaches 0 when a goal is found.

For a given planning problem, let $\mathcal{S}$ be the set of all generated states, since we have $|\mathcal{S}| \gg h^*(s_0)$, and $h^*(s)$ is a monotonic mapping from $\mathcal{S}$ to $[0, h^*(s_0)]$, we see that for most of the state pairs (s, s') where s' is explored right after s during the search, $h^*(s) = h^*(s')$.

The reasoning above shows that most of the time a best-first search for planning explores states without reducing h^*. This phenomenon is named *plateau exploration* as it involves state exploration without changing h^*. Therefore, to improve the performance of best-first search for planning, it is important to find a way that can reduce plateau exploration.

To address this challenge, in the Roamer planner, we use random walks to assist best-first search for planning to escape from plateau more quickly. Specifically, when the best-first search makes no progress on h^* for an extended period, we use a random walk algorithm to explore the search space and help escape from the plateau.

There are three advantages of using random walks to assist best-first search for planning. First, a random walk has the potential to directly and quickly jump out of a local minima region where it is not likely to find an "exit" state that reduces h^*, whereas a best-first search will have to explore all possible states around the local minima. Second, comparing to best-first search in which heuristic functions are evaluated at each state, the random walk algorithm can skip heuristic evaluations of most of the intermediate states during exploration, making space exploration more efficient. Third, random walks require little memory, and therefore do not add space complexity to the original best-first search.

Plateau Explorations

Plateaus during search have been well studied for both SAT and CSP problems (Hampson and Kibler 1993). In SAT and CSP problems, a plateau is defined as a set of neigh-

boring variable assignments that lead to the same number of unsatisfied constraints or clauses (Frank, Cheeseman, and Stutz 1997; Russell and Norvig 2003). Plateau structures have also been studied in planning under the context of local search. A detailed analysis on why some planning problems are simple and how long the maximum exiting distance is in enforced hill-climb are presented in (Hoffmann 2002). G-value plateau in planning has also been studied in (Benton et al. 2010).

Many works have been done to accelerate plateau exploration for local search algorithms. In CSP and SAT, tabu search (Glover and Laguna 1997) can be used to avoid falling back to the same states on a plateau. WalkSAT (Kautz and Selman 1996) is a random-walk based algorithm that can find an exit to escape from a local minima.

There are several lines of work to accelerate plateau exploration in best-first search. First, space reduction techniques like preferred operations (Richter and Helmert 2009) and partial order reduction (Chen and Yao 2009; Chen, Xu, and Yao 2009) can effectively reduce the number of states explored by the search algorithm, and subsequently reduce the number of states on a plateau. However, space reduction approaches are indirect approaches to accelerate plateau exploration. These approaches cannot efficiently accelerate plateau exploration when preferred operators or partial order reductions are not effective. Second, Monte Carlo random walk (MRW) algorithms have been used to solve planning problems with good performance (Nakhost and Msller 2009). It is capable of escaping from local minima. However, it is slower comparing to deterministic best-first search when heuristic functions are informative.

Our proposed random-walk assisted best-first search (RW-BFS) for planning is inspired by both the MRW approach and best-first heuristic search approach. We use a best-first search procedure for planning to conduct state space search for most of the time, as best-first search gives good performance when the heuristic functions are informative. In addition, under certain conditions, a random-walk procedure is invoked to assist the best-first search.

Random-Walk Assisted Best-First Search Algorithm

Now we introduce our random walk assisted best-first search (RW-BFS) algorithm framework.

Our proposed RW-BFS algorithm is presented in Algorithm 1. It is a variant of a standard best-first search procedure. In addition to the original best-first search algorithm, RW-BFS adds a *detect plateau* check after expanding a new state (Line 13 in Algorithm 1). If a plateau is detected, the *random walk exploration* procedure will be called to explore the search space in order to find a state that can reduce h^*. Meanwhile, h^*, the incumbent heuristic value, is updated whenever a state with a smaller heuristic value is found (Line 6-7 in Algorithm 1).

Algorithm 2 presents the *random walk exploration* procedure. It essentially adopts the Monte-Carlo exploration algorithm proposed in (Nakhost and Msller 2009). Given a state s and h^*, it explores s's neighbors using a sequence of

Algorithm 1: The RW-BFS Algorithm

```
   input : Initial state s_0
1  open ← s_0 ;
2  while open is not empty do
3      s ← fetch(open);
4      if s is goal then
5          return solution found ;
6      if h(s) ≤ h* then
7          h* ← h(s) ;
8      if s is not a dead end then
9          closed ← s;
10         foreach s_i ∈ successor(s) do
11             evaluate h(s_i);
12             open ← (s_i, h(s_i)) ;
13     if detect plateau then
14         open ← random walk exploration (s, h*);
15 return no solution
```

Algorithm 2: Random Walk Exploration

```
   input : a state s, the incumbent heuristic value h*
1  s' ← s;
2  for j ← 1 to t do
3      decide parameters l, n ;
4      s' ← walk(s', l, n) ;
5      if s' is dead-end then
6          s' ← s;
7      else if h(s') < h* then
8          return s';
```

Algorithm 3: Walk

```
   input : a state s, the parameter l, the parameter n
1  c ← 0 ;
2  s' ← s ;
3  for c ← 1 to n do
4      for j ← 1 to l do
5          o ← a random applicable action in s' ;
6          s' ← apply(s', o) ;
7      if h(s') < h_min then
8          s_min ← s' ;
9  return s_min ;
```

(t) random walks. A state is returned if its heuristic function value is lower than h^* (Line 7-8 in Algorithm 2).

At each iteration, it initializes parameters l, n that are used in *walk*. Then, it invokes *walk* to visit a new state s' (Line 4). If s' is a dead-end, this algorithm will restart from the input state s (Line 5-6). If s' has even smaller heuristic value than h^*, this state will be returned to Algorithm 1. Otherwise, s' will be used as a new starting state for the next walk. The number of walks is bounded by t, so that this algorithm always returns in finite time, whether a better state s' is found or not.

Algorithm 3 gives a detailed view of the *walk* procedure. Given a starting state s and two parameters l and n, Algorithm 3 will try n paths, where each path is a random sequence of l actions. The procedure will return the best ending state among the n paths. Note that for any path yielded by Algorithm 3, heuristic functions are evaluated only at the end of the l actions (Line 7).

Plateau Detection. The performance of our algorithm also depends on the performance of the *detect plateau* procedure used in Algorithm 1.

This plateau detection test can neither be too sensitive nor too unresponsive. If it is too sensitive, the *random walk exploration* procedure will be invoked frequently and the progress of the best-first search may be hindered by constant interruption. On the other hand, if this detection is unresponsive, our designed random walks cannot help the best-first search as desired. Therefore, a balanced plateau detection mechanism is needed. In our proposed algorithm, the best-first search is currently on a plateau if the value of h^* is not reduced for m consecutive states. In Roamer, we set $m = 3000 + (n_p - 1) * 1000$, where n_p is the number of plateaus found so far.

Parameter settings. The Random Walk Exploration algorithm has a few parameters affecting its performance, among which n and l are the most important since they directly control the process of escaping from extensive local minima and plateaus. If n and l are too small, the local search method is greedy as it tries to immediately exploit their local knowledge instead of exploring the neighborhood of current state. Following a misleading heuristic value may quickly lead to a much worse state than what could be achieved with a little more exploration. On the other hand, if they are too large, the search may take a long time on exploring the neighborhood of the current state. In our algorithm, we set $t = 4$ and let $l = 1 + (10 - 1) * j/(t - 1), n = 200 + (1000 - 200) * j/(t - 1)$ for $j = 1 \cdots t$.

Multiple Heuristic Evaluations

Using multiple heuristic functions in search usually gives better performance than a single one. Since different heuristic functions sort states in *open* lists in different orders (Helmert 2006), it involves different search topologies. When one heuristic function becomes uninformative on its value plateau, other heuristics may give informative guidance and find exits on a plateau. However, extra heuristic function calculation and extra open lists may increase the overall time and space complexity of the search algorithm.

The default heuristic evaluation of LAMA planner adapts action costs, which estimates the minimum cost of a relaxed plan from current state to goal. In our experimental results, it performs not as good in domains which some actions have large costs while others are small as other domains. In our planner, we add a heuristic evaluation which estimates the length of the relaxed plan besides estimating the minimum cost of the relaxed plan. Respectively, we add an *open* list in our planner which sorts states by the order of the length of the relaxed plan. This synthetic heuristic evaluations achieve a good trade-off performance in our experimental results.

Experimental Results

In this section, we report experimental results of our planner. We evaluate performances for four planners, i.e., a baseline planner - LAMA, Roamer with random walk (Roamer^{rw}), Roamer with multiple heuristic evaluations (Roamer^{mh}), and Roamer which integrating these two techniques. We test all domains in IPC-6 (The Sixth International Planning Competition 2008), including Elevators (Elevator), Openstacks (Open), PARC printer (Parc), Peg solitaire (Peg), Scanalyzer-3D (Scan), Sokoban (Sokoban), Transport (Trans) and Woodworking (Wood). All experiments are ran on a PC workstation with a 2.40 GHz CPU and 2GB memory. The running time limit for each instance is set to 300 seconds.

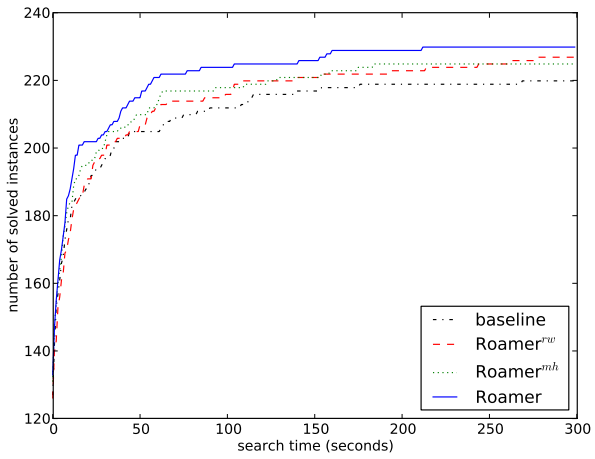

Figure 1: Number of instances (out of all the instances in the testing domains) that are solvable for a given time limit.

In Figure 1, we present the number of instances that are solvable in the testing domains with respect to a given time limit. Clearly, both Roamer^{rw} and Roamer^{mh} both solve more problem instances than the baseline planner. Roamer gives the best performance in these three algorithms.

Conclusions

In this paper, we have presented a random walk assisted best-first search algorithm, which can improve the efficacy

of heuristic search, and a multiple heuristic evaluations technique to balance the performance in different problem domains. Comparing to the baseline planner, the experimental results show that our planner, Roamer, outperforms in number of solved instances in testing domains.

Acknowledgments

This research was supported by China Scholarship Council, NSF grants IIS-0535257, DBI-0743797, IIS-0713109, and Microsoft Research New Faculty Fellowship.

References

Benton, J.; Talamadupula, K.; Eyerich, P.; Mattmüller, R.; and Kambhampati, S. 2010. G-value plateaus: A challenge for planning. In *Proc. ICAPS*, 259–262.

Chen, Y., and Yao, G. 2009. Completeness and optimality preserving reduction for planning. In *Proc. IJCAI*.

Chen, Y.; Xu, Y.; and Yao, G. 2009. Stratified planning. In *Proc. IJCAI*.

Frank, J. D.; Cheeseman, P.; and Stutz, J. 1997. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research* 7:249–281.

Glover, F., and Laguna, M. 1997. *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers.

Hampson, S., and Kibler, D. 1993. Plateaus and plateau search in boolean satisfiability problems: When to give up searching and start again. In *The 2nd DIMACS Implementation Challenge*, 437–456.

Helmert, M., and Röger, G. 2008. How good is almost perfect. In *Proc. AAAI*, 944–949.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J. 2002. Local search topology in planning benchmarks: A theoretical analysis. In *AIPS*, 92–100.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. AAAI*.

Nakhost, H., and Msller, M. 2009. Monte-carlo exploration for deterministic planning. In *Proc. IJCAI*, 1766–1771.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS*.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*.

Russell, S. J., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Pearson Education.

The Sixth International Planning Competition. 2008. http://ipc.informatik.uni-freiburg.de/.

SatPlanLM and SatPlanLM-c: Using Landmarks and Their Orderings as Constraints

Dunbo Cai
Wuhan Institute of Technology, Wuhan, China
dunbocai@gmail.com

Yanmei Hu and Minghao Yin
Northeast Normal University, Changchun, China
huym260@gmail.com ymh@nenu.edu.cn

Abstract

SatPlanLM and SatPlanLM-c are both built on top of SatPlan. The most important feather of the two planners is the exploitation of landmarks. Specifically, landmarks and their orderings are encoded as clauses, which serve as additional constraints. SatPlanLM supports the STRIPS subset of PDDL, while SatPlanLM-c additionally supports action costs in the language used in IPC-5. In this year's IPC, SatPlanLM uses *siege* as the default solver and SatPlanLM-c uses MiniSat as the default solver. We also implemented an approach in SatPlanLM-c that translates planning problems with action costs into sequences of partial weighted MaxSAT problems, which is more suitable for planning with action costs.

Introduction

SatPlanLM and SatPlanLM-c use a kind of knowledge, *landmarks* (Porteous et al. 2001; Hoffmann et al. 2003). Specifically, they encodes landmarks and their orderings of a planning task $\boldsymbol{T}$ into clauses, and integrates the clauses into the encoding of $\boldsymbol{T}$. SatPlanLM uses *siege* (http://www.cs.sfu.ca/research/groups/CL/software/siege/) as the default solver and entered the planning and learning track. SatPlanLM-c supports action costs specified in the language used in IPC-5, uses MiniSat (Eén and Sörensson 2003) as the default solver and entered the satisficing track of the deterministic part. We also designed a method in SatPlanLM-c that handles planning problems with action costs more suitably by encoding a planning task into a sequence of *partial weighted MaxSAT* problems (Fu and Malik 2006) like the way proposed by Robinson et al. (2010) and using the SMT solver Yices (http://yices.csl.sri.com/). In the rest of this paper, we first introduce our encoding schema of landmark knowledge and then describe an approach that is sensitive to action costs.

Encoding Landmarks and Their Orderings

SatPlanLM and SatPlanLM-c use the *thin-gp based encoding* in SatPlan'06 (Kautz et al. 2006). Additionally, they encode landmarks and their orderings (Richter et al. 2008) of a time bounded planning task into clauses, and then integrate these clauses into the problem encoding of the task. Landmarks and their orderings are encoded in the following way. Let $\boldsymbol{T}$ be a planning task, the time bound on $\boldsymbol{T}$ be k, and p, q be facts.

(1) For each fact that *p is a landmark*, generate a clause $p_0 \vee \ldots \vee p_k$.

(2) For each *natural ordering* $q \rightarrow p$, and for each time $i = 0 \ldots k$, generate a clause $\neg p_i \vee q_{i-1} \vee \ldots \vee q_0$.

(3) For each *necessary ordering* "$q \rightarrow_{\mathrm{n}} p$", and for each time $i = 0 \ldots k$, generate a clause $\neg p_i \vee p_{i-1} \vee q_{i-1}$. .

(4) For each *greedy necessary ordering* $q \rightarrow_{\mathrm{gn}} p$, for each time $i = 0 \ldots k$, generate a clause $\neg p_i \vee p_{i-1} \vee \ldots \vee p_0 \vee q_{i-1}$.

We make some simple optimizations in the above encoding schema. In (1), we skip a fact p if it is in initial conditions or goal conditions of $\boldsymbol{T}$. In (1), (2), (3) and (4), we use *first_level*(p), which is the first level of p in the planning graph for the current bounded task, to reduce clause length. Specifically, for a landmark p, if time step i < *first_level*(p), then p_i is removed from the respective clauses, as p_i cannot be true in these clauses.

SatPlan and SatPlanLM-c uses LAMA (Richter et al. 2008) to compute landmarks and the corresponding orderings. Specifically, a planning task $\boldsymbol{T}$ is first given to LAMA for extracting landmarks and landmark orderings. The knowledge is output to an external file. After that, the description of $\boldsymbol{T}$ and the file are given to SatPlan'06, where encodings of $\boldsymbol{T}$ and the landmark knowledge of $\boldsymbol{T}$ are

integrated. Note that in SatPlanLM-c the encoding produced above are given to MiniSat. It is obvious that the approach is not cost sensitive. However, for efficiency concern, we use this approach as the default configuration for SatPlanLM-c.

An Action Cost Sensitive Approach

In SatPlanLM-c, we also implemented an approach that is sensitive to costs of actions. It is like that proposed by Robinson et al. (2010). The most important idea is encoding an action $\boldsymbol{a}$ with positive cost $\boldsymbol{c}$ into a unit clause $\neg\boldsymbol{a}$ whose weight is $\boldsymbol{c}$. The difference is that we use the *thin-gp based encoding* to present hard constrains and we use the SMT solver Yices. Also note that, our approach doesn't guarantee optimality in terms of action costs. The effect of landmark knowledge on the approach is under test and will be studied in our future work.

Acknowledgement

Special thanks to authors of SatPlan, LAMA, MiniSat and Yices for sharing their codes and binaries.

References

Eén, N., and Sörensson, N. 2003. An Extensible SAT-solver. In *Proc. SAT'03*.

Fu, Z., and Malik, S. 2006. On solving the partial max-sat problem. In *Proc. SAT'06*.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2003. Ordered landmarks in planning. *JAIR* 22:215–278.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. SatPlan: Planning as Satisfiability. In *Abstracts IPC-5*.

Robinson, N.; Gretton, C.; Pham, D.; and Sattar, A. 2010. Cost-Optimal Planning using Weighted MaxSAT. In *Proc. ICAPS'10 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems.*

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *Proc. ECP'01*.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proc. AAAI'08*.

Extending Temporal Planning for the Interval Class

Bharat Ranjan Kavuluri
bharatranjan@gmail.com
AIDB Lab, IIT Madras, Chennai
Tamilnadu, India 600036

Abstract

The paper presents a planner called Sharaabi. It is an extension to DRIPS, a planner which tackled required concurrency with overall preconditions requiring continuous support. The paper describes the motivation behind the work and the way the planner operates. The extensions to DRIPS are described in brief. Examples are provided to show that the planner works.

Introduction

The solutions of temporal planning problems more often than not require the concurrent execution of durative actions due to goal deadlines or interactions among actions. The latter case has been labeled as Coordination/Required Concurrency (RC)(Halsey, Long, and Fox 2004; Cushing et al. 2007) and the problems for which it happens have been categorized as temporal planning problems with coordination/required concurrency. The planner CRIKEY (Halsey, Long, and Fox 2004; Coles et al. 2009b) has been specifically designed to handle this class of problems by maintaining envelopes of possibly interacting actions. It searches in the space of split actions- where a durative action D is divided into two point actions D_{start} and D_{end}. The at-start effect and preconditions are added to D_{start} and the at end preconditions and effects are added to D_{end}. The invariants are maintained in a separate queue to check for violations. At each step, CRIKEY decides on actions to add to the plan, If it chooses a start action, it updates the state description with immediate effects and creates a new envelope. If it chooses an end action, it must choose a start action to pair it with and close an envelope. It uses a simple temporal network (Dechter, Meiri, and Pearl 1991) to check for the validity of the current plan under consideration after every such addition of an action. There have been several variants to CRIKEY, notably CRIKEYshe (Coles et al. 2009b), and CRIKEY3 (Coles et al. 2008) and COLIN (Coles et al. 2009a) (which maintains a LP problem to tackle continuous durative planning). POPF (Coles et al. 2010) is a forward state space temporal POCL planner. Instead of committing on the start times of actions as in SAPA, or committing and then lifting a partial order in CRIKEY, LPG.s etc., POPF maintains a partial order from the start and tries to exploit this order for solving RC problems.

The Invariant Class

All these planners except (Coles et al. 2010) do not solve a class of problems we labelled The Invariant Class. The notable feature of this class is that every problem has all its plans requiring atleast one action with an overall precondition that needs to be supported/covered by an overlap of actions providing/establishing the condition. A few motivating examples are shown in the figure 1. Figure 1(a) shows the type of problems which the planners described in the previous paragraph can solve- problems where each producer supports multiple consumers. The problems shown in figure 1(b) require at least one consumer to be supported by multiple producers. The reason the above planners can not tackle these problems is because they do not generate the necessary temporal constraints which force the producers to overlap. In the absence of those constraints, each check for schedulability either takes a significant amount of time and the planner gets stuck or the planner returns invalid plans/no plan messages.

DRIPS

I designed a planner called DRIPS (Kavuluri 2010) to tackle this class of problems. DRIPS is a modification of SAPA (Minh and Kambhampati 2003). SAPA is a metric-temporal planner which searches in the space of time stamped states. It checks for applicable actions whenever an event occurs. Events occur when an action is applied on a state or when delayed effects of actions are applied on a state from its event queue -the time is advanced to the time stamp of the event. SAPA was modified by adding two more types of states after the at start effects of an action are applied.

- A state which signifies the time just before the end of each action
- A state which indicates the passage of one clock tick without any action

This causes the planner to check for actions before the producers finish and also causes the planner to generate all possibilities at each clock tick. In the event the planner fails to

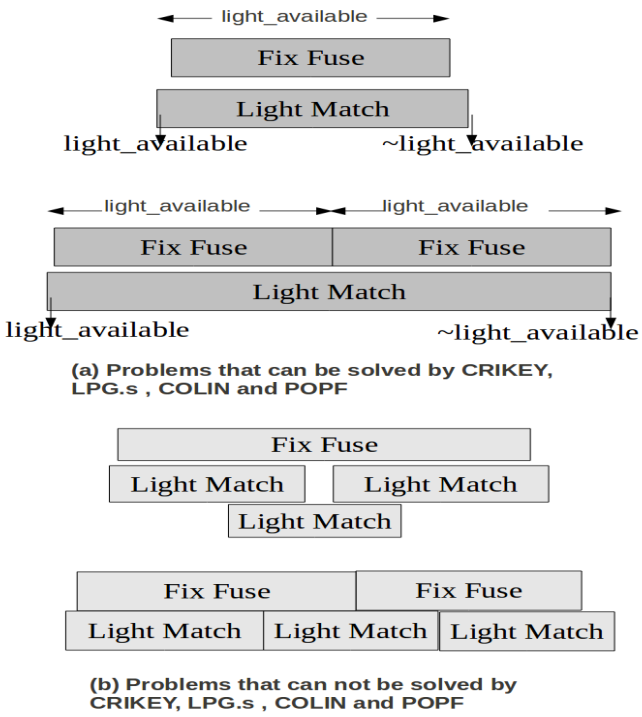

Figure 1: Required Concurrency and Invariant Class

find a plan using the first type of states, the clock tick based states ensure that completeness is not lost. So, given enough time and memory, if the given problem has a plan whose step size is not smaller than the clock tick, DRIPS theoretically should find a plan. More details about DRIPS are available in (Kavuluri 2010).

Sharaabi

The entry for this planning competition Sharaabi consists of modifications that have been made to DRIPS upto this point. The modifications are listed below.

- Sharaabi tackles derived predicates
- Sharaabi tackles problems requiring time-aware temporal planning

Both features of Sharaabi are described here with example problems.

Handling Derived Predicates

Derived predicates were implemented by implementing a class called Predicate formula and associating it with each derived predicate class. The derived predicates are evaluated when testing whether the action is applicable. Once the derived predicate is checked, its truth value is stored in the state so that it need not be computed again. The figures 2, 3 and 7 give an example match problem involving derived predicates where you need either a match or a bulb to be on to mend the fuse. The derived predicate bright becomes true when either of the derived predicates lightavail and bulbavail is true.

```
(define (problem fixfuse)
(:domain matchcellar)
(:objects
match1 - match
bulb - bulb
fuse1 fuse2 - fuse)
(:init
(unused match1)
(working bulb)
(handfree))
(:goal (and (mended fuse1))))
```

Figure 2: Derived Predicate Example- Problem

```
;; Search time 22 milisecs
;; State generated: 9 State explored: 4
0.0: (LIGHT_MATCH match1) [7.0]
0.0: (MEND_FUSE fuse1) [9.0]
6.99: (Put_on bulb) [4.0]
```

Figure 3: Plan for the derived match problem

```
(at start
(assign (convoy-tail-left-at ?v1 ?e) (+ t (/ (convoy-length ?c)
(convoy-speed ?c))))) - effect
(at start
(<= (convoy-tail-left-at ?v1 ?e) (+ (+ t (- 0 (+ (inner-
convoy-distance) (convoy-edge-travel-time ?e)))) (?dura-
tion)))) - precondition
```

Figure 4: Time-aware temporal planning example

```
(define (problem fixfuse)
(:domain TAPmatchcellar)
(:objects
match1 match2  match
fuse1 fuse2 - fuse)
(:init
(unused match1) (unused match2)
(= (cansolve) 7.0) (handfree) (q))
(:goal (and (mended fuse2)))))
```

Figure 5: Time-aware temporal planning example contd..

```
;; Search time 14 milisecs
;; State generated: 12 State explored: 8
0.0: (setTime) [1.0]
5.0: (LIGHT_MATCH match2) [15.0]
5.0: (MEND_FUSE fuse2 match2) [9.0]
```

Figure 6: Time-aware temporal planning example contd..

```
(define (domain matchcellar)
(:requirements :typing :durative-actions :derived-predicates)
(:types match fuse bulb)
(:predicates
(light ?match - match)
(bulblight ?bulb - bulb)
(switchon ?bulb - bulb)
(handfree)
(unused ?match - match)
(working ?bulb - bulb)
(mended ?fuse - fuse))
(:durative-action LIGHT_MATCH
:parameters (?match - match)
:duration (= ?duration 7)
:condition (and
(at start (unused ?match))
(over all (light ?match)))
:effect (and
(at start (not (unused ?match)))
(at start (light ?match))
(at end (not (light ?match)))))
(:durative-action Put_on
:parameters (?bulb - bulb)
:duration (= ?duration 4)
:condition (and
(at start (working ?bulb)))
:effect (and
(at start (not (working ?bulb)))
(at start (bulblight ?bulb))
(at start (switchon ?bulb))
(at end (not (switchon ?bulb)))
(at end (not (bulblight ?bulb)))))

(:durative-action MEND_FUSE
:parameters (?fuse - fuse)
:duration (= ?duration 9)
:condition (and
(at start (handfree))
(over all (bright)))
:effect (and
(at start (not (handfree)))
(at end (mended ?fuse))
(at end (handfree))))
(:derived (lightavail)
(and ((exists (?z) (light ?z)))))
(:derived (bulbavail)
(and ((exists (?z) (bulblight ?z)))))
(:derived (bright) (or ((lightavail) (bulbavail)))))
```

Figure 7: Example for Derived Predicates- Domain

Time-aware temporal planning

Time-aware temporal planning indicates that the current time and action durations are part of actions preconditions and effects. An example of the same is provided in figure 4. The first expression (effect) assigns the value of the current time plus the time the convoy takes to travel a distance equal to its length to the convoy-tail-left-at function. The precondition verifies if the convoy-tail-left-at function satisfies the inter-convoy distance to be maintained if the convoy were to start now.

This was implemented by using the $\#t$ and $?duration$ features in SAPA. This was done to add a function called time which returns the current time of the state when it is called. An example is shown in the figures 5, 6 and 8 to demonstrate this capability. The time when the planner can begin to solve the problem is set by the setTime action and this is verified in the preconditions of the action LIGHT_MATCH. The domain and problem files are given in figures 5 and 6. The plan is shown in Figure 8.

```
(define (domain TAPmatchcellar)
(:requirements :fluents :typing :durative-actions)
(:types match fuse)

(:predicates
(light ?m - match)
(handfree)
(unused ?m - match)
(mended ?f  fuse) (p) (q) )

(:functions (cansolve) )

(:durative-action setTime
:parameters ()
:duration (= ?duration 1)
:condition (and (at start (q)))
:effect (and (at start (assign cansolve 5.0))
(at end (p)) (at start (not (q)))) )

(:durative-action LIGHT_MATCH
:parameters (?m - match)
:duration (= ?duration 15)
:condition (and
(at start (p))
(at start ( == cansolve t))
(at start (unused ?m)) (over all (light ?m)))
:effect (and
(at start (not (unused ?m)))
(at start (light ?m))
(at end (not (light ?m)))))

(:durative-action MEND_FUSE
:parameters (?f - fuse ?m - match)
:duration (= ?duration 9)
:condition (and (at start (handfree))
(over all (light ?m)))
:effect (and (at start (not (handfree)))
(at end (mended ?f))
(at end (handfree)))))
```

Figure 8: Time-aware temporal planning example Domain

Conclusions and Future Work

This paper presents Sharaabi, an extension to DRIPS which deals with Derived predicates and Time-aware temporal

planning. The paper explains in brief how these features were implemented and demonstrates the planners capabilities with brief examples. Currently we are verifying the implementation for scalability and coverage. Future work would entail improving the efficiency of the planner like looking at different heuristic functions to improve speed.

References

Coles, A.; Fox, M.; Long, D.; and Smith, A. 2008. Planning with Problems Requiring Temporal Coordination. In *AAAI-2008*.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2009a. Temporal Planning in Domains with Linear Processes. In *IJCAI 2009*.

Coles, A.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. 2009b. Managing concurrency in temporal planning using planner-scheduler interaction. *AI Magazine*.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. . In *Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. 2007. When is temporal planning really temporal? In *IJCAI*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence*.

Halsey, K.; Long, D.; and Fox, M. 2004. CRIKEYA planner looking at the integration of scheduling and planning. In *Proceedings of the Workshop on Integration of Scheduling Into Planning at 13th International Conference on Automated Planning and Scheduling (ICAPS '03)*.

Kavuluri, B. R. 2010. Required Concurrency without a Scheduler. In *28th Workshop of UK Planning and Scheduling Special Interest Group*.

Minh, D. B., and Kambhampati, S. 2003. SAPA: A Scalable Multi-Objective Metric Temporal Planner. *JAIR*.

YAHSP2: Keep It Simple, Stupid

Vincent Vidal
ONERA – DCSD
Toulouse, France
Vincent.Vidal@onera.fr

Abstract

The idea of computing lookahead plans from relaxed plans and using them in the forward state-space heuristic search YAHSP planner has first been published in 2003. We show in this paper that this simple idea still leads to very efficient planners in comparison with state-of-the-art planners, in terms of running time. We describe the new implementation of lookahead search that has been made in the second version of the YAHSP planner, which has been considerably simplified since the first implementation. We then show through an extensive comparison, over all existing IPC benchmarks, that the resulting YAHSP2 planner outperforms state-of-the-art planners in terms of cumulated number of solved problems and running time. We also briefly describe YAHSP2-MT, an attempt to parallelize YAHSP2 for multi-core machines with shared memory.

Introduction

Since the 6th edition of the deterministic part of the International Planning Competition (IPC) in 2008, an emphasis has been put on solution quality rather than on speed in computing a single plan. In the 2008 and 2011 competitions, deterministic planners were run during a fixed amount of time and their objective was to find the best possible plan within this time constraint. Although for real-world applications plan quality is generally as important as finding a solution (if not even more), we think that designing fast planners is still a relevant task. Particularly, deterministic planners can be embedded into wider systems that frequently call them with different initial states, goals or even domain definitions, and use the solution plans for a particular objective. For example, the probabilistic planners FF-Replan (Yoon, Fern, and Givan 2007) and RFF (Teichteil-Königsbuch, Kuter, and Infantes 2010), winners of the probabilistic tracks of the non-deterministic IPCs in 2004 and 2008 respectively, make heavy use of the FF planner to solve determinized problems extracted from the probabilistic one. They then combine the solutions given by FF into a policy for the probabilistic problem. Another example is the DAE_X planner (Bibai et al. 2010; Dréo et al. 2011), which embeds the YAHSP planner (Vidal 2004) into an evolutionary algorithm whose objective is to produce optimized plans. Optimization is performed through the evolution of a population of individuals, which represent sequences of intermediate goals that must be reached in turn from the initial state to the goal of the problem, by successive calls to YAHSP with an upper bound on the number of expanded nodes. Within a typical single run of DAE_X during 30 minutes, YAHSP may be called hundreds of thousands of times. The need for a fast planner to embed in DAE_X motivated the design of the YAHSP2 planner. Indeed, in opposition to modern planners such as LAMA (Richter and Westphal 2010) which require heavy preprocessing techniques for each different problem, even on the same domain (transformation to SAS+, landmark generation, landmark orderings, etc.), YAHSP does not perform any preprocessing, computing everything on-the-fly during search. Embedded into a wider system, search in YAHSP for a new initial state and goal can then be performed immediately, allowing fast and frequent calls.

The goals in the design of a new version of YAHSP were (1) to extend its expressivity to cost-based and temporal planning, and (2) to simplify its implementation with efficiency in mind. The former has been easily performed: YAHSP2 simply does not take into account costs and durations when computing a single solution, and performs a post-deordering (Bäckström 1998) of the sequential solution plans to produce concurrent plans (de facto forbidding temporally expressive planning). The idea behind this is that the planner embedded into DAE_X should concentrate on the task of finding a plan, working only on the combinatorial problem, the optimization being held by the evolutionary algorithm. In order to fully use the time contract of 30 minutes in the IPC, search in YAHSP2 alone is pursued when solutions are found and states whose cost (plan length, sum of action costs, or makespan after deordering) exceeds the best cost found so far are pruned. One subtlety is that deordering for temporal planning is made during search, in order to be able to perform that pruning. The latter goal in the design of YAHSP2, simplicity, has consisted in simplifying the way relaxed plans and lookahead plans are computed, and removing many other ideas introduced in the first version of YAHSP which were not strictly needed to reach good performance. Indeed, some of these ideas were useful in some cases on the very limited number of benchmarks available when YAHSP has been conceived, but do not reveal to be that interesting when performing experiments on the full set of benchmarks which is now available.

We provide in this paper a complete picture of the techniques and algorithms used in the YAHSP2 planner as it has been entered into the 7th International Planning Competition. We also show through an extensive experimental evaluation that YAHSP2 improves the state-of-the-art (before the competition!) in terms of cumulated number of solved problems and running time efficiency for finding a single plan. We finish by a short description of YAHSP2-MT, an attempt to benefit from multi-core processors in lookahead heuristic search planning previously detailed in (Vidal, Bordeaux, and Hamadi 2010).

Background

The basic STRIPS model of planning can be defined as follows. A *state* of the world is represented by a set of ground atoms. A *ground action* a built from a set of atoms A is a tuple $\langle pre(a), add(a), del(a)\rangle$ where $pre(a) \subseteq A$, $add(a) \subseteq A$ and $del(a) \subseteq A$ represent the preconditions, add effects and del effects of a respectively. A *planning problem* can be defined as a tuple $\Pi = \langle A, O, I, G\rangle$, where A is a finite set of *atoms*, O is a finite set of ground actions built from A, $I \subseteq A$ represents the *initial state*, and $G \subseteq A$ represents the *goal* of the problem. The *application* of an action a to a state s is possible if and only if $pre(a) \subseteq s$ and the resulting state is defined by $s' = (s \setminus del(a)) \cup add(a)$. A *solution plan* is a sequence of actions $\langle a_1, \ldots, a_n\rangle$ such that for $s_0 = I$ and for all $i \in \{1, \ldots, n\}$, the intermediate states defined by $s_i = (s_{i-1} \setminus del(a_i)) \cup add(a_i)$ are such that $pre(a_i) \subseteq s_{i-1}$ and $G \subseteq s_n$. This simple STRIPS model has been enriched in many ways through the evolution of PDDL. However, the objective in the design of YAHSP2 is to consider the combinatorial difficulty of finding a solution plan only, and thus we stick to the basic STRIPS model. Action costs and durations are simply ignored, a temporal plan being obtained by a deordering of a valid sequential plan.

The *lookahead strategy* implemented in the first version of YAHSP has been described in (Vidal 2004). Briefly, the idea is to produce in polynomial time a sequence of actions that hopefully can bring search closer to a goal state, and to introduce this state in the open list of a best-first search algorithm just as if it was a normal state. To this end, relaxed plans (Hoffmann and Nebel 2001) which are often of high quality are used in YAHSP to compute such a sequence. This is performed by a simple algorithm which tries to apply as much actions as possible from a relaxed plan to the state for which it has been computed. When no more action can be applied, a simple *repair strategy* tries to replace an action of the relaxed plan by another one, taken from the global set of actions, which can be applied and produces an unsatisfied precondition of another action in the relaxed plan. The idea of producing lookahead plans and states has been recently enriched, for example by the computation of low-conflicts relaxed plans and a repair strategy based on insertion instead of replacement (Baier and Botea 2009), or by computing lookahead plans in a different way than extracting them from relaxed plans, using sophisticated techniques such as landmarks and causal chains (Lipovetzky and Geffner 2011).

YAHSP2: The Algorithms

In the design of the second version of the YAHSP planner, we took the opposite direction: instead of augmenting the techniques and components used inside the planner, we simplified its design and removed many unnecessary steps, following in that the KISS principle: "Keep It Simple, Stupid". The motivations behind this work were first to implement a planner that could be easy to maintain and to embed into a wider system such as DAE_X, and second to better understand what makes YAHSP an efficient planner. Indeed, if some ideas were sometimes useful on the small set of benchmarks available when YAHSP was written, experiments on the much larger set of benchmarks now available changes the picture. The implementation, with respect to the version described in (Vidal 2004), has been modified and simplified in the following main ways:

- The relaxed plans used to compute lookahead plans are not any more computed from relaxed planning graphs. We found more convenient and easy to extract relaxed plans directly from the computation of a critical path heuristic such as h^{add} or h^1: all what is needed is a cost associated to each action. This has the advantage to avoid the need of complex data structures to build planning graphs, and considerably simplifies the algorithm.
- The heuristic value of states is no longer the length of relaxed plans, but the h^{add} value of the goal set. Among several variants that we have experimented, we found that using h^{add} for both evaluating states and extracting lookahead plans was a good strategy.
- Some refinements introduced in YAHSP are abandoned, due to their lack of robustness on the whole set of benchmarks. Among them are helpful actions first introduced in FF and used in YAHSP to define a lexicographic order on the nodes to be expanded (always preferring nodes coming from the application of an helpful action). Although some recent experiments show that they may be of interest (Richter and Helmert 2009), their use in YAHSP finally does not reveal to be that interesting. Also, goal-preferred actions (actions that do not delete a goal) which were used to compute twice a relaxed planning graph: the first one with goal-preferred actions only, and the second one with all actions of the problem in case of a failure in reaching the goals, are not used any more.

The simplified design of YAHSP2 allows us to completely describe the algorithms, which are implemented in around 450 lines of C code. The prerequisites are a parsing and grounding process (without any complex preprocessing such as mutex, landmarks, etc.), and a few helpers to easily access some data (in particular, the list of actions which consume, add and delete an atom are precomputed). States are implemented with bit vectors such that checking the presence of an atom in a state is performed in constant time. The open and closed lists are represented with red-black trees. Nodes of the search tree are tuples $n = \langle s, p, t, l, f, a\rangle$ where s is a state, p is the parent node of n, t is the sequence of actions (a single action for a classical transition, a sequence for lookahead states) yielding n from p, l is the length of

Algorithm 1: plan-search

input : a planning problem $\Pi = \langle A, O, I, G \rangle$ and a weight ω for the heuristic function
output : a plan if search succeeds, $\bot$ otherwise

$open \leftarrow closed \leftarrow \emptyset$
create a new node n:
  $n.state \leftarrow I$
  $n.parent \leftarrow \bot$
  $n.steps \leftarrow \langle\rangle$
  $n.length \leftarrow 0$
$n' \leftarrow$ `compute-node`$(\Pi, \omega, n, open, closed)$
if $n' \neq \bot$ **then return** `extract-plan`(n')
else
  while $open \neq \emptyset$ **do**
    $n \leftarrow \arg\min_{n \in open} n.heuristic$
    $open \leftarrow open \setminus \{n\}$
    foreach $a \in n.applicable$ **do**
      create a new node n':
        $n'.state \leftarrow (n.state \setminus del(a)) \cup add(a)$
        $n'.parent \leftarrow n$
        $n'.steps \leftarrow \langle a \rangle$
        $n'.length \leftarrow n.length + 1$
      $n'' \leftarrow$ `compute-node`$(\Pi, \omega, n', open, closed)$
      if $n'' \neq \bot$ **then return** `extract-plan`(n'')
  return $\bot$

Algorithm 2: compute-node

input : a planning problem $\Pi = \langle A, O, I, G \rangle$, a weight ω for the heuristic function, a node n, the $open$ and $closed$ lists
output : a goal node if search succeeds, $\bot$ otherwise; $open$ and $closed$ are updated

if $\exists\, n' \in closed \mid n'.state = n.state$ **then return** $\bot$
else
  $closed \leftarrow closed \cup \{n\}$
  $\langle cost, app \rangle \leftarrow$ `compute-hadd`$(\Pi, n.state)$
  $gcost \leftarrow \Sigma_{g \in G}\, cost[g]$
  if $gcost = 0$ **then return** n
  else if $gcost = \infty$ **then return** $\bot$
  else
    $n.applicable \leftarrow app$
    $n.heuristic \leftarrow n.length + \omega \times gcost$
    $open \leftarrow open \cup \{n\}$
    $\langle state, plan \rangle \leftarrow$ `lookahead`$(\Pi, n.state, cost)$
    create a new node n':
      $n'.state \leftarrow state$
      $n'.parent \leftarrow n$
      $n'.steps \leftarrow plan$
      $n'.length \leftarrow n.length + \mathrm{length}(plan)$
    return `compute-node`$(\Pi, \omega, n', open, closed)$

the plan reaching n from the initial state, f is the numerical heuristic evaluation of s and a is the set of actions applicable in s. The notations $n.state$, $n.parent$, $n.steps$, $n.length$, $n.heuristic$ and $n.applicable$ refer to s, p, t, l, f and a respectively. The operator $\oplus$ concatenates two sequences.

Algorithm 1 (`plan-search`) constitutes the core of the best-first search algorithm (a weighted-A* here). The first call to `compute-node` allows to find a solution to the problem without search, by recursive calls to the lookahead process. Nodes are extracted from the open list following their heuristic evaluation and are expanded with the applicable actions (already computed and stored in nodes inserted into the open list), and a solution plan is returned as soon as possible. In the version submitted to the 7th IPC, search is pursued in order to improve the solution, with pruning of partial plans whose quality is lower than that of the best plan found so far. Also, the weight ω is set to 3.

Algorithm 2 (`compute-node`) first performs duplicate state detection, even if the quality (length, cost or makespan) of the plan which yields such a state is improved; as we deliberately avoid optimization. It then computes the heuristic, checks if the goal is obtained or contrarily cannot be reached, and updates the node with the heuristic and the applicable actions given by `compute-hadd`. The node is then stored in the open list and a lookahead state/plan is computed by a call to `lookahead`. A new node corresponding to the lookahead state is then created and `compute-node` is recursively called. Recursion is stopped when a goal state, a duplicate state or a dead-end state is reached.

Algorithm 3 (`compute-hadd`) computes h^{add} and returns a vector of costs for all atoms and actions, as well as actions applicable in the state for which h^{add} is computed obtained as a side-effect. Several ways are possible to compute h^{add}, e.g. by mutually recursive functions triggered by the updates; the one shown here has the advantage to be very simple and efficient, even if it looks laborious at first sight because of multiple iterations over the whole set of actions.

Algorithm 4 (`lookahead`) computes a lookahead state/plan from a relaxed plan given by a call to `extract-relaxed-plan`. Once a first applicable action of the relaxed plan is encountered, it is appended to the lookahead plan and the lookahead state is updated. A second applicable action is then sought from the beginning of the relaxed plan, and so on. When no applicable action is found, a repair strategy tries to find an applicable action of minimum cost from the whole set of actions, in order to replace an action of the relaxed plan which produces an unsatisfied precondition of another action of the relaxed plan, and the process loops.

Algorithm 5 (`extract-relaxed-plan`) computes a relaxed plan from a vector of action costs. A sequence of goals to produce is maintained, starting from the goals of the problem. The first one is extracted, and an action which produces it with the lowest cost is selected and stored in the relaxed plan. Its preconditions are appended to the sequence of goals, and the process loops until the sequence of goals is empty. An atom already satisfied, i.e. produced by an action of the relaxed plan, is not considered twice. The relaxed plan is finally sorted before being returned, by increasing costs first, and for equal costs by trying to order first an action which does not delete a precondition of the next action.

Algorithm 3: compute-hadd

input : a planning problem $\Pi = \langle A, O, I, G\rangle$ and a state s
output : the vector of action and atom costs and the set of actions applicable in s

foreach $a \in O$ **do**
  $cost[a] \leftarrow \infty$
  $update[a] \leftarrow (pre(a) = \emptyset)$
foreach $p \in A$ **do**
  if $p \in s$ **then**
    $cost[p] \leftarrow 0$
    foreach $a \in O \,|\, p \in pre(a)$ **do**
      $update[a] \leftarrow true$
  else $cost[p] \leftarrow \infty$
$app \leftarrow \emptyset$
$loop \leftarrow true$
while $loop$ **do**
  $loop \leftarrow false$
  foreach $a \in O$ **do**
    if $update[a]$ **then**
      $update[a] \leftarrow false$
      $c \leftarrow \Sigma_{p \in pre(a)}\, cost[p]$
      if $c < cost[a]$ **then**
        $cost[a] \leftarrow c$
        if $c = 0$ **then** $app \leftarrow app \cup \{a\}$
        foreach $p \in add(a)$ **do**
          if $c + 1 < cost[p]$ **then**
            $cost[p] \leftarrow c + 1$
            foreach $a \in O \,|\, p \in pre(a)$ **do**
              $loop \leftarrow true$
              $update[a] \leftarrow true$
return $\langle cost, app\rangle$

Algorithm 4: lookahead

input : a planning problem $\Pi = \langle A, O, I, G\rangle$, a state s, and a vector of action costs $cost$
output : a lookahead state and a lookahead plan

$plan \leftarrow \langle\rangle$
$rplan \leftarrow$ `extract-relaxed-plan`$(\Pi, s, cost)$ `// with` $rplan = \langle a_1, \ldots, a_n\rangle$
$loop \leftarrow true$
while $loop$ **do**
  $loop \leftarrow false$
  if $\exists\, i \in \{1, \ldots, n\} \,|\, pre(a_i) \subseteq s$ **then**
    $loop \leftarrow true$
    $i \leftarrow \min(i \in \{1, \ldots, n\} \,|\, pre(a_i) \subseteq s)$
    $s \leftarrow (s \setminus del(a_i)) \cup add(a_i)$
    $plan \leftarrow plan \oplus \langle a_i\rangle$
    $rplan \leftarrow \langle a_1 \ldots, a_{i-1}, a_{i+1}, \ldots, a_n\rangle$
  else
    $i \leftarrow j \leftarrow 1$
    while $\neg loop \wedge i \leq n$ **do**
      while $\neg loop \wedge j \leq n$ **do**
        if $i \neq j \wedge add(a_i) \cap pre(a_j) \neq \emptyset$ **then**
          $candidates \leftarrow \{a \in O \,|\, pre(a) \subseteq s \wedge add(a_i) \cap pre(a_j) \cap add(a) \neq \emptyset\}$
          if $candidates \neq \emptyset$ **then**
            $loop \leftarrow true$
            $a \leftarrow \arg\min_{a \in candidates} cost[a]$
            $rplan \leftarrow \langle a_1 \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n\rangle$
        $j \leftarrow j + 1$
      $i \leftarrow i + 1$
return $\langle s, plan\rangle$

Algorithm 5: extract-relaxed-plan

input : a planning problem $\Pi = \langle A, O, I, G\rangle$, a state s, and a vector of action costs $cost$
output : a relaxed plan for Π

$rplan \leftarrow \langle\rangle$
$goals \leftarrow \langle g \,|\, g \in G\rangle$
$satisfied \leftarrow s$
while $goals \neq \emptyset$ **do**
  $g \leftarrow$ pop-first$(goals)$
  if $g \notin satisfied$ **then**
    $satisfied \leftarrow satisfied \cup \{g\}$
    $a \leftarrow \arg\min_{a \in O \,|\, g \in add(a)} cost[a]$
    if $a \notin rplan$ **then**
      $rplan \leftarrow rplan \oplus \langle a\rangle$
      $goals \leftarrow goals \oplus \langle p \,|\, p \in pre(a)\rangle$
sort $rplan = \langle a_1, \ldots, a_n\rangle$: $\forall a_i, a_j \in rplan \,|\, i < j$, $cost[a_i] < cost[a_j] \vee (cost[a_i] = cost[a_j] \wedge (del(a_i) \cap pre(a_j) = \emptyset$ if possible))
return $rplan$

Experiments

We performed extensive experiments on the whole set of benchmarks, from the 1st to the 6th IPC, that YAHSP2 can handle (i.e. without ADL and numerical domains). The objective of the experiments is to demonstrate that a simple heuristic search planner with a lookahead strategy is competitive with the state-of-the-art in terms of number of solved problems and running time. All experiments are performed on an Intel Xeon X5670 running at 2.93GHz with 4GB of memory and a timeout of 30 minutes.

Sequential Planning

Seven planners are compared on 1534 sequential planning problems. Costs have been removed from domains of the 6th IPC, in order to run planners that do not accept them such as FF and LPG-td. The planners are FF (Hoffmann and Nebel 2001), LAMA (Richter and Westphal 2010), LPG-td (Gerevini, Saetti, and Serina 2003), Mp (Rintanen 2010), SGPlan6 (Chen, Wah, and Hsu 2006), YAHSP version 1 with two different settings: $Y1_{lbfs}$ similar to YAHSP2 and $Y1_{lobfs}$ with the "optimistic" strategy (i.e. expanding first nodes coming from the application of an helpful action), and YAHSP2. Most of these planners have been awarded at previous IPCs, except the recent Mp and YAHSP2 planners. We included Mp as it is the first SAT-based planner competitive with other types of satisficing planners (Rintanen 2010).

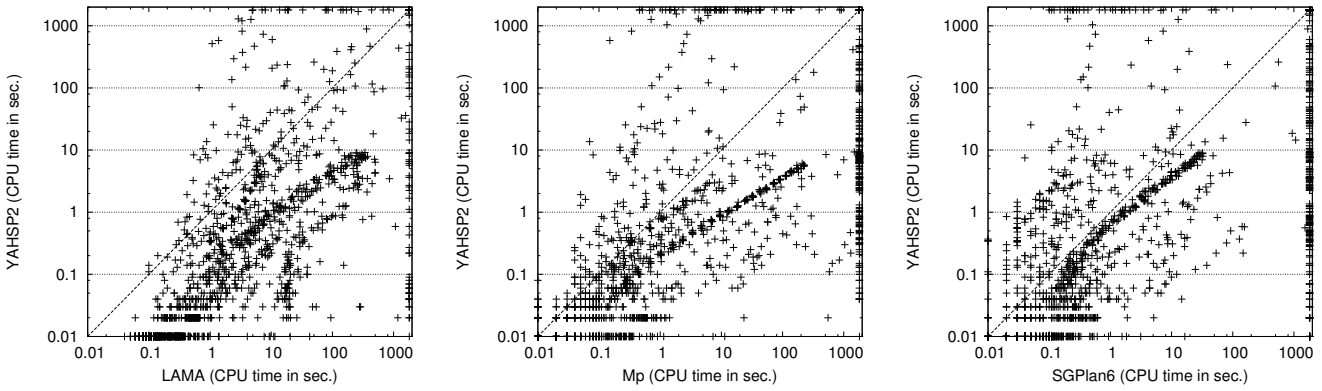

Figure 3: Comparison of the total running time for the three best sequential planners (except YAHSP1) versus YAHSP2.

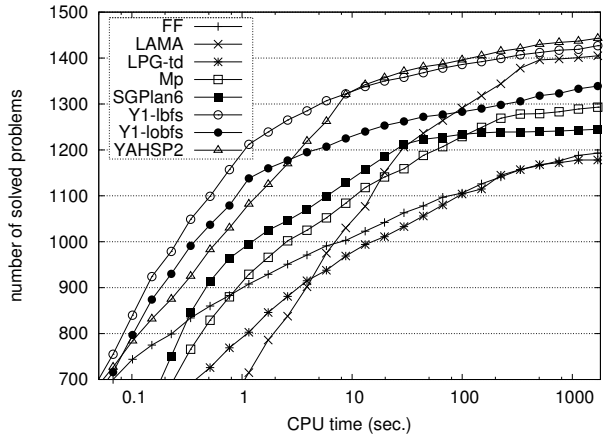

Figure 1: Cumulated number of solved problems for sequential planners in function of the total running time.

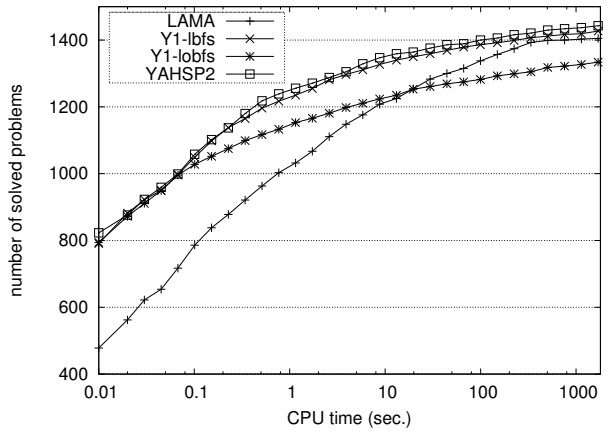

Figure 2: Cumulated number of solved problems in function of the search time for the four best sequential planners.

Figure 1 shows the cumulated number of solved problems in function of the total running time. For each CPU time t on the x axis, the corresponding value on the y axis gives the number of problems solved in under t seconds. YAHSP2 and Y1_{lbfs} clearly outperform the other planners. Y1_{lbfs} is a bit faster than YAHSP2 for problems solved in under 10 seconds, but YAHSP2 finally solves more problems. This mainly comes from the parser of YAHSP1, which has been better designed and is much more efficient than that of YAHSP2. The comparison with Y1_{lobfs} clearly confirms that giving priority to nodes coming from helpful actions was finally not a so good idea, in conjunction with the lookahead strategy. LAMA nearly reaches YAHSP2, solving 1405 problems (91.6%) with respect to 1444 (94.1%) for YAHSP2, but is significantly slower than YAHSP2. One reason is that it performs a heavy preprocessing step in order to translate to SAS+ and to compute landmarks, but Figure 2 which compares the search time only of the four best planners shows that search in LAMA is less efficient than in YAHSP2 and Y1_{lbfs}. It should be mentioned that although (Rintanen 2010) shows that Mp outperforms LAMA, this is probably due to the 300 seconds timeout which clearly disadvantages LAMA: on small runtimes it is the slowest among all planners compared here, but finally is in the top three. Figure 3 depicts scatter plot comparisons of the running time between YAHSP2 and the three other best planners (except YAHSP1), which are LAMA, Mp and SGPlan6. YAHSP2 very often outperforms them by several orders of magnitude. Finally, Table 1 shows the detail of the number of solved problems, over each IPC and each domain.

Temporal Planning

Four planners are compared on 664 temporal planning problems. The planners are LPG-td, SGPlan6, TFD (Eyerich, Mattmüller, and Röger 2009) and YAHSP2. The first three ones have been awarded at previous IPCs.

Figure 4 shows the cumulated number of solved problems in function of the total running time. YAHSP2 outperforms all planners, solving 594 problems (89.5%) against 434 problems (65.4%) for SGPlan6, the second best planner. SGPlan6 outperforms LPG-td that solves 403 problems (60.7%), which itself outperforms TFD that solves 287 problems (43.2%). Figure 5 depicts scatter plot comparisons of

IPC	domain	#pbs	#solved (difference with best)							
			FF	LAMA	LPG-td	Mp	SGPlan6	$Y1_{lbfs}$	$Y1_{lobfs}$	YAHSP2
1	grid	5	**5**	**5**	**5**	4 (1)	**5**	**5**	**5**	**5**
	gripper	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
	logistics	35	**35**	**35**	29 (6)	22 (13)	**35**	**35**	**35**	31 (4)
	movie	30	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
	mprime	35	34 (1)	**35**	**35**	**35**	33 (2)	**35**	**35**	**35**
	mystery	30	18 (4)	**22**	20 (2)	18 (4)	19 (3)	18 (4)	20 (2)	**22**
	total	155	142 (5)	**147**	139 (8)	129 (18)	142 (5)	143 (4)	145 (2)	143 (4)
	% solved		91.6%	**94.8%**	89.7%	83.2%	91.6%	92.3%	93.5%	92.3%
2	blocks	60	48 (12)	55 (5)	**60**	52 (8)	39 (21)	42 (18)	41 (19)	47 (13)
	miconic	150	**150**	**150**	**150**	**150**	**150**	**150**	**150**	**150**
	freecell	60	**60**	58 (2)	12 (48)	40 (20)	59 (1)	**60**	**60**	**60**
	logistics	198	197 (1)	196 (2)	**198**	178 (20)	**198**	**198**	**198**	**198**
	total	468	455 (4)	**459**	420 (39)	420 (39)	446 (13)	450 (9)	449 (10)	455 (4)
	% solved		97.2%	**98.1%**	89.7%	89.7%	95.3%	96.2%	95.9%	97.2%
3	depots	22	**22**	20 (2)	**22**	**22**	**22**	19 (3)	20 (2)	**22**
	driverlog	20	16 (4)	**20**	**20**	**20**	17 (3)	**20**	**20**	19 (1)
	freecell	20	**20**	**20**	3 (17)	11 (9)	19 (1)	**20**	**20**	**20**
	rovers	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
	satellite	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
	zenotravel	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
	total	122	118 (3)	120 (1)	105 (16)	113 (8)	118 (3)	119 (2)	120 (1)	**121**
	% solved		96.7%	98.4%	86.1%	92.6%	96.7%	97.5%	98.4%	**99.2%**
4	airport	50	30 (16)	38 (8)	45 (1)	**46**	43 (3)	39 (7)	39 (7)	45 (1)
	pipesworld-notankage	50	36 (14)	44 (6)	43 (7)	36 (14)	0 (50)	**50**	48 (2)	44 (6)
	pipesworld-tankage	50	22 (27)	39 (10)	26 (23)	24 (25)	10 (39)	**49**	21 (28)	43 (6)
	promela-optical-telegraph	14	2 (12)	2 (12)	1 (13)	**14**	**14**	13 (1)	13 (1)	6 (8)
	promela-philosophers	29	14 (15)	13 (16)	2 (27)	**29**	**29**	**29**	5 (24)	**29**
	psr-small	50	42 (8)	**50**	48 (2)	**50**	**50**	**50**	47 (3)	**50**
	satellite-strips	36	**36**	34 (2)	**36**	32 (4)	35 (1)	**36**	**36**	**36**
	total	279	182 (84)	220 (46)	201 (65)	231 (35)	181 (85)	**266**	209 (57)	253 (13)
	% solved		65.2%	78.9%	72.0%	82.8%	64.9%	**95.3%**	74.9%	90.7%
5	openstacks	30	7 (23)	**30**	22 (8)	20 (10)	23 (7)	**30**	**30**	**30**
	pathways	30	10 (20)	28 (2)	**30**	**30**	**30**	20 (10)	26 (4)	29 (1)
	pipesworld	50	6 (44)	40 (10)	20 (30)	23 (27)	17 (33)	**50**	22 (28)	43 (7)
	rovers	40	16 (24)	**40**	30 (10)	**40**	30 (10)	**40**	**40**	**40**
	storage	30	18 (12)	19 (11)	**30**	**30**	**30**	25 (5)	21 (9)	18 (12)
	tpp	30	12 (18)	**30**	15 (15)	**30**	20 (10)	**30**	**30**	**30**
	trucks	30	4 (26)	13 (17)	5 (25)	**30**	6 (24)	11 (19)	14 (16)	16 (14)
	total	240	73 (133)	200 (6)	152 (54)	203 (3)	156 (50)	**206**	183 (23)	**206**
	% solved		30.4%	83.3%	63.3%	84.6%	65.0%	**85.8%**	76.2%	**85.8%**
6	cybersec	30	0 (30)	**30**	6 (24)	6 (24)	6 (24)	12 (18)	10 (20)	**30**
	elevators	30	**30**	**30**	25 (5)	**30**	**30**	**30**	**30**	**30**
	openstacks	30	**30**	**30**	**30**	15 (15)	27 (3)	**30**	**30**	**30**
	parcprinter	30	**30**	25 (5)	29 (1)	**30**	**30**	**30**	26 (4)	**30**
	pegsol	30	**30**	29 (1)	11 (19)	**30**	12 (18)	**30**	**30**	**30**
	scanalyzer	30	**30**	**30**	24 (6)	28 (2)	29 (1)	28 (2)	26 (4)	28 (2)
	sokoban	30	27 (2)	26 (3)	0 (29)	6 (23)	8 (21)	24 (5)	25 (4)	**29**
	transport	30	29 (1)	**30**	20 (10)	23 (7)	**30**	**30**	**30**	**30**
	woodworking	30	17 (13)	29 (1)	16 (14)	**30**	**30**	29 (1)	26 (4)	29 (1)
	total	270	223 (43)	259 (7)	161 (105)	198 (68)	202 (64)	243 (23)	233 (33)	**266**
	% solved		82.6%	95.9%	59.6%	73.3%	74.8%	90.0%	86.3%	**98.5%**
total		1534	1193 (251)	1405 (39)	1178 (266)	1294 (150)	1245 (199)	1427 (17)	1339 (105)	**1444**
% solved			77.8%	91.6%	76.8%	84.4%	81.2%	93.0%	87.3%	**94.1%**

Table 1: Number and percentage of solved problems in all sequential domains of the IPCs from 1998 to 2008. Numbers in bold indicate the best results and numbers in parenthesis indicate the number of unsolved problems with respect to the best result.

the running time between YAHSP2 and the three other planners, and confirms that YAHSP2 has much better performances. The detail of the number of solved problems over each IPC and each domain can be found in Table 2.

YAHSP2-MT: A Multi-Threaded Planner

We now briefly describe YAHSP2-MT, a multi-threaded version of YAHSP2 which aims at benefiting from the computing power offered by multi-core processors with shared memory. A more detailed description can be found in (Vidal, Bordeaux, and Hamadi 2010).

The key idea is similar to that of KBFS (Felner, Kraus, and Korf 2003): always expanding first the best node of the open list, giving a maximum trust to the heuristic, may lead search to unpromising parts of the search space; while better parts could have been reached by expanding nodes ranked

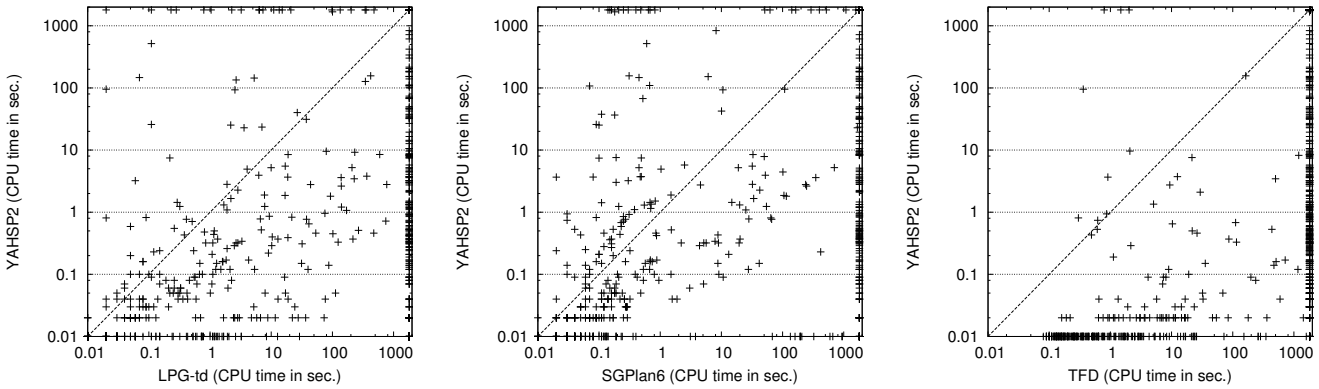

Figure 5: Comparison of the total running time for all temporal planners versus YAHSP2.

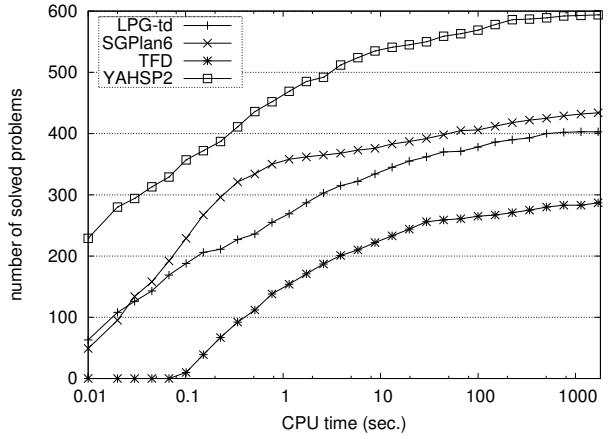

Figure 4: Cumulated number of solved problems for temporal planners.

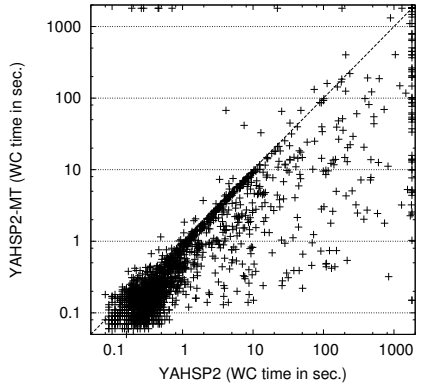

Figure 6: Comparison between the sequential version and the multi-threaded version with restarts of YAHSP2.

lower by the heuristic. KBFS expands the K best nodes of the open list, and then adds all their children to the open list. With the goal of avoiding as much as possible modifications to the existing YAHSP2 code, we simply start K threads that share the same open and closed list, expanding nodes in a concurrent way. This can be made very easily by inserting OpenMP directives between carefully selected lines of code. This simple strategy is used in conjunction with restarts triggered by limits on the number of evaluated nodes, where each restart increases the number of active threads. We also used a slightly different strategy than (Vidal, Bordeaux, and Hamadi 2010): two distinct open and closed lists are each attacked by half of the threads. The first half behave classically, whereas the second half runs an incomplete algorithm, pruning nodes which are obtained with the same number of actions and have the same heuristic value. Figure 6 compares the wall-clock time between YAHSP2 and YAHSP2-MT on a 12-core machine with 24GB of memory and a wall-clock timeout of 30 minutes, on the full set of 2198 problems. The restart strategy starts from 1 thread and goes up to 384 threads (128 for the version submitted to the 7$^{\text{th}}$ IPC). YAHSP2 solves 2038 problems (92.7%), while YAHSP2-MT solves 2082 problems (94.7%). We can see that very often, the multi-threaded version offers super-linear speedups. Furthermore, much less problems are solved faster by the sequential version than in previous tests (Vidal, Bordeaux, and Hamadi 2010), probably because a 4-core machine was used.

Conclusion

We described in this paper the new version of YAHSP, a heuristic search planner that uses a lookahead strategy. Its design has been led by an objective of simplicity, both in the algorithms and the source code, implying many changes with respect to the first version. The resulting planner outperforms state-of-the-art sequential and temporal planners in terms of cumulated number of solved problems and running time. We deliberately avoided analyzing plan quality, as the goal was to produce a fast planner easily embeddable into a wider system such as the DAE_{X} planner. Thus, we expect YAHSP2 to be outperformed by at least $\text{DAE}_{\text{YAHSP}}$ at the 7$^{\text{th}}$ IPC. We also briefly described YAHSP2-MT, the multi-threaded version of YAHSP2 that aims at exploiting multi-core processors, which very often obtains super-linear speedups in comparison with the sequential version.

IPC	domain	#pbs	#solved (difference with best)			
			LPG-td	SGPlan6	TFD	YAHSP2
3	depots	22	**22**	21 (1)	2 (20)	**22**
	driverlog	20	**20**	18 (2)	10 (10)	19 (1)
	rovers	20	**20**	**20**	19 (1)	**20**
	satellite	20	**20**	**20**	**20**	**20**
	zenotravel	20	**20**	**20**	14 (6)	**20**
	total	102	**102**	99 (3)	65 (37)	101 (1)
	% solved		**100.0%**	97.1%	63.7%	99.0%
4	airport	50	42 (3)	43 (2)	10 (35)	**45**
	airport-timewindows	50	0 (46)	0 (46)	6 (40)	**46**
	pipesworld-notankage-deadlines	30	0 (30)	**30**	11 (19)	**30**
	pipesworld-notankage	50	43 (1)	0 (44)	20 (24)	**44**
	pipesworld-tankage	50	28 (15)	10 (33)	6 (37)	**43**
	satellite	36	**36**	35 (1)	7 (29)	**36**
	satellite-timewindows	36	0 (21)	0 (21)	3 (18)	**21**
	total	302	149 (116)	118 (147)	63 (202)	**265**
	% solved		49.3%	39.1%	20.9%	**87.7%**
5	openstacks	20	18 (2)	**20**	4 (16)	**20**
	storage	30	**30**	**30**	8 (22)	19 (11)
	trucks	30	24 (6)	24 (6)	18 (12)	**30**
	total	80	72 (2)	**74**	30 (44)	69 (5)
	% solved		90.0%	**92.5%**	37.5%	86.2%
6	crewplanning	30	11 (19)	**30**	29 (1)	**30**
	elevators	30	0 (30)	**30**	17 (13)	**30**
	openstacks	30	**30**	**30**	**30**	**30**
	parcprinter	30	20 (5)	**25**	13 (12)	18 (7)
	pegsol	30	17 (13)	18 (12)	28 (2)	**30**
	sokoban	30	2 (19)	10 (11)	12 (9)	**21**
	total	180	80 (79)	143 (16)	129 (30)	**159**
	% solved		44.4%	79.4%	71.7%	**88.3%**
total		664	403 (191)	434 (160)	287 (307)	**594**
% solved			60.7%	65.4%	43.2%	**89.5%**

Table 2: Number and percentage of solved problems in all temporal domains of the IPCs from 2002 to 2008. Numbers in bold indicate the best results and numbers in parenthesis indicate the number of unsolved problems with respect to the best result.

Acknowledgments

This work has been supported by the French National Research Agency (ANR) through COSINUS program (project DESCARWIN n°ANR-09-COSI-002). Many thanks to my colleagues of the DAE_{YAHSP} team for their enthusiasm and numerous insightful discussions: Johann Dréo, Pierre Savéant and Marc Schoenauer; as well as to Lucas Bordeaux and Youssef Hamadi for their multi-core expertise.

References

Bäckström, C. 1998. Computational aspects of reordering plans. *JAIR* 9:99–137.

Baier, J. A., and Botea, A. 2009. Improving planning performance using low-conflict relaxed plans. In *Proc. ICAPS*, 10–17.

Bibai, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *Proc. ICAPS*, 18–25.

Chen, Y.; Wah, B. W.; and Hsu, C.-W. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *JAIR* 26:323–369.

Dréo, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2011. Divide-and-evolve: the marriage of descartes and darwin. In *Booklet of the 7th IPC*.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proc. ICAPS*, 130–137.

Felner, A.; Kraus, S.; and Korf, R. E. 2003. KBFS: K-best-first search. *AMAI* 39(1-2):19–39.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs in LPG. *JAIR* 20:239–290.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Lipovetzky, N., and Geffner, H. 2011. Searching for plans with carefully designed probes. In *Proc. ICAPS*.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS*, 273–280.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:127–177.

Rintanen, J. 2010. Heuristic planning with sat: Beyond uninformed depth-first search. In *Proc. Australasian Conf. on AI*, 415–424.

Teichteil-Königsbuch, F.; Kuter, U.; and Infantes, G. 2010. Incremental plan aggregation for generating policies in MDPs. In *Proc. AAMAS*, 1231–1238.

Vidal, V.; Bordeaux, L.; and Hamadi, Y. 2010. Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In *Proc. SOCS*, 100–107.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proc. ICAPS*, 150–160.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: A baseline for probabilistic planning. In *Proc. ICAPS*, 352–359.

BJOLP: The Big Joint Optimal Landmarks Planner*

Carmel Domshlak
Technion

Malte Helmert
Albert-Ludwigs-Universität Freiburg

Erez Karpas
Technion

Emil Keyder
Universitat Pompeu Fabra

Silvia Richter
NICTA

Gabriele Röger
Albert-Ludwigs-Universität Freiburg

Jendrik Seipp
Albert-Ludwigs-Universität Freiburg

Matthias Westphal
Albert-Ludwigs-Universität Freiburg

Abstract

BJOLP, The Big Joint Optimal Landmarks Planner uses landmarks to derive an admissible heuristic, which is then used to guide a search for a cost-optimal plan. In this paper we review landmarks and describe how they can be used to derive an admissible heuristic. We conclude with presenting the BJOLP planner.

Introduction

Landmarks for deterministic planning are (possibly logically compound) facts that must take place at some point in every plan for a given planning task (Porteous, Sebastia, and Hoffmann 2001; Porteous and Cresswell 2002; Hoffmann, Porteous, and Sebastia 2004). For example, if a goal in a Blocksworld task is to have block *A* stacked on block *B*, and initially this does not hold, then *clear(B)* must hold at some point for the goal to be achieved, and thus it is a landmark for that task. Goals are trivially landmarks, and thus *on(A,B)* is a landmark as well. We can also infer that *clear(B)* must be achieved before stacking *A* on *B*, establishing an ordering between these two landmarks.

The two issues with planning landmarks are how to discover them, and how to exploit them. Even for propositional landmarks only, sound and complete discovery of all such landmarks is known to be PSPACE-complete (Porteous, Sebastia, and Hoffmann 2001). Still, many landmarks can often be efficiently discovered (Hoffmann, Porteous, and Sebastia 2004; Richter and Westphal 2010; Keyder, Richter, and Helmert 2010).

Once discovered, landmarks can be extremely helpful in guiding the search for a plan, as evidenced by the performance of the LAMA planner (Richter and Westphal 2010) in IPC-2008. LAMA uses landmarks to derive a (inadmissible) pseudo-heuristic, used to guide a satisficing heuristic search.

In this paper,we describe a method for deriving *admissible estimates from a set of planning landmarks*, with its instances varying from easy to compute, to, in some sense, optimally accurate. The resulting heuristics are what we call *multi-path dependent*. We also describe a simple best-first search that exploits such heuristics, and finds optimal solutions more efficiently than standard A^*.

*This paper is strongly based upon Karpas and Domshlak (2009) and Keyder, Richter and Helmert (2010).

Notation and Background

We consider planning in the SAS$^+$ formalism (Bäckström and Nebel 1995); a SAS$^+$ description of a planning task can be automatically generated from its PDDL description (Helmert 2009). A SAS$^+$ task is given by a 4-tuple $\Pi = \langle V, A, s_0, G\rangle$. $V = \{v_1, \ldots, v_n\}$ is a set of *state variables*, each associated with a finite domain $dom(v_i)$, where (assuming name uniqueness) the union of the variable domains $F = \bigcup_i dom(v_i)$ is the set of *facts*. Each complete assignment s to V is called a *state*; s_0 is an *initial state*, and the *goal* G is a partial assignment to V. A is a finite set of *actions*, where each action a is a pair $\langle \mathsf{pre}(a), \mathsf{eff}(a)\rangle$ of partial assignments to V called *preconditions* and *effects*, respectively. Each action $a \in A$ has a non-negative cost $\mathcal{C}(a)$.

An action a is applicable in a state s iff $\mathsf{pre}(a) \subseteq s$. Applying a changes the value of each state variable v to $\mathsf{eff}(a)[v]$ if $\mathsf{eff}(a)[v]$ is specified. The resulting state is denoted by $s[\![a]\!]$; by $s[\![\langle a_1, \ldots, a_k\rangle]\!]$ we denote the state obtained from sequential application of the (respectively applicable) actions $a_1, \ldots, a_k$ starting at state s. Such an action sequence is a plan if $G \subseteq s_0[\![\langle a_1, \ldots, a_k\rangle]\!]$, and the cost of the plan is $\Sigma_{i=1}^{k}\mathcal{C}(a_i)$. In cost-optimal planning, we are interested in finding a plan with a minimal cost.

Let $\Pi = \langle V, A, s_0, G\rangle$ be a planning task, ϕ be a propositional logic formula over facts F, $\pi = \langle a_1, \ldots, a_k\rangle$ be an action sequence applicable in s_0, and $0 \leq i \leq k$. Following the terminology of Hoffmann *et al.*, we say that ϕ is *true at time* i in π iff $s_0[\![\langle a_1, \ldots, a_i\rangle]\!] \models \phi$, ϕ is *first added at time* i in π iff ϕ is true in π at time i, but not at any time $j < i$, and ϕ is a *landmark* of Π iff in each plan for Π, it is true at some time.

In addition to knowing landmarks, it is also useful to know in which order they should be achieved on the way to the goal. Hoffmann *et al.* define different types of potentially useful orderings. In particular, landmark ϕ is said to be *greedy-necessarily ordered* before landmark ψ iff, for each action sequence applicable in s_0, if ψ is first added in π at time i, then ϕ is true in π at time $i - 1$.

Porteous *et al.* show that deciding if just a single fact is a landmark, as well as deciding an ordering between two fact landmarks, are PSPACE-complete problems. There-

fore, practical methods for finding landmarks are either incomplete or unsound. In what follows we assume access to a sound such procedure; in particular, we combine the landmarks from the RHW landmark generation method (Richter and Westphal 2010) and the h^m landmark generation method (Keyder, Richter, and Helmert 2010).

In what follows we assume that a planning task Π is simply given to us with a landmark structure $\langle L, Ord\rangle$, where L is a set of Π's landmarks, and Ord is a set of typed orderings over L, containing, in particular, the greedy-necessary ordering over L.

Admissible Landmark Heuristics

Deriving heuristic estimates from landmarks has been proposed by Richter et al. (2010) who estimate the goal distance of a state s, reached via a sequence of actions π from the initial state, by the *number of landmarks $L(s,\pi)$ yet to be achieved from s onwards*. Specifically, if the search starts with the landmark structure $\langle L, Ord\rangle$, then

$$L(s,\pi) = (L \setminus \mathsf{Accepted}(s,\pi)) \cup \mathsf{ReqAgain}(s,\pi) \quad (1)$$

where $\mathsf{Accepted}(s,\pi) \subseteq L$ and $\mathsf{ReqAgain}(s,\pi) \subseteq \mathsf{Accepted}(s,\pi)$ are the sets of *accepted* and *required again* landmarks, respectively. A landmark is accepted if it is made true at some time along π. An accepted landmark is required again if (i) it does not hold in s, and (ii) it is ordered greedy-necessarily before some landmark which is not accepted, or is a goal.

The estimate $|L(s,\pi)|$ is not a proper heuristic in the usual sense, but rather *path-dependent*; it is a function of both an evaluated state s, and a path from s_0 to s. However, $|L(s,\pi)|$ can still be used like a state-dependent heuristic in best-first search. In particular, combined with some other helpful techniques, it has been successfully used by the LAMA planner at the Sequential Satisficing Track of the IPC-2008 competition.

Action Cost Sharing by Landmarks

It is not hard to verify that the estimate $|L(s,\pi)|$ is not admissible. For instance, in a Blocksworld task, let $L(s,\pi) = \{$*crane-empty*, *on(A,B)*$\}$. While $|L(s,\pi)| = 2$, it is possible a single action *stack(A,B)* reaches the goal from s. However, below we show that the gap between the estimate $|L(s,\pi)|$ and admissibility is not that hard to close.

Considering the landmarks through the actions that can possibly achieve them, let $cost(\phi)$ be a cost assigned to each landmark ϕ, and $cost(a,\phi)$ be a cost "assigned" by the action a to ϕ. Suppose also that these (all non-negative) costs satisfy

$$\begin{aligned} \forall a \in A: \quad & \sum_{\phi \in L(a|s,\pi)} cost(a,\phi) \le \mathcal{C}(a) \\ \forall \phi \in L(s,\pi): \quad & cost(\phi) \le \min_{a \in \mathsf{ach}(\phi|s,\pi)} cost(a,\phi) \end{aligned} \quad (2)$$

where each action subset $\mathsf{ach}(\phi|s,\pi) \subseteq A$ (in particular) contains all the actions that can possibly be used to directly achieve landmark ϕ along a goal-achieving suffix of π, and, reversely, $L(a|s,\pi) = \{\phi \mid \phi \in L(s,\pi), a \in \mathsf{ach}(\phi|s,\pi)\}$. Informally, Eq. 2 enforces partitioning of each action cost among the landmarks this action can possibly establish, and verifies that the cost of each landmark ϕ is no greater than the minimum cost assigned to ϕ by its possible achievers.

In our planner, we use the (initial-state dependent and efficiently computable) set of "possible", and its subset of "first-time possible", achievers of ϕ (Porteous and Cresswell 2002) to estimate the achievers of simple or disjunctive landmarks (the achievers of a disjunctive landmark ϕ can be simply estimated by the set of all actions achieving some element of ϕ). The possible achievers of a conjunctive landmark ϕ are estimated as the actions which achieve (at least) one of the conjuncts, without deleting any of the other conjuncts, and the first-time possible achievers of a conjunctive landmark ϕ are estimated as the subset of the possible achievers, which do not have ϕ as a landmark for achieving their preconditions.

If $\phi \notin \mathsf{Accepted}(s,\pi)$, then we set $\mathsf{ach}(\phi|s,\pi)$ to the first-time possible achievers of ϕ, and otherwise to the possible achievers of ϕ. In any event, action cost sharing is all we need to derive from $L(s,\pi)$ an admissible estimate of the goal distance.

Proposition 1 *Given a set of action-to-landmark and landmark costs satisfying Eq. 2, $h_L(s,\pi) = cost(L(s,\pi)) = \sum_{\phi \in L(s,\pi)} cost(\phi)$ is an admissible estimate of the goal distance $h^*(s)$.*

Proof sketch: Let $P = \langle a_1, a_2, \ldots, a_r\rangle$ be any plan from s to the goal. Let $lm(a) = \{\phi | \phi \in \mathsf{eff}(a) \cap L(s,\pi)\}$ be the set of landmarks that are achieved by action a. Then $\bigcup_{i=1}^r lm(a_i)$ is the set of landmarks that are achieved by P. By the definition of landmarks, P must achieve all the landmarks in $L(s,\pi)$, and therefore $L(s,\pi) \subseteq \bigcup_{i=1}^r lm(a_i)$, and $cost(L(s,\pi)) \le cost(\bigcup_{i=1}^r lm(a_i))$. It is easy to see that $cost(\bigcup_{i=1}^r lm(a_i)) \le \sum_{i=1}^r cost(lm(a_i))$, because some landmarks could potentially be counted twice in the right-hand side expression (i.e. achieved by two or more actions). From the requirements on landmark costs we have that $cost(lm(a_i)) \le \mathcal{C}(a_i)$, and therefore $\sum_{i=1}^r cost(lm(a_i)) \le \sum_{i=1}^r \mathcal{C}(a_i)$. If we combine all of this we get $h_L(s,\pi) = cost(L(s,\pi)) \le cost(\bigcup_{i=1}^r lm(a_i)) \le \sum_{i=1}^r cost(lm(a_i)) \le \sum_{i=1}^r \mathcal{C}(a_i) = \mathcal{C}(P)$ ∎

Proposition 1 leaves the choice of the actual action-cost partitioning open. The most straightforward choice here is probably *uniform cost sharing* in which each action partitions its costs equally among all the landmarks it can possibly achieve, that is, $cost(a,\phi) = \mathcal{C}(a)/|L(a|s,\pi)|$. The advantage of such a uniform cost sharing is the efficiency of its computation. However, the induced action-cost partition can be sub-optimal. For instance, consider a planning task with a landmark set $\{p_1, \ldots, p_k, q\}$ such that the only possible achiever of each p_i is a unit-cost action a_i with $\mathsf{eff}(a_i) = \{p_i, q\}$. For $1 \le i \le k$, the uniform cost sharing assigns here $cost(a_i,p_i) = cost(a_i,q) = 0.5$, which gives $cost(p_i) = cost(q) = 0.5$, and thus $h_L(s,\pi) = k/2 + 0.5$. In contrast, the optimal cost sharing would assign, for all

$1 \leq i \leq k$, $cost(a_i, p_i) = 1$ and $cost(a_i, q) = 0$, implying $cost(p_i) = 1$, $cost(q) = 0$, and thus $h_L(s, \pi) = k$.

The good news, however, is that such an optimal cost sharing can be computed in poly-time by compiling Eq. 2 into strictly linear constraints, and solving the linear program induced by these constraints and the objective $\max \sum_{\phi \in L(s,\pi)} cost(\phi)$. In addition, this cost sharing scheme alleviates an annoying shortcoming of ad hoc (e.g., uniform) cost sharing schemes, and satisfies *monotonicity along the inclusion relation of the landmark sets* $L(s, \pi)$. It is not hard to verify that, for any two sets of landmarks L and L' such that $L \subseteq L'$, the LP-based cost sharing ensures $cost(L') \geq cost(L)$ by the very virtue of being optimal, and thus yields at least as informative heuristic estimate with L' as with L. This property is appealing as it allows separating landmark discovery and landmark exploitation without any loss of accuracy, leaving the phase of discovery with the simple principle of "more landmarks can never hurt". In contrast, the simple yet ad hoc uniform cost sharing cannot guarantee such monotonicity. For instance, the uniform cost sharing in the example above but *without* landmark q yields $h_L(s, \pi) = k$, while with q it results in $h_L(s, \pi) = k/2 + 0.5$.

Action Landmarks

The LP-based "admissibilization" of the landmark sets $L(s, \pi)$ is optimal, but this, of course, only when the landmark costs are estimated with respect to solely Eq. 2. Any additional information about landmarks may help improving the accuracy of the estimate. One type of such information corresponds to *action landmarks* (Zhu and Givan 2004; Vidal and Geffner 2006). Similarly to landmarks over facts, an action a is an action landmark of a planning task Π iff it is taken along every plan for Π.

Although it is possible to discover action landmarks in a pre-processing phase (a sufficient and efficiently testable condition for a being an action landmark is that a relaxed planning task without a is not solvable), The BJOLP planner discovers action landmarks dynamically during uniform cost-partitioning as follows: whenever $|\mathsf{ach}(\phi|s, \pi)| = 1$ (that is, there is only one achiever of ϕ), then that single achiever, which we denote a, is an action landmark. Since a must be used to achieve ϕ, it makes no sense to divide its cost between other landmarks it might possibly achieve. Therefore we assign the full cost of a to ϕ (that is, $cost(a, \phi) = \mathcal{C}(a)$), and assign 0 cost from a to its other effects (that is, $cost(a, \phi') = 0$ for $\phi \neq \phi' \in L(a|s, \pi)$). This allows us to improve upon "naive" uniform cost-partitioning. We call the heuristic resulting from the use of action landmarks h_{LA}. Clearly h_{LA} is still admissible.

We remark that this dynamic action landmark discovery was not implemented originally in Karpas and Domshlak (2009), but was added later in Keyder, Richter, and Helmert (2010).

From Path to Multi-Path Dependence

Let us now return to the definition of the path-dependent set $L(s, \pi)$ in Eq. 1. Both LAMA's heuristic $|L(s, \pi)|$, and the admissible heuristics h_L and h_{LA}, exploit information provided by the path π to better estimate the goal distance from s. Suppose now that we are given a set of paths from s_0 to s; such a set of paths can in particular be discovered anyway by any systematic, forward-search procedure. Proposition 2 shows that such a set of paths can be much more informative than any of its individual components.

Proposition 2 *Let Π be a planning task with a landmark set L, s be a state of Π, $\mathcal{P}$ be a set of paths from s_0 to s, and π_g be a goal achieving path from s. Then for each path $\pi \in \mathcal{P}$, π_g achieves all landmarks in $L \setminus \mathsf{Accepted}(s, \pi)$.*

The proof is straightforward: Assume a landmark ϕ is achieved by a path $\pi \in \mathcal{P}$ but not by a path $\pi' \in \mathcal{P}$. The latter implies that all the extensions of π' should still achieve ϕ, and the extensions of π' are exactly the extensions of π.

Proposition 2 immediately leads to *multi-path dependent* versions of h_L and h_{LA}. Given a set of landmarks L, and a set of paths $\mathcal{P}$ from s_0 to s, let

$$L(s, \mathcal{P}) = (L \setminus (\mathsf{Accepted}(s, \mathcal{P})) \cup \mathsf{ReqAgain}(s, \mathcal{P}) \quad (3)$$

where $\mathsf{Accepted}(s, \mathcal{P}) = \bigcap_{\pi \in \mathcal{P}} \mathsf{Accepted}(s, \pi)$, and $\mathsf{ReqAgain}(s, \mathcal{P}) \subseteq \mathsf{Accepted}(s, \mathcal{P})$ is specified as before by s and the greedy-necessary orderings over L. Given that, the multi-path dependent versions of h_L and h_{LA} straightforwardly reflect their path-dependent counterparts, by replacing $L(s, \pi)$ with $L(s, \mathcal{P})$.

The improvement in accuracy in switching to multi-path landmark heuristics can be substantial. For instance, if we have access to two paths to s, each suggesting that half of the landmarks have been achieved, yet they entirely disagree on the identity of the achieved landmarks, then the estimate of the (still admissible) multi-path dependent heuristic might be twice as high as this of the path-dependent heuristic.

Finally, utilizing multi-path dependent estimates for optimal search requires adapting the standard A^* search procedure. In fact, a slight adaptation of A^* is desirable even in case of such path-dependent heuristics. Designed for state-dependent estimates, A^* computes $h(s)$ for each state s only when s is first generated. This will still guarantee optimality with path-dependent estimates as well, yet, if π and π' are the current path and a newly discovered path from s_0 to s, respectively, then we may have $h(s, \pi') > h(s, \pi)$. That is, a newly discovered path may better inform us about the goal distance from s. We can slightly modify A^* to compute the heuristic value each time a new path to a state is discovered, and utilize the highest estimate discovered so far. This, of course, preserves search admissibility, and potentially reduces the number of expanded nodes. Note that this does not contradict "optimal efficiency" of the basic A^* as the latter holds only for monotonic, state-dependent heuristics (Dechter and Pearl 1985).

The modification of A^* for multi-path dependent heuristics is very much similar in spirit. Each time a new path to state s is discovered, it is stored in the list of such paths $\mathcal{P}(s)$, and s's heuristic value is marked as "dirty". Of course, storing all paths to s is generally infeasible, and the algorithm is usable only in cases where the relevant information carried by $\mathcal{P}(s)$ can be captured and stored compactly.

In fact, the adaptation of A^* to path-dependent heuristics as above constitutes such a special case of all the relevant information of a set of paths being the maximal value of the heuristic estimates induced by them individually. Nicely, the multi-path dependent landmark heuristics h_L and h_{LA} also constitute a usable special case as above. In our variant of A^*, referred later as *LM-A**, we associate each state s with the landmark set $L(s, \mathcal{P}(s))$ as in Eq. 3. When a new path π to s is discovered (and extends $\mathcal{P}(s)$), the landmarks are incrementally updated to $L(s, \mathcal{P}(s) \cup \{\pi\})$ by exploiting the monotonicity of the intersection set operator.

When a state s is removed from the open list for expansion, before actually performing the expansion, *LM-A** first checks whether the s's heuristic is marked as "dirty" (which happens when new paths to s have been discovered between the time s was inserted into the open list, and the time it is remove from the open list). If s's h-value is dirty, we reevaluate $h(s)$ (using the new information), and if the new heuristic value is higher than the previous heuristic value, we reinsert s into the open list with the new h-value. Note that both the old and new h-values are admissible, and so if the new h-value is lower (which could happen when using uniform cost-partitioning), admissibility is maintained. If the s is not "dirty", or if the newly computed h-value is not higher than the old value, then s is expanded as usual. *LM-A** is described in pseudo-code in Figure 1.

Implementation

We have implemented h_L, h_{LA} and *LM-A** on top of the Fast Downward planning system. The BJOLP planner uses *LM-A** with the h_{LA} heuristic (using uniform cost-partitioning and the new dynamic action-landmark discovery).

As mentioned before, the landmarks we use for BJOLP are obtained by combining the landmarks discovered by the RHW method (Richter and Westphal 2010) and the h^m landmarks (Keyder, Richter, and Helmert 2010) with $m = 1$. First the entire landmarks graph is generated by each of these discovery methods. Then, the landmarks and orderings are merged, and dominated (in the sense of logical implication) landmarks are discarded. For example, if one method discovers landmark ϕ and the other discovers landmark $\phi \vee \phi'$, then $\phi \vee \phi'$ will be discarded (along with all its orderings). Dominated orderings are also eliminated by logical implication (remember that every greedy-necessary ordering is a natural ordering, but not vice versa).

Finally, this version of BJOLP uses a much more efficient implementation of the way landmark information for each state is stored. While previous versions used a set (of the C++ standard template libraries) to store, for each state, the set of accepted landmarks, BJOLP uses a boolean vector, which uses one bit per landmark. This speeds BJOLP up considerably over previous versions and dramatically reduces its memory footprint.

LM-A*

1. Put the start node s on a list called *OPEN* of unexpanded nodes. Assign g(s) = 0.
2. If *OPEN* is empty, exit with failure; no solution exists.
3. Remove from *OPEN* a node n at which f=g+h is minimum. Break ties in favor of low h (although ties can be broken arbitrarily, as long as goal nodes are favored).
4. If n is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from n to s (pointers are assigned in steps 7 and 8).
5. If n is marked as dirty, calculate $h(n)$. Else, goto step 7.
6. Compare the newly computed $h(n)$ with that previously assigned to n. If the new value is greater, substitute it for the old, update $f(n)$, move n back to *OPEN*, and goto step 2.
7. Place n on a list called *CLOSED* to be used for expanded nodes, and expand node n, generating all its successors with pointers back to n.
8. For every successor n' of n:
   (a) Store the current path to n' (through n).
   (b) Calculate $g(n')$ = $g(n)$ + $\mathcal{C}(a)$, where a is the action leading from n to n'.
   (c) Calculate $h(n')$.
   (d) If n' is neither in *OPEN* nor in *CLOSED*, then add it to *OPEN*. Assign the newly computed $g(n')$ and $h(n')$ to node n'.
   (e) If n' already resides in *OPEN* or *CLOSED*:
      i. Store the new path to n' and mark n' as dirty.
      ii. Compare the newly computed $g(n')$ with that previously assigned to n'. If the new value is lower, substitute it for the old (n' now points back to n instead of to its predecessor), and update $f(n')$. Move the matching node n' back to *OPEN* if it resided in *CLOSED*.
9. Go to step 2.

Figure 1: Pseudo-code of *LM-A**

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Comp. Intell.* 11(4):625–655.

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *J. ACM* 32(3):505–536.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. 173:503–535.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. 22:215–278.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *IJCAI*, 1728–1733.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In *ECAI*, 335–340.

Porteous, J., and Cresswell, S. 2002. Extending landmarks

analysis to reason about resources and repetition. In *PLANSIG*.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *ECP*.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. 170(3):298–335.

Zhu, L., and Givan, R. 2004. Heuristic planning via roadmap deduction. In *IPC-4*, 64–66.

Improving Cost-Optimal Domain-Independent Symbolic Planning

Peter Kissmann and **Stefan Edelkamp**
TZI Universität Bremen, Germany
{edelkamp,kissmann}@tzi.de

Abstract

Symbolic search with BDDs has shown remarkable performance for cost-optimal deterministic planning by exploiting a succinct representation and exploration of state sets. In this paper we enhance BDD-based planning by applying a combination of domain-independent search techniques: the optimization of the variable ordering in the BDD by approximating the linear arrangement problem, pattern selection for improved construction of search heuristics in form of symbolic partial pattern databases, and a decision procedure for the amount of bidirection in the symbolic search process.

Introduction

As documented in the international planning competition IPC 2008 symbolic search with Binary Decision Diagrams (Bryant 1986), BDDs for short, has shown considerable success for *cost-optimal* planning and for planning with soft goals for maximizing the *net-benefit*, which is defined as the payoff for satisfying goal preferences minus the total cost of the actions in the plan, so that our planner GAMER (Edelkamp and Kissmann 2009) won the corresponding optimization tracks.

While *explicit-state search* is concerned with the expansion of single states and the calculation of successors of a single state, in *symbolic search* (McMillan 1993) with BDDs sets of states are handled. E. g., the assignments satisfying a Boolean formula can be seen as sets of states. Similarly, we can represent any state vector (in binary representation) as a Boolean formula with one satisfying assignment. To achieve this, we represent any state as a conjunction of (binary) variables. Thus, a set of states is the disjunction of such a conjunction of variables, so that we can easily represent it in form of a BDD.

This paper contains three contributions to improve our existing symbolic search technology to come up with a new version of GAMER that clearly outperforms the 2008 version.

1. The variable ordering has a significant effect on the size of the BDD representation for many Boolean functions (Bryant 1986), but the automated inference of a good ordering has not been studied for symbolic planning. As the problem of optimizing the ordering in a BDD is hard in general and existing reordering techniques show performance drawbacks in practice, this paper follows an alternative approach based on exploiting causal dependencies of state variables.
2. The BDD exploration on state sets can be improved by including heuristics into the search process. Based on projecting away all but a selected *pattern* of state variables, in backward search symbolic versions of pattern databases (Culberson and Schaeffer 1998) can be constructed. In this paper we provide an alternative realization for greedy pattern selection based on constructing symbolic pattern databases.
3. While symbolic backward search is conceptually simple and often effective, in some domains the sizes of the BDDs increase too quickly to be effective for a bidirectional exploration of the original state space due to a large number of invalid states. For this case we integrate a simple decision procedure to allow backward search in abstract state space search only.

This paper is structured as follows. First, we introduce deterministic planning with finite domain variables and the causal dependencies among them. Next, we turn to the linear arrangement problem that we greedily optimize to find a good BDD variable ordering. Then, we present symbolic search strategies including symbolic BFS, symbolic single-source shortest-path and symbolic A* as needed for step- and cost-optimal planning. We continue with a new automated construction of search heuristics in form of symbolic pattern databases. Then, we consider a decision procedure of whether or not to start backward search. Finally, we evaluate the new version of GAMER on the full set of problems from IPC 2008.

Planning and Causal Graphs

In (deterministic) planning we are confronted with a description of a planning problem and are interested in finding a good solution (a good plan) for this problem. Such a plan is a set of actions whose application transforms the initial state to one of the goal states. In *propositional* planning states are sets of Boolean variables. However, propositional encodings are not necessarily the best state space representations for solving propositional planning problems. A multi-valued

variable encoding is often better. It transforms a propositional planning task into an *SAS$^+$ planning* task.

Definition 1 (Cost-Based SAS$^+$ Planning Task)**.** *A cost-based SAS$^+$* planning task $\mathcal{P} = \langle \mathcal{S}, \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{T}, c \rangle$ *consists of a set $\mathcal{V} = \{v_1, \ldots, v_n\}$ of state variables for states in $\mathcal{S}$, where each $v \in \mathcal{V}$ has finite domain D_v. Some sets of states are* partial assignments*, i. e., functions s over $\mathcal{V}$, such that $s(v) \in D_v$, if $s(v)$ is defined. The initial state $\mathcal{I}$ is a* full assignment*, i. e., a total function over $\mathcal{V}$. The goal states $\mathcal{T}$ are specified in form of a* partial assignment*. An action $a \in \mathcal{A}$ is given by a pair $\langle pre, \mathit{eff} \rangle$ of preconditions and effects, where pre and eff are partial assignments. The cost function $c : \mathcal{A} \mapsto \mathbb{N}_0^+$ specifies the cost of each action.*

Definition 2 (Plan and Cost of a Plan)**.** *A* plan P *is a sequence of actions $(A_1, A_2, \ldots, A_m) \in \mathcal{A}^m$ with $A_m(A_{m-1}(\ldots A_1(\mathcal{I})\ldots)) \in \mathcal{T}$. The* total cost $\mathcal{C}(P)$ *of a plan P is the sum of the costs of all actions within P, i. e., $\mathcal{C}(P) := c(A_1) + c(A_2) + \ldots + c(A_m)$.*

Definition 3 (Optimal Plan)**.** *A plan P is called* optimal*, if there is no plan P' with $\mathcal{C}(P') < \mathcal{C}(P)$.*

The following definition specifies the dependencies among the variables.

Definition 4 (Causal Graph)**.** *The* causal graph *of an SAS$^+$ planning task $\mathcal{P}$ with variable set $\mathcal{V}$ is a directed graph (V, E) with $V = \mathcal{V}$ and $(u, v) \in E$ if and only if $u \neq v$ and there is an action $\langle pre, \mathit{eff} \rangle \in \mathcal{A}$, such that $\mathit{eff}(v)$ is defined and either $pre(u)$ or $\mathit{eff}(u)$ is defined. This implies that an edge is drawn from one variable to another if the change of the second variable is dependent on the current assignment of the first variable.*

To arrive at a symmetrical relation, the dependencies in the causal graph are applied in both directions.

Optimal Linear Variable Arrangement

In short, BDDs are memory-efficient data structures used to represent Boolean functions as well as to perform set-based search. A BDD is a directed acyclic graph with one root and two terminal nodes, the 0- and the 1-sink. Each internal node corresponds to a binary variable and has two successors, one representing that the current variable is *false* and the other representing that it is *true*. The assignment of the variables derived from any path from the root to the 1-sink corresponds to an assignment for which the represented function evaluates to *true*.

The variable ordering problem in a BDD is co-NP-complete (Bryant 1986), so that optimal algorithms like the one by Friedman and Supowit (1990) are practically infeasible even for small-sized planning problems. Dynamic reordering algorithms as provided in current BDD packages require sifting operations on existing BDDs and are often too slow to be effective in planning. One reason is that they do not exploit any knowledge on variable dependencies.

Thus, we decided to approximate another optimization problem to find a good variable ordering without BDDs.

Definition 5 (Optimal Linear Arrangement)**.** *Given a weighted graph $G = (V, E, c)$ on n vertices, in the* optimal linear arrangement *problem the goal is to find a permutation $\pi : V \to \{1, \ldots, n\}$ that minimizes $\sum_{e=(u,v)\in E} c(e) \cdot |\pi(v) - \pi(u)|$.*

We set the weights $c(e)$ to 1, so that we actually solve the version called *simple optimal linear arrangement*. This problem is (still) NP-hard (via a reduction to Max-Cut and Max-2-Sat (Garey, Johnson, and Stockmeyer 1976)). It is also hard to provide approximations within any constant factor (Devanur et al. 2006).

There are different known generalizations to the optimal linear arrangement problem. The *quadratic assignment* problem $\sum_{e=(u,v)\in E} c(e) \cdot c'(\pi(u), \pi(v))$ for some weight c' is a generalization, which includes the traveling salesman problem TSP as one case (Lawler 1963). Here, we consider the optimization function

$$\Phi(\pi) = \sum_{(u,v)\in E \vee (v,u)\in E} d(\pi(u), \pi(v)),$$

subject to the metric $d(x, y) = ||x-y||_1 = \sum_{i=1}^{n} |x_i - y_i|$ or $d(x, y) = ||x-y||_2^2 = \sum_{i=1}^{n} (x_i - y_i)^2$. For our experiments we chose to use the latter.

In our case G is the causal graph, i. e., the nodes in V are the multi-valued variables in the SAS$^+$ encoding and the edges in E reflect the causal dependencies.

As the problem is complex we apply a greedy search procedure for optimization with two loops. The outer loop calculates a fixed number ρ of random permutation, while the inner loop performs a fixed number η of transpositions. To increase the values of ρ and η we decided to incrementally compute $\Phi(\pi) = \sum_{(u,v)\in E} d(\pi(u), \pi(v))$ as follows. Let $\tau(i, j)$ be the transposition applied to the permutation π to obtain the new permutation π', and let x and y be the associated variables to the indices i and j in π. We have

$$\begin{aligned}
\Phi(\pi') &= \sum_{(u,v)\in E \vee (v,u)\in E} d(\pi'(u), \pi'(v)) \\
&= \Phi(\pi) - \sum_{(w,x)\in E \,\vee\, (x,w)\in E} d(\pi(w), i) \\
&\quad - \sum_{(w,y)\in E \,\vee\, (y,w)\in E} d(\pi(w), j) \\
&\quad + \sum_{(w,x)\in E \,\vee\, (x,w)\in E} d(\pi'(w), j) \\
&\quad + \sum_{(w,y)\in E \,\vee\, (y,w)\in E} d(\pi'(w), i).
\end{aligned}$$

The complexity for computing $\Phi(\pi')$ reduces from quadratic to linear time for the incremental computation. This has a considerable impact on the performance of the optimization process as it performs millions of updates in a matter of seconds instead of minutes (depending on the causal graph and the domain chosen).

Now that the variable ordering has been fixed we consider how to optimally solve deterministic planning problems.

Step-Optimal Symbolic Planning

To calculate the set of successors of a given set of states, the *image* operator is used. Though we are able to find efficient variable orderings for many problems, we cannot expect to be able to calculate exponential search spaces in polynomial time. This comes from the fact that the calculation of the *image* is NP-complete (McMillan 1993). It is, however, not essential to compute one image for all actions in common. Instead, we apply the *image* operator to one action after the other and calculate the union of these afterwards.

Using the *image* operator the implementation for a symbolic breadth-first search (BFS) is straight-forward. All we need to do is to apply *image* first to the initial state and afterwards to the last generated successor states. The search ends when a fix-point is reached. For this we must remove the duplicate states. If we store the set of all reachable states as one BDD, duplicate elimination is done implicitly by the BDD, but if we store each BFS layer as a BDD, we must remove the states of the previous layers. In the first case, the fix-point is reached if the call of the *image* operator does not increase the number of states stored in the BDD, in the second case it is reached if the result of the application of the *image* operator along with the removal of duplicate states results in a BDD representing the empty set.

For the search in backward direction we use the *pre-image* operator, which is similar to *image* but calculates all predecessors of a given set of states.

Given the *image* and *pre-image* operators we also devise a symbolic bidirectional breadth-first search. For this we start one search in forward direction (using *image*) at the initial state and another in backward direction (using *pre-image*) at the goal states. Once the two searches overlap we stop and generate an optimal solution.

Cost-Optimal Symbolic Sequential Planning

Handling action costs is somewhat more complicated. Now, an optimal plan is no longer one of minimal length but rather one with minimal total cost, i. e., the sum of the costs of all actions within the plan needs to be minimal.

Dijkstra's single-source shortest-paths search (1959) is a classical algorithm used for state spaces with edge weights. In its normal form it consists of a list of nodes denoted with their distance to the start node. Initially, the distance for the start node is 0 and that for the other nodes ∞.

It chooses the node u with minimal distance, removes it from the list and updates the distances of the neighboring nodes. For each neighbor v it recalculates its distance to the start node to $\min(d(v), d(u) + c(u,v))$, with $c(u,v)$ being the cost of edge (u,v) and $d(u)$ the distance of node u to the start node calculated so far.

For the symbolic version of this algorithm we need a way to access the states having a certain distance from $\mathcal{I}$. For this typically a priority queue is used. As we are concerned only with discrete action costs we can partition the priority queue into buckets, resulting in an *open* list (Dial 1969). In this list, we store all the states that have been reached so far in the bucket representing the distance from $\mathcal{I}$ that they have been found in.

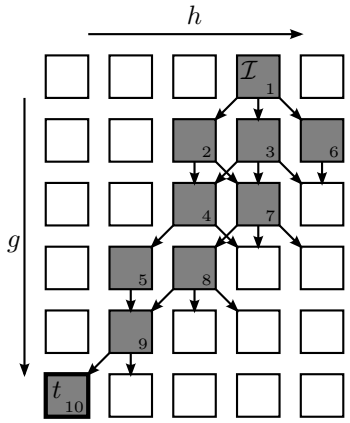

Figure 1: The working of BDDA*. The buckets that are expanded in the course of the search are shaded in gray, the numbers in them denote the order of their expansion and the arrows point to the buckets containing successors of the current one.

In case of zero-cost actions we calculate a fix-point for the current bucket by performing a BFS using only the zero-cost actions. The resulting states correspond to the application of one non-zero-cost action followed by a number of zero-cost actions. For duplicate elimination, we might also keep a BDD *closed* for all the states we have already expanded.

A* (Hart, Nilsson, and Raphael 1968) corresponds to Dijktra's algorithm with the new costs $\hat{c}(u,v) = c(u,v) - h(u) + h(v)$. Both algorithms, Dijkstra's algorithm with reweighted costs and A*, perform the same, if the heuristic function h is consistent, i. e., if for all $a = (u,v) \in \mathcal{A}$ we have $h(u) \leq h(v) + c(u,v)$.

Efficient symbolic adaptations of A* with consistent heuristics are SetA* by Jensen, Bryant, and Veloso (2002) and BDDA* by Edelkamp and Reffel (1998). While for Dijkstra's algorithm a bucket list was sufficient (in the case of no zero-cost actions), in BDDA* we need a two-dimensional matrix (cf. Figure 1). One dimension represents the distance from $\mathcal{I}$ (the g value), the other one the heuristic estimate on the distance to a goal (the h value).

A* expands according to $f = g + h$, which corresponds to a diagonal in the matrix. If the heuristic is consistent we will never have to reopen an f diagonal that we already expanded. Furthermore, as we do not allow negative action costs, the g value will never decrease. So, for each f diagonal we start at the bucket with smallest g value, expand this, go to the next one on the same f diagonal, until either the diagonal has been completely expanded or we find a goal state $t \in \mathcal{T}$. In the first case we go on to the next f diagonal, in the second case we are done.

For the case of zero-cost actions, things again get a bit more difficult, as we – similar to the symbolic version of Dijkstra's algorithm – need to store a list of BDDs in each bucket of the matrix to keep the different layers of the zero-cost BFS.

If, due to large action costs, the matrix becomes very large and at the same time very sparse, finding the next non-empty

bucket takes a long time and after a few steps it will not fit into the available memory any more. To avoid this we switched from using a matrix to a map. This way, only the non-empty (g, h)-positions are stored, which can greatly reduce the memory overhead.

A question that still remains is how we can come up with a consistent symbolic heuristic. For this, in the following we will present how to compute symbolic pattern databases.

Symbolic Planning Pattern Databases

Pattern databases have originally been proposed by Culberson and Schaeffer (1998). They were designed to operate in the *abstract space* of a problem. Thus, we first need to define such an *abstract planning problem*, for which we use one of several possibilities.

Definition 6 (Restriction Function)**.** *Let $\mathcal{V}' \subseteq \mathcal{V}$. For a set of variables $v \in \mathcal{V}$, a* restriction function $\phi_{\mathcal{V}'} : 2^{\mathcal{V}} \mapsto 2^{\mathcal{V}'}$ *is defined as the projection of S to $\mathcal{V}'$ containing only those variables that are also present in $\mathcal{V}'$, while those in $\mathcal{V} \setminus \mathcal{V}'$ are removed. $\phi_{\mathcal{V}'}(S)$ is also denoted as $S|_{\mathcal{V}'}$.*

Definition 7 (Abstract Planning Problem)**.** *Let $\mathcal{P} = \langle \mathcal{S}, \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{T}, c \rangle$ be a planning problem and $\mathcal{V}' \subseteq \mathcal{V}$ a set of variables. An* abstract planning problem $\mathcal{P}|_{\mathcal{V}'} = \langle \mathcal{S}|_{\mathcal{V}'}, \mathcal{V}', \mathcal{A}|_{\mathcal{V}'}, \mathcal{I}|_{\mathcal{V}'}, \mathcal{T}|_{\mathcal{V}'}, c \rangle$ *is $\mathcal{P}$ restricted to $\mathcal{V}'$, where $\mathcal{S}|_{\mathcal{V}'} = \{s|_{\mathcal{V}'} \mid s \in \mathcal{S}\}$, and $\mathcal{A}|_{\mathcal{V}'} = \{\langle pre|_{\mathcal{V}'}, \mathit{eff}|_{\mathcal{V}'} \rangle \mid \langle pre, \mathit{eff} \rangle \in \mathcal{A} \wedge \mathit{eff}|_{\mathcal{V}'} \neq \emptyset\}$*

In other words, the abstraction of a planning problem results in a (typically smaller) planning problem where certain variables (those in $\mathcal{V} \setminus \mathcal{V}'$) are ignored.

Definition 8 (Pattern Database)**.** *A* pattern database *(PDB) Φ for an abstract planning problem $\mathcal{P}_{\mathcal{V}'} = \langle \mathcal{S}|_{\mathcal{V}'}, \mathcal{V}', \mathcal{A}|_{\mathcal{V}'}, \mathcal{I}|_{\mathcal{V}'}, \mathcal{T}|_{\mathcal{V}'}, c \rangle$ is a set of pairs (d, s) where $d \in \mathbb{N}_0^+$ denotes the minimal distance of the state $s \in \mathcal{S}|_{\mathcal{V}'}$ to one of the abstract goal states, i. e., $d = \delta_{\mathcal{V}'}(s)$.*

The construction of symbolic PDBs (Edelkamp 2002) performs symbolic Dijkstra search. Starting at the goal states $\mathcal{T}|_{\mathcal{V}'}$ it operates in backward direction until all states are inserted into the database, which consists of a vector of BDDs, the BDD within each bucket of this vector representing the abstract states that have a corresponding distance to an abstract goal state.

If the abstract spaces are too large, the complete database calculation might take too long. Thus, we probably do not want the full information we could get. Therefore, we consider combining PDBs with *perimeter search* (Dillenburg and Nelson 1994).

Felner and Ofek (2007) propose two algorithms combining perimeter search and PDBs. The first one, called *simplified perimeter PDB* (SP_PDB) uses a classical PDB and a perimeter for a distance of d. All states not on the perimeter and also not within it are known to have a distance of at least $d + 1$. Thus, the heuristic estimate for a state n can be set to $h(n) = \max(\mathit{PDB}(n), d + 1)$ with $\mathit{PDB}(n)$ denoting the heuristic estimate that would be calculated by the PDB alone. So, the perimeter is used merely to correct too low heuristic estimates from the PDB up to the depth bound of $d + 1$.

The second approach, which they called *perimeter PDB* (P_PDB), performs a perimeter search up to a depth of d as a first step, followed by a PDB calculation starting at the states on the perimeter. The heuristic estimates result from the PDB and give estimates on the distance to the perimeter. The forward search is then performed using these heuristic estimates until a state on the perimeter is reached and chosen for expansion.

The idea of *partial PDBs*, which we use in our implementation, is due to Anderson, Holte, and Schaeffer (2007). A partial PDB is a PDB that is not calculated completely. Similar to perimeter search, it is created up to a maximal distance d to a goal state, which in this case is an abstract goal state. For all states that have been reached during the PDB construction process the heuristic estimate is calculated according to the PDB, while for states not within the PDB the estimate can be set to d (or even to $d + 1$, if all states up to distance d are stored in the PDB).

An adaptation to symbolic search is again rather simple. All we need to do is perform symbolic Dijkstra search starting at the goal states $\mathcal{T}|_{\mathcal{V}'}$ up to a maximal distance d and then stop the PDB creation process. In our implementation we do not set the value of d explicitly but generate the PDB until a certain timeout is reached, which we typically set to half the available time for one planning problem. The maximal distance up to which states are within the PDB is then denoted as d. As we are concerned with symbolic search and have one BDD for all states sharing the same distance to the goal states, all unexpanded states have a distance that is greater than d, so that we assign all such states a distance of $d + 1$.

Automated Pattern Selection

The automated selection of pattens for PDBs has been considered by Haslum et al. (2007). For one PDB the approach greedily constructs a pattern $\mathcal{V}'$ by adding one variable $v \in \mathcal{V}$ of the unchosen ones at a time. For the greedy selection process the quality of the unconstructed PDBs $\mathcal{V}' \cup \{v\}$ is evaluated by drawing and abstracting n random samples in the original state space. These subproblems are solved using heuristic search wrt. the already constructed PDB $\mathcal{V}'$. If v is fixed then the PDB for $\mathcal{V}' \cup \{v\}$ is constructed and the process iterates. The decision criterion for selecting the next candidate variable is the search tree prediction formula for iterative-deepening A* proposed by Korf, Reid, and Edelkamp (2001).

Similarly, our incremental symbolic PDB is constructed by greedily extending the variable set that is mentioned in the goal using backward search. However, we neither use sampling nor heuristics but construct the symbolic PDBs for all candidate variables. Moreover, the result is a single PDB, not a selection as in Haslum et al. (2007). If the pattern $\mathcal{V}' \subset \mathcal{V}$ is chosen, we construct the symbolic PDBs for $\mathcal{V}' \cup \{v\}$ for all variables $v \in \mathcal{V} \setminus \mathcal{V}'$ if the causal graph contains an edge from v to any of the variables in $\mathcal{V}'$. To select among the candidate patterns we then compute the mean heuristic value in the PDBs of the respective candidate variable by using model counting (sat-count as proposed by Bryant (1986)), which is a linear operation in the size of the

constructed symbolic PDB and a little simpler to compute than evaluating the prediction formula. After we are done with testing the insertion of all variables we actually add all those that achieved the best mean heuristic value (if that is greater than the mean heuristic value of $\mathcal{V}'$) and start the process over.

In contrast to explicit search, backward symbolic search in the according abstract state spaces is often possible even if the set of goal states is large. If the time reserved for construction is exhausted we use the PDB constructed so far as a partial PDB.

Bidirection

We observed that in some (but rare) domains unabstracted symbolic backward search is rather complex and limited to very few backward images that dominate the search process and require lots of resources in time and space. The reason is that the goal specification is partial, and in these domains backward search generates lots of states unreachable in forward direction, which results in tremendous work.

Fortunately, for most example problems we investigated this effect can be detected in the first backward iteration, resulting in a manifold increase of the time to compute the first pre-image. Hence, we perform one image from $\mathcal{I}$ and one pre-image from $\mathcal{T}$ and impose a threshold α on the time ratio of the two. If the ratio exceeds α we turn off bidirectional search (for perimeter database construction in the original space) and insist on partial PDB construction in the abstract space (as in the previous section), where due to the smaller size of the state vector the effect of backward search for the exploration is less prominent.

Experiments

We performed all the experiments on one core of a desktop computer (Intel $i7$ CPU with 2.67 GHz and 24 GB RAM). The planner is written in Java and uses the Java native interface to utilize Fabio Somenzi's CUDD library for the BDD calculations.

The set of competitors includes the two best planners at IPC 2008[1], namely the 2008 version of our planner (GAMER08) supporting BDDA* search with a symbolic perimeter database without abstraction (Edelkamp and Kissmann 2009) as well as the organizers' baseline planner BASE. The only difference between GAMER08 and the version used at IPC 2008 is the use of the map for the A* matrix, which we proposed in the our 2009 paper, so that the problems of the Parcprinter domain, which contains immensely large actions costs, can be handled. Concerning the other domains there is no difference in the number of solved instances. The baseline planner BASE performs Dijkstra search (A* with a zero-heuristic) and is based on LAMA's (Richter and Westphal 2010) search code.

After the competition promising improvements for explicit-state heuristic search planning have been proposed (see, e. g., Bonet and Helmert (2010) or Katz and Domshlak (2010)). Unfortunately, so far no results for such a planner on the IPC 2008 problems, the first ones to include action costs, have been published. The most recent planner that can handle cost-optimal and net-benefit planning problems is the GPU based planner GPUPLAN by Sulewski, Edelkamp, and Kissmann (2011), which is the third competitor in our experiments.

[1] ipc.informatik.uni-freiburg.de/Results

Table 1: Number of solved instances for cost-optimal planning benchmarks from IPC 2008.

Domain	BASE	GAMER08	GPUPLAN	GAMER
Elevators	16	**22**	19	21
Openstacks	23	21	27	**30**
Parcprinter	**11**	10	9	10
Pegsol	28	24	**29**	27
Scanalyzer	**12**	9	11	9
Sokoban	**24**	17	23	19
Transport	**11**	**11**	**11**	**11**
Woodworking	9	13	9	**20**
Total	134	127	138	**147**

For our current version of GAMER we set the parameters for the variable reordering to $\rho = 20$ and $\eta = 50{,}000$ resulting in one million incremental updates. Concerning the bidirection we set the ration to $\alpha = 25$.

The benchmarks are all the eight domains from the sequential-optimal track of IPC 2008, each consisting of 30 problem instances and containing non-uniform action costs. In contrast to the competition setting we use a shorter timeout of 15 minutes but no limit on the amount of usable memory.

The results are shown in Table 1. We see that GAMER performs significantly better than its competitors in the total number of solved instances.

In some individual domains like Scanalyzer and Sokoban the gap to explicit-state planners, however, has not always been bridged. We looked into the Sokoban benchmark and found that the lack of performance is due to the unfortunate problem specification in the domain. There are other problem formulations in which GAMER solves more problems than explicit-state planners.

In two of the domains, namely Openstacks and Woodworking, GAMER clearly outperforms all the competitors and is the only one to find a solution for all 30 Openstacks instances.

A comparison in terms of runtimes between all these planners would be rather unfair because the two versions of GAMER are the only planners in the chosen set that perform backward search for half the time (if it does not finish) and start the actual planning in forward direction only afterwards, so that in many cases the runtime is dominated by the time needed for the backward search. In our scenario, the timeout for the backward search is 450 seconds. To get a feeling for GAMER's gain in efficiency due to the modifications we provide a comparison of the runtimes of all instances solved by at least one version (cf. Figure 2).

We also experimented with computing the optimal net-benefit, as we intended to participate in the preference track again, which eventually was cancelled due to too few participants. For this, we chose all the net-benefit problems

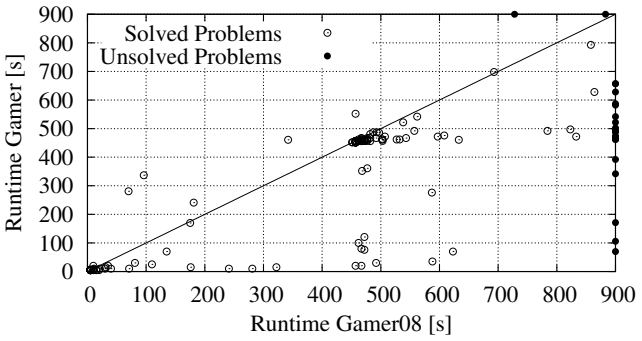

Figure 2: Comparison of the runtimes of the two versions of GAMER on the problems of the sequential optimal track of IPC 2008. *Solved Problems* are problems solved by both planners, while *Unsolved Problems* are those that were only solved by one planner.

Table 2: Number of solved instances for optimal net-benefit planning benchmarks from IPC 2008.

Domain	GAMER08	MIPS-XXL	GPUPLAN	GAMER
Crewplanning	4	**8**	**8**	4
Elevators	18	3	**19**	16
Openstacks	7	1	4	**11**
Pegsol	**29**	1	0	**29**
Transport	**14**	6	6	**14**
Woodworking	14	5	**15**	14
Total	86	24	52	**88**

from IPC 2008, each of the six domains again consisting of 30 problem instances. The competitors here are the winner and follow-up of the net-benefit track of IPC 2008, namely, GAMER08, performing uni-directional symbolic branch-and-bound-planning implemented in C++, and MIPS-XXL (Edelkamp and Jabbar 2008), an explicit-state breadth-first external-memory planner, as well as GPUPLAN. The results in numbers of solved instances are shown in Table 2 and the runtime comparison of the two versions of GAMER in Figure 3.

The results are not as good as in cost-optimal planning, due to the following reasons. First, the search for net-benefit domains is unidirectional, so that no PDB is generated and the bidirectional criterion does not appliy. The only advances are in the variable ordering. However, many net-benefit benchmarks are metric planning problems which contain numerical fluents (propositional atoms mapped to numbers). Fortunately, (after possible rescaling) all fluents map to small numbers and can be represented as state variables of finite domains. The only variable improvement we have applied so far considers the fluents. The dependencies of the corresponding state variables are not computed. The only reordering we enforced was that half of the numerical variables for the violation of the preferences are put at the start and half of them are placed to the end of the ordering. The reason for this rule of thumb is that we do not have dependencies among the violation variables and that the accumulated distance measurement to the other state

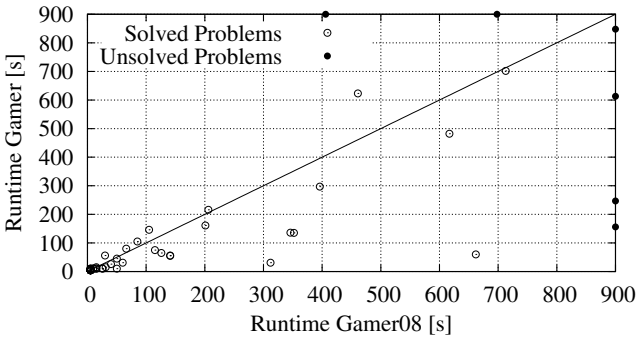

Figure 3: Comparison of the runtimes of the two versions of GAMER on the problems of the net-benefit track of IPC 2008. *Solved Problems* are problems solved by both planners, while *Unsolved Problems* are those that were only solved by one planner.

variables (with possible dependencies) can become smaller having them framed.

Conclusion

In this paper we pushed the envelope for symbolic planning with action costs and with soft constraints. Even though each applied technique is rather fundamental or aligns with ideas that have been applied to explicit-state planning, the combined impact with respect to the number of solved benchmarks is remarkable. The improvement wrt. the 2008 version of GAMER across the domains for minimizing action costs is statistically significant.

In the future we will refine the variable ordering heuristic for the numerical fluents. Also, experimenting with more general abstraction schemes and selection algorithms is on our research agenda. Furthermore, we are searching for a method for finding and using multiple PDBs effectively in most benchmark domains.

As there are no results available on action cost domains for alternative search approaches, we are exited to get hands on the planners after the planning competition IPC 2011.

References

Anderson, K.; Holte, R.; and Schaeffer, J. 2007. Partial pattern databases. In *SARA*, 20–34.

Bonet, B., and Helmert, M. 2010. Strengthening landmark heuristics via hitting sets. In *ECAI*, 329–334.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Devanur, N. R.; Khot, S.; Saket, R.; and Vishnoi, N. K. 2006. Integrality gaps for sparsest cut and minimum linear arrangement problems. In *STOC*, 537–546.

Dial, R. B. 1969. Shortest-path forest with topological ordering. *Communications of the ACM* 12(11):632–633.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Dillenburg, J. F., and Nelson, P. C. 1994. Perimeter search. *Artificial Intelligence* 65(1):165–178.

Edelkamp, S., and Jabbar, S. 2008. MIPS-XXL: Featuring external shortest path search for sequential optimal plans and external branch-and-bound for optimal net benefit. In *IPC*.

Edelkamp, S., and Kissmann, P. 2009. Optimal symbolic planning with action costs and preferences. In Boutilier, C., ed., *IJCAI*, 1690–1695.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In Herzog, O., and Günter, A., eds., *KI*, volume 1504 of *LNCS*, 81–92. Springer.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *AIPS*, 274–283. AAAI Press.

Felner, A., and Ofek, N. 2007. Combining perimeter search and pattern database abstractions. In *SARA*, 155–168.

Friedman, S. J., and Supowit, K. J. 1990. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers* 39(5):710–713.

Garey, M. R.; Johnson, D. S.; and Stockmeyer, L. 1976. Some simplified NP-complete graph problems. *Theoretical Computer Science* 1(3):237–267.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In Holte, R. C., and Howe, A., eds., *AAAI*, 1007–1012. AAAI Press.

Jensen, R. M.; Bryant, R. E.; and Veloso, M. M. 2002. SetA*: An efficient BDD-based heuristic search algorithm. In *AAAI*, 668–673. AAAI Press.

Katz, M., and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence* 174(12–13):767–798.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time Complexity of Iterative-Deepening-A*. *Artificial Intelligence* 129(1–2):199–218.

Lawler, E. L. 1963. The quadratic assignment problem. *Management Science* 9(4):586–599.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:127–177.

Sulewski, D.; Edelkamp, S.; and Kissmann, P. 2011. Exploiting the computational power of the graphics card: Optimal state space planning on the GPU. In *ICAPS*. AAAI Press.

LM-Cut: Optimal Planning with the Landmark-Cut Heuristic*

Malte Helmert
Albert-Ludwigs-Universität Freiburg, Germany
helmert@informatik.uni-freiburg.de

Carmel Domshlak
Technion, Israel
dcarmel@ie.technion.ac.il

Abstract

The LM-Cut planner uses the landmark-cut heuristic, introduced by the authors in 2009, within a standard A* progression search framework to find optimal sequential plans for STRIPS-style planning tasks. This short paper recapitulates the main ideas surrounding the landmark-cut heuristic and provides pointers for further reading.

Introduction

Heuristic search, either in the space of world states reached through progression or in the space of subgoals reached through regression, is a common and successful approach to classical planning. Apart from the choice of search algorithm, the main feature that distinguishes heuristic planners is their heuristic estimator. Most current heuristic functions are based on one of the following four ideas:

1. *delete relaxation*: e. g., h^+ (Hoffmann and Nebel 2001), $h^{\max}$ (Bonet and Geffner 2001), h^{add} (Bonet and Geffner 2001), h^{FF} (Hoffmann and Nebel 2001), h^{pmax} (Mirkis and Domshlak 2007), h^{sa} (Keyder and Geffner 2008)
2. *critical paths*: the h^m heuristic family (Haslum and Geffner 2000)
3. *abstraction*: pattern databases (Edelkamp 2001), merge-and-shrink abstractions (Helmert, Haslum, and Hoffmann 2007), and structural patterns (Katz and Domshlak 2008)
4. *landmarks*: LAMA's h^{LM} (Richter, Helmert, and Westphal 2008; Richter and Westphal 2010) and the admissible landmark heuristics h^{L} and h^{LA} (Karpas and Domshlak 2009)

These ideas have been developed in relative isolation. For a long time, apart from Haslum and Geffner's (2000) result that $h^{\max}$ is a special case of the h^m family ($h^{\max} = h^1$), no formal connections between these different ideas for devising heuristic estimators had been known. In a recent paper (Helmert and Domshlak 2009), we addressed this issue by proving a number of *dominance results*, which established, subject to the usual complexity-theoretic assumption that polynomial overhead is acceptable, the following relationships:

- Landmark heuristics dominate additive $h^{\max}$ heuristics.
- Additive $h^{\max}$ heuristics dominate landmark heuristics.
- Additive critical path heuristics with $m \geq 2$ strictly dominate landmark heuristics and additive $h^{\max}$ heuristics.
- Merge-and-shrink abstraction heuristics strictly dominate landmark heuristics and additive $h^{\max}$ heuristics.
- Pattern database heuristics are incomparable with landmark heuristics and additive $h^{\max}$ heuristics.

These statements are informal summaries, and some restrictions apply. In particular, the results for landmark heuristics only apply to *relaxation-based* landmarks, which are verifiable by a relaxed planning graph criterion. Until very recently, all landmark heuristics in the literature fell into this class. However, this has changed with the work of Keyder, Richter, and Helmert (2010), who introduced landmarks based on the h^m heuristic family.

On the positive side, all dominance results are constructive, showing how to compute a dominating heuristic in polynomial time. Moreover, some of the compilations are efficient enough to be worth implementing in practice. We implemented one such construction, from the regular (non-additive) $h^{\max}$ heuristic to landmarks, to obtain a new heuristic, which we called the *landmark-cut heuristic* $h^{\text{LM-cut}}$.

The Landmark-Cut Heuristic

The landmark-cut heuristic can alternatively be viewed as a landmark heuristic, a cost-partitioning scheme for additive $h^{\max}$, or an approximation to the (intractable) optimal relaxation heuristic h^+.

Here, we briefly recapitulate the computation of $h^{\text{LM-cut}}$. We assume familiarity with fundamental concepts such as delete relaxation, landmarks, and the $h^{\max}$ and h^+ heuristics. For readers who are new to these concepts, we refer to our original paper on $h^{\text{LM-cut}}$ (Helmert and Domshlak 2009) and the later work by Bonet and Helmert (2010), which related $h^{\text{LM-cut}}$ to hitting sets and showed that a generalization of $h^{\text{LM-cut}}$ based on hitting sets always achieves the perfect delete relaxation estimate h^+ when allowed exponential computation time.

To determine the $h^{\text{LM-cut}}$ estimate of a state s, we first compute $h^{\max}(s)$. If this value is zero or infinite, this implies that $h^+(s)$ is also zero or infinite, respectively, and hence

*Our presentation in this paper borrows heavily from the earlier paper in which we introduced the landmark-cut heuristic (Helmert and Domshlak 2009).

this is the best possible information that can be extracted from the delete relaxation of the task. In these cases, we set $h^{\text{LM-cut}}(s) = h^{\max}(s)$.

Otherwise, the cost to solve the delete relaxation of the given task from state s is finite and nonzero. In this case, we compute a *nontrivial disjunctive action landmark* of the delete relaxation, which is a set L of actions of nonzero cost such that each relaxed plan solving the task from the given state must include an action from L.

After computing such a landmark, we add the minimal cost c among all actions in L to the heuristic value computed so far (which is initialized to 0), reduce the cost of all actions in L by c, and then start again by recomputing the $h^{\max}$ values based on the reduced action costs, computing a new disjunctive action landmark, and so on. The process ends once action costs have been reduced to the extent that the $h^{\max}$ estimate of the resulting problem becomes zero.

The main challenge in this computation is finding a suitable landmark L. It is not particularly hard to find *some* such landmark: the set of all actions of nonzero cost will do. However, the larger the set L, the more actions will have their cost reduced at the end of the current iteration of the main landmark-cut loop, leading to potentially fewer landmarks that can be extracted in future rounds. The challenge, then, is in finding a reasonably small such set.

The landmark-cut heuristic addresses this issue by computing so-called *justification graphs*, which "justify" the $h^{\max}$ values of the facts of the planning task by linking each effect of an action a to the most expensive precondition of a (or one of the most expensive ones, in case of ties).

Arcs in justification graphs are weighted by the costs of the actions that induce them. A shortest paths in a justification graph corresponds to a causal chain whose cost explains the $h^{\max}$ value of a fact, and *cuts* in justification graphs (sets of arcs whose removal disconnects the current state from the goal) correspond to disjunctive action landmarks. These relationships are explored in more depth by Bonet and Helmert (2010), who show that *all* relevant landmarks of the delete relaxation can be computed as cuts in justification graphs when arbitrary preconditions are allowed to induce arcs in the graph (rather than just preconditions with maximal $h^{\max}$ value).

The landmark selected by the landmark-cut heuristic is based on a particular cut close to the goal facts of the task, which is sufficient to guarantee that the final heuristic value is always at least as large as $h^{\max}(s)$. (In our experiments, it is usually much larger.)

The LM-Cut Planner

In our earlier work (Helmert and Domshlak 2009), we demonstrated experimentally that $h^{\text{LM-cut}}$ gives excellent approximations to h^+ and compares favourably to other admissible planning heuristics in terms of accuracy. We also showed that an optimal planner based on A* search with the landmark-cut heuristic was highly competitive with the state of the art at the time.

The LM-Cut planner entered into IPC 2011 is almost identical to the system used in these experiments. The only two changes since then are:

- minor bug fixes and performance improvements in various components of the Fast Downward planner that serves as the basis of our implementation, and
- support for actions of non-unit cost. (While our original description of the landmark-cut heuristic was fully general, our implementation was restricted to the unit-cost case.)

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Bonet, B., and Helmert, M. 2010. Strengthening landmark heuristics via hitting sets. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 329–334. IOS Press.

Edelkamp, S. 2001. Planning with pattern databases. In Cesta, A., and Borrajo, D., eds., *Pre-proceedings of the Sixth European Conference on Planning (ECP 2001)*, 13–24.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 140–149. AAAI Press.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1728–1733.

Katz, M., and Domshlak, C. 2008. Structural patterns heuristics via fork decomposition. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 182–189. AAAI Press.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 588–592.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the*

19th European Conference on Artificial Intelligence (ECAI 2010), 335–340. IOS Press.

Mirkis, V., and Domshlak, C. 2007. Cost-sharing approximations for h^+. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 240–247. AAAI Press.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 975–982. AAAI Press.

The Merge-and-Shrink Planner: Bisimulation-based Abstraction for Optimal Planning

Raz Nissim
Ben-Gurion University
Beer-Sheva, Israel
raznis@cs.bgu.ac.il

Jörg Hoffmann
INRIA
Nancy, France
joerg.hoffmann@inria.fr

Malte Helmert
University of Freiburg
Freiburg, Germany
helmert@informatik.uni-freiburg.de

Abstract

Merge-and-shrink abstraction is a general approach to heuristic design whose key advantage is its capability to make very fine-grained choices in defining abstractions. The Merge-and-shrink planner uses two different strategies for making these choices, both based on the well-known notion of bisimulation. The resulting heuristics are used in two sequential runs of A* search.

Introduction

Many optimal planning systems are based on state-space search using A* and admissible heuristics. Merge-and-shrink abstraction (Dräger *et al.* 2006; Helmert *et al.* 2007), short M&S, uses solution distances in a smaller, *abstract* state space to deliver a consistent and admissible heuristic function.

The abstract state space is built in an incremental fashion, starting with a set of atomic abstractions corresponding to individual variables, then iteratively *merging* two abstractions – replacing them with their synchronized product – and *shrinking* them – aggregating pairs of states into one. Thus, despite the exponential size of the state space, M&S allows to select individual pairs of states to aggregate. This freedom in abstraction design comes with significant advantages. M&S dominates most other known frameworks for computing admissible planning heuristics: for any given state, it can with polynomial overhead compute a larger lower bound (Helmert and Domshlak 2009).

The M&S planner employs two different shrinking strategies, which choose the states to aggregate using the notion of bisimulation. In this paper we briefly describe bisimulation and label reduction, and conclude with a detailed description of the M&S planner.

Background

Our approach is based on the notion of *bisimulation*, a well-known criterion under which an abstract state space "exhibits the same observable behavior" as the original state space (Milner 1990). Two states s, t are bisimilar if: (1) they agree on whether or not the goal is true; and (2) every transition label, i.e., every planning operator, leads into the same abstract state from both s and t. If we aggregate only bisimilar states during M&S, then the heuristic is guaranteed to be perfect. However, bisimulations are exponentially big even in trivial examples. Our key observation is that, for the purpose of computing a heuristic, we can relax bisimulation significantly without losing any information. Namely, we do not need to distinguish the transition labels. Such a *fully label-reduced* bisimulation still preserves solution distance, while often being exponentially smaller.

Unfortunately, while full label reduction does not affect solution distances per se, its application within the M&S framework is problematic. The merging step, in order to synchronize transitions, needs to know which ones share the same label. We tackle this by using *partial* label reductions, ignoring the difference between two labels only if they are equivalent for "the rest" of the M&S construction. We thus obtain, again, a strategy that guarantees to deliver a perfect heuristic.

Even label-reduced bisimulations are sometimes too big, thus for practicality one needs a strategy to approximate further if required. The M&S planner uses two such strategies, each relaxing the strict rules of bisimulation in a different way.

For more details on bisimulation, label reduction, and using their combination to create perfect heuristics in polynomial time in some planning domains, we refer to a conference paper (forthcoming).

The M&S Strategies

The merging strategy we use is linear (meaning only one non-atomic abstraction is maintained), and follows Fast-Downward's "level heuristic" (Helmert 2006). This orders variables so that those "closest to the root of the causal graph" go first (this is beneficial for operator projection because the most influential variables are projected away earlier on).

Our planner uses two shrinking strategies, each having different strengths and weaknesses. After experimenting with many shrinking strategies, we did not find one that greatly outperformed the others. Our choice of strategies for our planner was therefore guided by the relatively high variance of tasks solved by the two. Since these strategies are the core feature of the planner, we describe them in some more detail in what follows.

The Greedy Bisimulation Shrinking Strategy

The bisimulation shrinking strategy computes the coarsest bisimulation, and in the shrinking step, aggregates only bisimilar (abstract) states. In most benchmark domains, however, coarsest bisimulations are still large even under operator projection. Greedy bisimulation is a relaxed variant of bisimulation, which demands the bisimulation property only for transitions $s \to s'$ where the abstract goal distance from s is at most as large as the abstract goal distance from s'. This relaxation forfeits the guarantee of providing a perfect heuristic.

The **greedy bisimulation shrinking strategy** therefore computes the coarsest greedy bisimulation using partial label reduction, aggregating only greedily bisimilar states. Since the (greedy) bisimulation strategy is given no limit on the size of the abstraction, the actual abstraction size depends only on the size of the coarsest greedy bisimulation. We observed that using greedy bisimulation dramatically reduces abstraction size, increasing the number of cases where the abstraction can be built completely. In fact, when using the bisimulation strategy, the abstraction was built completely only in 203 out of 876 IPC tasks we experimented on. Using greedy bisimulation brings this number up to 795. This reduction in size also improves speed, making the abstraction much faster to compute.

The DFP-gop Shrinking Strategy

Motivated by the size of bisimulations, Dräger et al. (2006) propose a more approximate shrinking strategy that we will call the **DFP shrinking strategy**. When building the coarsest bisimulation, the strategy keeps separating states until the size limit N is reached. The latter may happen before a bisimulation is obtained, in which case we may lose information. The strategy prefers to separate states close to the goal, thus attempting to make errors only in more distant states where the errors will hopefully not be as relevant.

The **DFP-gop** shrinking strategy ("g" stands for greedy, "op" for operator projection) enhances the DFP strategy in two ways. First, partial label reduction is used when computing the coarsest bisimulation. Second, if bisimulation breaks the abstraction size limit, the greedy coarsest bisimulation is used to select which states to aggregate. For the abstraction size limit, we chose to set $N = 200,000$. Because DFP-gop aggregates only bisimilar states as long as N is not reached, its high value allows computation of perfect heuristics, in cases where there exist sufficiently small coarsest bisimulations. In all 20 tasks of the IPC benchmark domain Gripper, for example, the final abstraction computed by this strategy is a bisimulation of the original search space, and therefore provides the perfect heuristic.

The high N value comes at a cost – computing the abstraction is slow and requires much memory. Out of the 876 tasks experimented on, only in 490 was the final abstraction computed without running out of time/memory.

The Planner

The M&S planner is implemented on top of the Fast Downward planning system. For further information on Fast Downward's PDDL-to-finite-domain translator, please refer to the paper by Helmert (2009). For details regarding how M&S abstractions are used in the search process, refer to the paper by Helmert et al. (2007). Finally, for general information about the planner, we refer the reader to its original description (Helmert 2006).

The Hybrid Implementation

In order to take advantage of the strengths of both strategies, our planner is designed to divide the given time limit between two sequential runs, using $\frac{4}{9}$ of the available time for the greedy bisimulation shrinking strategy, followed – if no solution is found – with DFP-gop for the remaining time. (The value of $\frac{4}{9}$ was determined experimentally based on data for IPC 1998–2008 benchmarks.) In each run of the planner, the M&S abstraction is computed according to the strategy, and A* search is performed using solution distances in the abstraction as heuristic values.

We chose the hybrid implementation for two reasons. First, cutting the time limit had little effect on coverage results for the two strategies. This allowed us to take advantage of two different M&S strategies. Second, we tried computing both abstractions and using the maximal of the two in one search run, but this turned out to be too costly both in time and in memory. In most cases, construction of the two abstractions either exceeded the 30 minute time limit or the 2GB memory limit.

References

Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In *Proc. SPIN 2006*, pages 19–34, 2006.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, pages 162–169, 2009.

Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS 2007*, pages 176–183, 2007.

Malte Helmert. The Fast Downward planning system. *JAIR*, 26:191–246, 2006.

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *AIJ*, 173:503–535, 2009.

Robin Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of TCS*, pages 1201–1242. 1990.

The SelMax Planner: Online Learning for Speeding up Optimal Planning*

Carmel Domshlak Technion
Malte Helmert Albert-Ludwigs-Universität Freiburg
Erez Karpas Technion
Shaul Markovitch Technion

Abstract

The SelMax planner combines two state-of-the-art admissible heuristics using an online learning approach. In this paper we describe the online learning approach employed by SelMax, briefly review the Fast Downward framework, and describe the SelMax planner.

Introduction

One of the most prominent approaches to cost-optimal planning is using the A^* search algorithm with an admissible heuristic. Many admissible heuristics have been proposed, varying from cheap to compute yet typically not very informative to expensive to compute but often very informative. Since the accuracy of heuristic functions varies for different problems, and even for different states of the same problem, we can produce a more robust optimal planner by combining several admissible heuristics. The simplest way of doing this is by using their point-wise maximum at each state. Presumably, each heuristic is more accurate, that is, provides a higher estimate, in different regions of the search space, and thus their maximum is at least as accurate as each of the individual heuristics. In some cases it is also possible to use additive (Felner, Korf, and Hanan 2004; Haslum, Bonet, and Geffner 2005; Katz and Domshlak 2008) or mixed additive/maximizing (Coles et al. 2008; Haslum et al. 2007) combinations of admissible heuristics.

An important issue with both max-based and sum-based approaches is that the benefit of adopting them over sticking to just a single heuristic is assured only if the planner is not constrained by time. Otherwise, the time spent on computing numerous heuristic estimates at each state may outweigh the time saved by reducing the number of expanded states. Selective Max (SelMax) is a novel method for combining admissible heuristics that aims at providing the accuracy of their max-based combination while still computing just a single heuristic for each search state.

At a high level, selective max can be seen as a hyper-heuristic (Burke et al. 2003) — a heuristic for choosing between other heuristics. Specifically, selective max is based on a seemingly useless observation that, if we had an oracle indicating the most accurate heuristic for each state, then computing only the indicated heuristic would provide us with the heuristic estimate of the max-based combination. In practice, of course, such an oracle is not available. However, in the time-limited settings of our interest, this is not our only concern: It is possible that the extra time spent on computing the more accurate heuristic (indicated by the oracle) may not be worth the time saved by the reduction in expanded states.

Addressing the latter concern, we first analyze an idealized model of a search space and deduce a decision rule for choosing a heuristic to compute at each state when the objective is to minimize the overall search time. Taking that decision rule as our target concept, we then describe an online active learning procedure for that concept that constitutes the essence of selective max.

Notation

We consider planning in the SAS$^+$ formalism (Bäckström and Nebel 1995); a SAS$^+$ description of a planning task can be automatically generated from its PDDL description (Helmert 2009). A SAS$^+$ task is given by a 4-tuple $\Pi = \langle V, A, s_0, G\rangle$. $V = \{v_1, \ldots, v_n\}$ is a set of *state variables*, each associated with a finite domain $dom(v_i)$. Each complete assignment s to V is called a *state*; s_0 is an *initial state*, and the *goal* G is a partial assignment to V. A is a finite set of *actions*, where each action a is a pair $\langle \mathsf{pre}(a), \mathsf{eff}(a)\rangle$ of partial assignments to V called *preconditions* and *effects*, respectively.

An action a is applicable in a state s iff $\mathsf{pre}(a) \subseteq s$. Applying a changes the value of each state variable v to $\mathsf{eff}(a)[v]$ if $\mathsf{eff}(a)[v]$ is specified. The resulting state is denoted by $s[\![a]\!]$; by $s[\![\langle a_1, \ldots, a_k\rangle]\!]$ we denote the state obtained from sequential application of the (respectively applicable) actions $a_1, \ldots, a_k$ starting at state s. Such an action sequence is a plan if $G \subseteq s_0[\![\langle a_1, \ldots, a_k\rangle]\!]$.

A Model for Heuristic Selection

Given a set of admissible heuristics and the objective of minimizing the overall search time, we are interested in a decision rule for choosing the right heuristic to compute at each search state. In what follows, we derive such a decision rule for a pair of admissible heuristics with respect to an idealized search space model corresponding to a tree-structured

*This paper is strongly based upon Domshlak, Karpas, and Markovitch (2010)

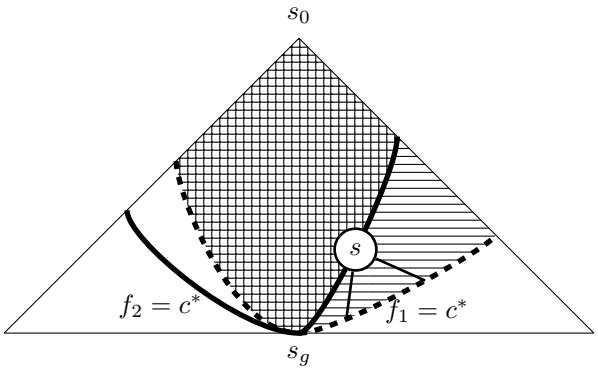

Figure 1: An illustration of the idealized search space model and the f-contours of two admissible heuristics.

search space with a single goal state, constant branching factor b, and uniform cost actions (Pearl 1984). Two additional assumptions we make are that the heuristics are consistent, and that the time t_i required for computing heuristic h_i is independent of the state being evaluated; w.l.o.g. we assume $t_2 \geq t_1$. Obviously, most of the above assumptions do not hold in typical search problems, and later we carefully examine their individual influences on our framework.

Adopting the standard notation, let $g(s)$ be the cost of the cheapest path from s_0 to s. Defining $\max_h(s) = \max(h_1(s), h_2(s))$, we then use the notation $f_1(s) = g(s) + h_1(s)$, $f_2(s) = g(s) + h_2(s)$, and $\max_f(s) = g(s) + \max_h(s)$. The A^* algorithm with a heuristic h expands states in increasing order of $f = g + h$. Assuming the goal state is at depth c^*, let us consider the states satisfying $f_1(s) = c^*$ (the dotted line in Fig. 1) and those satisfying $f_2(s) = c^*$ (the solid line in Fig. 1). The states above the $f_1 = c^*$ and $f_2 = c^*$ contours are those that are surely expanded by A^* with h_1 and h_2, respectively. The states above both these contours (the grid-marked region in Fig. 1), that is, the states $SE = \{s \mid \max_f(s) < c^*\}$, are those that are surely expanded by A^* using $\max_h$ (see Theorem 4, p. 79, Pearl 1984).

Under the objective of minimizing the search time, observe that the optimal decision for any state $s \in SE$ is not to compute any heuristic at all, since all these states are surely expanded anyway. The optimal decision for all other states is a bit more complicated. $f_2 = c^*$ contour that separates between the grid-marked and lines-marked areas. Since $f_1(s)$ and $f_2(s)$ account for the same $g(s)$, we have $h_2(s) > h_1(s)$, that is, h_2 is more accurate in state s than h_1. If we were interested solely in reducing state expansions, then h_2 would obviously be the right heuristic to compute at s. However, for our objective of reducing the actual search time, h_2 may actually be the wrong choice because it might be much more expensive to compute than h_1.

Let us consider the effects of each of our two alternatives. If we compute $h_2(s)$, then s is no longer surely expanded since $f_2(s) = c^*$, and thus whether A^* expands s or not depends on tie-breaking. In contrast, if we compute $h_1(s)$, then s is surely expanded because $f_1(s) < c^*$. Note that not computing h_2 for s and then computing h_2 for one of the descendants s' of s is surely a sub-optimal strategy as we do pay the cost of computing h_2, yet the pruning of A^* is limited only to the search sub-tree rooted in s'. Therefore, our choices are really either computing h_2 for s, or computing h_1 for all the states in the sub-tree rooted in s that lie on the $f_1 = c^*$ contour. Suppose we need to expand l complete levels of the state space from s to reach the $f_1 = c^*$ contour. This means we need to generate order of b^l states, and then invest $b^l t_1$ time in calculating h_1 for all these states that lie on the $f_1 = c^*$ contour. In contrast, suppose we choose to compute $h_2(s)$. Assuming favorable tie-breaking, the time required to "explore" the sub-tree rooted in s will be t_2.

Putting things together, the optimal decision in state s is thus to compute h_2 iff $t_2 < b^l t_1$, or if we rewrite this, if

$$l > \log_b(t_2/t_1).$$

As a special case, if both heuristics take the same time to compute, this decision rule boils down to $l > 0$, that is, the optimal choice is simply the more accurate (for state s) heuristic.

The next step is to somehow estimate the "depth to go" l. For that, we make another assumption about the rate at which f_1 grows in the sub-tree rooted at s. Although there are many possibilities here, we will look at two estimates that appear to be quite reasonable. The first estimate assumes that the h_1 value remains constant in the subtree rooted at s, that is, the additive error of h_1 increases by 1 for each level below s. In this case, f_1 increases by 1 for each expanded level of the sub-tree (because h_1 remains the same, and g increases by 1), and it will take expanding $\Delta_h(s) = h_2(s) - h_1(s)$ levels to reach the $f_1 = c^*$ contour. The second estimate we examine assumes that the absolute error of h_1 remains constant, that is, h_1 increases by 1 for each level expanded, and so f_1 increases by 2. In this case, we will need to expand $\Delta_h(s)/2$ levels. This can be generalized to the case where the estimate h_1 increases by any constant additive factor c, which results in $\Delta_h(s)/(c+1)$ levels being expanded. In either case, the dependence of l on $\Delta_h(s)$ is linear, and thus our decision rule can be reformulated to compute h_2 if

$$\Delta_h(s) > \alpha \log_b(t_2/t_1),$$

where α is a hyper-parameter for our algorithm. Note that, given b, t_1, and t_2, the quantity $\alpha \log_b(t_2/t_1)$ becomes fixed and in what follows we denote simply by *threshold* τ.

Dealing with Model Assumptions

The idealized model above makes several assumptions, some of which appear to be very problematic to meet in practice. Here we examine these assumptions more closely, and when needed, suggest pragmatic compromises.

First, the model assumes that the search space forms a tree with a single goal state and uniform cost actions, and that the heuristics in question are consistent. Although the first assumption does not hold in most planning problems, and the second assumption is not satisfied by some state-of-the-art heuristics, they do not prevent us from using the decision rule suggested by the model. Furthermore, there is some empirical evidence to support our conclusion about exponential growth of the search effort as a function of heuristic

error, even when the assumptions made by the model do not hold. In particular, the experiments of Helmert and Röger (2008) with heuristics with small constant additive errors clearly show that the number of expanded nodes typically grows exponentially as the (still very small and additive) error increases.

The model also assumes that both the branching factor and the heuristic computation times are constant across the search states. In our application of the decision rule to planning in practice, we deal with this assumption by adopting the average branching factor and heuristic computation times, estimated from a random sample of search states. Finally, the model assumes perfect knowledge about the surely expanded search states. In practice, this information is obviously not available. We approach this issue conservatively by treating all the examined search states as if they were on the decision border, and thus apply the decision rule at all the search states. Note that this does not hurt the correctness of our algorithm, but only costs us some heuristic computation time on the surely expanded states. Identifying the surely expanded region during search is the subject of ongoing work, and can hopefully be used to improve search efficiency even further.

Online Learning of the Selection Rule

Our decision rule for choosing a heuristic to compute at a given search state s suggests to compute the more expensive heuristic h_2 when $h_2(s) - h_1(s) > \tau$. However, computing $h_2(s) - h_1(s)$ requires computing in s both heuristics, defeating the whole purpose of reducing search time by selectively evaluating only one heuristic at each state. To overcome this pitfall, we take our decision rule as a target concept, and suggest an *active online learning* procedure for that concept. Intuitively, our concept is the set of states where the more expensive heuristic h_2 is "significantly" more accurate than the cheaper heuristic h_1. According to our model, this corresponds to the states where the reduction in expanded states by computing h_2 outweighs the extra time needed to compute it. In what follows, we present our learning-based methodology in detail, describing the way we select and label training examples, the features we use to represent the examples, the way we construct our classifier, and the way we employ it within A^* search.

To build a classifier, we first need to collect training examples, which should be representative of the entire search space. One option for collecting the training examples is to use the first k states of the search where k is the desired number of training examples. However, this method has a bias towards states that are closer to the initial state, and therefore is not likely to well represent the search space. Hence, we instead collect training examples by sending "probes" from the initial state. Each such "probe" simulates a stochastic hill-climbing search with a depth limit cutoff. All the states generated by such a probe are used as training examples, and we stop probing when k training examples have been collected. In our evaluation, the probing depth limit was set to twice the heuristic estimate of the initial state, that is $2\max_h(s_0)$, and the next state s for an ongoing probe was chosen with a probability proportional to $1/\max_h(s)$.

evaluate(s)
$\langle h, confidence\rangle :=$ CLASSIFY($s, model$)
if ($confidence > \rho$) **then return** $h(s)$
else
  $label := h_1$
  if $h_2(s) - h_1(s) > \alpha \log_b(t_2/t_1)$ **then** $label := h_2$
  update $model$ with $\langle s, label\rangle$
  return $\max(h_1(s), h_2(s))$

Figure 2: The selective max state evaluation procedure.

This "inverse heuristic" selection biases the sample towards states with lower heuristic estimates, that is, to states that are more likely to be expanded during the search. It is worth noting here that more sophisticated procedures for search space sampling have been proposed in the literature (e.g., see Haslum et al. 2007), but as we show later, our much simpler sampling method is already quite effective for our purpose.

After the training examples T are collected, they are first used to estimate b, t_1 and t_2 by averaging the respective quantities over T. Once b, t_1 and t_2 are estimated, we can compute the threshold $\tau = \alpha \log_b(t_2/t_1)$ for our decision rule. We generate a label for each training example by calculating $\Delta_h(s) = h_2(s) - h_1(s)$, and comparing it to the decision threshold. If $\Delta_h(s) > \tau$, we label s with h_2, otherwise with h_1. If $t_1 > t_2$ we simply switch between the heuristics—our decision is always *whether to compute the more expensive heuristic or not*; the default is to compute the cheaper heuristic, unless the classifier says otherwise.

Besides deciding on a training set of examples, we need to choose a set of features to represent each of these examples. The aim of these features is to characterize search states with respect to our decision rule. While numerous features for characterizing states of planning problems have been proposed in previous literature (see, e.g., Yoon, Fern, and Givan (2008); de la Rosa, Jiménez, and Borrajo (2008)), they were all designed for inter-problem learning, and most of them are not suitable for intra-problem learning like ours. In our work we decided to use only elementary features corresponding simply to the actual state variables of the planning problem.

Once we have our training set and features to represent the examples, we can build a binary classifier for our concept. This classifier can then play the role of our hypothetical oracle indicating which heuristic to compute where. However, as our classifier is not likely to be a perfect such oracle, we further consult the confidence the classifier associates with its classification. The resulting state evaluation procedure of selective max is depicted in Figure 2. If state s is to be evaluated by A^*, we use our classifier to decide which heuristic to compute. If the classification confidence exceeds a parameter threshold ρ, then only the indicated heuristic is computed for s. Otherwise, we conclude that there is not enough information to make a selective decision for s, and compute the regular maximum over $h_1(s)$ and $h_2(s)$. However, we use this opportunity to improve the quality of our prediction for states similar to s, and update our classifier. This is done by generating a label based on $h_2(s) - h_1(s)$ and learning from

this new example.[1] This can be viewed as the active part of our learning procedure.

The last decision to be made is the choice of classifier. Although many classifiers can be used here, there are several requirements that need to be met due to our particular setup. First, both training and classification must be very fast, as both are performed during time-constrained problem solving. Second, the classifier must be incremental to allow online update of the learned model. Finally, the classifier should provide us with a meaningful confidence for its predictions. While several classifiers meet these requirements, we found the classical Naive Bayes classifier to provide a good balance between speed and accuracy (Mitchell 1997). One note on the Naive Bayes classifier is that it assumes a very strong conditional independence between the features. Although this is not a fully realistic assumption for planning problems, using a SAS$^+$ formulation of the problem instead of the classical STRIPS helps a lot: instead of many binary variables which are highly dependent upon each other, we have a much smaller set of variables which are less dependent upon each other.

As a final note, extending selective max to use more than two heuristics is rather straightforward—simply compare the heuristics in a pair-wise manner, and choose the best heuristic by a vote, which can either be a regular vote (i.e., 1 for the winner, 0 for the loser), or weighted according to the classifier's confidence. Although this requires a quadratic number of classifiers, training and classification time (at least with Naive Bayes) appear to be much lower than the overall time spent on heuristic computations, and thus the overhead induced by learning and classification is likely to remain relatively low.

The Fast Downward Planning Framework

We have implemented selective max on top of the Fast Downward planning system. In this section we review the relevant (for optimal planning) capabilities of the IPC-2011 version of the Fast Downward planning system. Since Fast Downward incorporates many different algorithms and approaches, which have each been published separately in peer-reviewed conferences and/or journals, we will simply list the available components with pointers to further information for the interested reader.

The Fast Downward planning system (Helmert 2006) is composed of three main parts: the translator, the preprocessor, and the search component, which are run sequentially in this order. The translator (Helmert 2009) is responsible for translating the given PDDL task into an equivalent one in SAS$^+$ representation. This is done by finding groups of propositions which are mutually exclusive and combining them into a single SAS$^+$ variable. The preprocessor performs a relevance analysis and precomputes some data structures that are used by the search and certain heuristics. The search component then searches for a solution to the given SAS$^+$ task.

[1] We do not change the estimates for b, t_1 and t_2, so the threshold τ remains fixed.

Search The search component features three main types of search algorithms: eager best-first search, lazy best-first search (Richter and Helmert 2009), and enforced hill-climbing (Hoffmann and Nebel 2001). For the purposes of optimal planning, only eager search is relevant, since A^* is implemented on top of eager search by using $f = g + h$ and tie-breaking on h.

Heuristics Selective-max can combine arbitrary admissible heuristics from among the following admissible heuristics which are implemented in Fast Downward:

- Blind — 0 for goal states, 1 (or cheapest action cost for non-unit-cost tasks) for non-goal states
- $h^{\max}$ (Bonet, Loerincs, and Geffner 1997; Bonet and Geffner 1999) — the relaxation-based maximum heuristic
- h^m (Haslum and Geffner 2000) — a very slow implementation of the h^m heuristic family
- $h^{\text{M\&S}}$ (Helmert, Haslum, and Hoffmann 2007; 2008) — the merge-and-shrink heuristic
- h_{LA} (Karpas and Domshlak 2009; Keyder, Richter, and Helmert 2010) — the admissible landmark heuristic
- $h^{\text{LM-cut}}$ (Helmert and Domshlak 2009) — the landmark-cut heuristic

Chosen Configuration

Given the number of parameters available for selective-max, as well as the wealth of options of choosing which heuristics to combine, it is difficult to choose one configuration for a submission to the IPC. One option (which was implemented in the FD Autotune planner) is to use some automated algorithm configuration tool (Hutter et al. 2009) to choose a configuration.

In this submission, we chose to combine the two best heuristics available in Fast Downward (according to previous empirical results): $h^{\text{LM-cut}}$ (Helmert and Domshlak 2009) and h_{LA} (Karpas and Domshlak 2009; Keyder, Richter, and Helmert 2010). Since we are using h_{LA}, we also use the *LM-A** search algorithm (rather than regular A^*).

The h_{LA} heuristic uses landmarks generated by two methods: the RHW method (Richter, Helmert, and Westphal 2008) and h^m landmarks with $m = 1$ (Keyder, Richter, and Helmert 2010), which were combined into the same landmark graph (see BJOLP submission paper for details). The parameters for selective-max were chosen based on a limited set of experiments, and are described in the following table:

Parameter	Value
α (heuristic difference bias)	1
ρ (confidence threshold)	0.6
initial sample size	1000
Sampling Method	Probing
Classifier	Naive Bayes

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS^+ planning. *Comp. Intell.* 11(4):625–655.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *ECP*, 360–372.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *AAAI*, 714–719.

Burke, E.; Kendall, G.; Newall, J.; Hart, E.; Ross, P.; and Schulenburg, S. 2003. Hyper-heuristics: an emerging direction in modern search technology. In *Handbook of meta-heuristics*. chapter 16, 457–474.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. Additive-disjunctive heuristics for optimal planning. In *ICAPS*, 44–51.

de la Rosa, T.; Jiménez, S.; and Borrajo, D. 2008. Learning relational decision trees for guiding heuristic planning. In *ICAPS*, 60–67.

Domshlak, C.; Karpas, E.; and Markovitch, S. 2010. To max or not to max: Online learning for speeding up optimal planning. In *AAAI*, 1071–1076.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. 22:279–318.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *AIPS*, 140–149.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, 1007–1012.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *AAAI*, 1163–1168.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 162–169.

Helmert, M., and Röger, G. 2008. How good is almost perfect? In *AAAI*, 944–949.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2008. Explicit-state abstraction: A new method for generating heuristic functions. In *AAAI*, 1547–1550.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *IJCAI*, 1728–1733.

Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In *ICAPS*, 174–181.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In *ECAI*, 335–340.

Mitchell, T. M. 1997. *Machine Learning*. New York: McGraw-Hill.

Pearl, J. 1984. *Heuristics — Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *ICAPS*, 273–280.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *AAAI*, 975–982.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. 9:683–718.

ArvandHerd: Parallel Planning with a Portfolio

Richard Valenzano, Hootan Nakhost, Martin Müller, Jonathan Schaeffer
University of Alberta
{valenzan, nakhost, mmueller, jonathan}@cs.ualberta.ca

Nathan Sturtevant
University of Denver
sturtevant@cs.du.edu

Abstract

ArvandHerd is a satisficing parallel planner that has been entered in the 2011 International Planning Competition (IPC 2011). It uses a portfolio-based approach where the portfolio contains four configurations of the Arvand planner and one configuration of the LAMA planner. Each processor runs a single planner, and the execution is mostly independent from the other processors so as to minimize overhead due to communication. ArvandHerd also uses the Aras plan-improvement system to improve plan quality.

Introduction

If a planner is to be successful, it must be able to handle problems from a diverse set of domains. Unfortunately, no single algorithm can be expected to dominate all other challengers on all possible domains. Even within a single domain, it has been shown that to achieve the best possible performance, it is often necessary to use different *parameterizations* or *configurations* of an algorithm on different problems (Valenzano et al. 2010). These issues suggest the use of an *algorithm porfolio*. This means that instead of using a single strategy, problems should be tackled with a set of strategies that differ by their configuration or in the underlying algorithm.

For parallel planning, different members of the portfolio can be assigned to separate processors. This is a simple alternative to the difficult process of parallelizing a single-core algorithm and it mostly avoids overhead from communication and synchronization. These ideas form the backbone of our ArvandHerd planner.

In this paper, we begin with a description of the individual members of the ArvandHerd portfolio. This is followed by a description of the general architecture of ArvandHerd, including communication between processors, memory management, and the use of the Aras plan-improvement system.

The ArvandHerd Portfolio

The portfolio was selected so as to maximize the coverage of ArvandHerd by including different configurations of two significantly different planning approaches. More specifically, the portfolio contains four configurations of the random walk based Arvand planner (Nakhost and Müller 2009) and one configuration of the WA*-based LAMA planner (Richter and Westphal 2010). Below, these planners and their configurations are described in more detail.

The Arvand Planner

Arvand is a sequential satisficing planner that uses heuristically evaluated random walks as the basis for its search. The execution of Arvand consists of a series of *search episodes*. In the simplest version of Arvand, each search episode begins with n random walks, each being a sequence of m legal random actions originating from the initial state s_i, where n and m are parameters. The heuristic value of the final state reached by each random walk is also computed using some heuristic function. Once all n walks have been performed, the search *jumps* to the end of the walk whose final state, s, has the lowest heuristic value. This means that Arvand now runs a new set of n random walks of length m, only this time the walks originate from state s. This is followed by another jump to the end of the most promising walk from this new set of walks. This process repeats until either a goal state is encountered, or some number of jumps are made without any improvement in the heuristic values being seen. In the latter case, the current search episode is terminated and a new episode begins with random walks originating from s_i.

Arvand has been shown to be able to solve many difficult problems that traditional planners have been unable to solve (Nakhost and Müller 2009). This increase in coverage generally comes at the expense of solution quality, though the quality can be improved significantly by using the any-time and plan-improvement strategies described later in this paper. Arvand also requires very little memory which makes it ideal for running simultaneously in a shared-memory environment with other memory-intensive planners.

Configurations Four different Arvand configurations have been included in the ArvandHerd portfolio. Below, the parameters that differ between configurations in the portfolio are described in more detail. We omit any description of most of the other system parameters. For a more comprehensive discussion of all the parameters in Arvand, see (Nakhost and Müller 2009), (Nakhost, Hoffmann, and Müller 2010), (Nakhost et al. 2011).

Config	Bias Type	Initial Walk Length	Extending Rate
1	MDA	1	2.0
2	MDA	3	1.5
3	MHA	1	1.5
4	MHA	10	1.5

Table 1: `Arvand` configurations used in `ArvandHerd`.

The first important difference between configurations relates to the biasing of the random action selection. `Arvand` allows for random walks to either be unbiased, biased to avoid actions that have previously led to dead-ends (referred to as *MDA*), or biased to using *helpful actions* identified by the heuristic function (referred to as *MHA*). These different biasing strategies have been shown to be useful for different domains (Nakhost and Müller 2009). The bias used for each configuration in the portfolio is shown in Table 1.

The portfolio configurations also differ in parameters related to the random walk length. In `Arvand`, this length is adjusted online if little progress is being made in the heuristic values seen during a set of random walks. Such stagnation may occur if the current state is in a *heuristic plateau*. In an attempt to escape these plateaus, the walk length is increased over time. The initial walk length, the frequency with which walks are lengthened, and the factor by which they are lengthened (called the *extending rate*) are all parameters affecting this process. The initial walk length and the extending rate for each configuration in the portfolio is shown in Table 1. Note, the frequency with which walks were lengthened did not vary between configurations.

Heuristic Function All `Arvand` configurations use the FF heuristic (Hoffmann and Nebel 2001). For this heuristic, a possibly suboptimal plan starting at the current state is found to a relaxed version of the problem. This relaxation corresponds to the removal of delete effects from operators.

Techniques used for solving the relaxed problem vary. The implementation used for `Arvand` is from the `Fast Downward` planning system (Helmert 2006). In this implementation, the heuristic value ignores operator costs and is given by the number of operators in the relaxed plan. This heuristic will be referred to as the FF_{FD} heuristic.

Smart Restarts If `Arvand` makes a number of jumps without seeing any progress in the heuristic values encountered, the current search episode is terminated. However, instead of always restarting from scratch, as is done in the simplest version of `Arvand`, the planner can build upon progress made by previous search episodes through the use of a *walk pool* (Nakhost, Hoffmann, and Müller 2010). For some a (called the *pool size*), the walk pool holds the a "best" trajectories seen in all search episodes performed thus far. A trajectory t_1 is preferred over a trajectory t_2 if the state with the lowest heuristic value in t_1 is lower than the state with the lowest heuristic value in t_2. Qualifying trajectories are added to the walk pool at the termination of the corresponding search episode. For each new search episode, a trajectory t is randomly selected from the walk pool. Instead of starting from s_i, the new search episode then begins from a state that has been randomly selected from t.

Note, for the first b search episodes — where b is a parameter called the *pool activation level* — the search begins from the initial state. It is only after the first b episodes are completed that partial trajectories from the walk pool are used to find new starting positions for search. This prevents the walk pool from becoming completely biased towards trajectories that are all similar to the very first trajectory.

Configuration Selection as a Bandit Problem `Arvand` has been enhanced with a system that, given a set of configurations C, selects a configuration for the next search episode from C based on the performance of the configurations during previous search episodes. This system views configuration selection as a multi-armed bandit problem in which C is the set of bandits and the search episodes correspond to arm pulls. This paradigm requires the definition of a payoff function for search episodes. For this system, the reward given to a search episode e performed with configuration c is given as follows: where s is the state on the trajectory of e that achieved the lowest heuristic value, the reward given to c is $\max(0, 1 - h(s)/h(s_i))$, where $h(r)$ is the heuristic value of state r.

Using this reformulation of the problem, configurations can be selected online using any of the multi-armed bandit algorithms. In `Arvand` , the UCB algorithm (Auer, Cesa-Bianchi, and Fischer 2002) is used.

Any-time Planning with `Arvand` The solutions found by `Arvand` are generally suboptimal and so this planner does not terminate once a solution is found. Instead, the solution is added to the walk pool and a new search episode is started. The cost of the best solution found thus far is used as a bound on all future trajectories. This planner can then be run indefinitely or until some resource limit is reached.

The `LAMA` Planner

`LAMA` is a WA*-based planner that won the sequential satisficing track of IPC 2008 (Helmert, Do, and Refanidis 2008). It uses both multiple heuristic functions and helpful action open lists. Given a set of k heuristics $H = \{h_1, ..., h_k\}$, `LAMA` will have two sets of k open lists, denoted $O = \{o_1, ..., o_k\}$ and $O^p = \{o_1^p, ..., o_k^p\}$. `LAMA` must also be given a second set of heuristics, denoted $H^p = \{h'_1, ..., h'_j\}$, for the generation of helpful actions. Note, we will let $pref_{h'_i}(s)$ denote the set of children corresponding to the helpful actions found with heuristic h'_i for state s.

When it is time to expand a state, one of the open lists from either O or O^p is selected in a process described below. This open list will return the state s it identifies as the best state it contains. If s is a goal state, the solution is extracted from the closed list and returned. If s is not a goal state, the children of s, denoted C, are then generated, as is the set of *preferred children* of s, given by $C' = pref_{h'_1}(s) \cup ... \cup pref_{h'_j}(s)$. The states in C are then added to each of the lists in O for which states in any $o_i \in O$ are sorted by the cost function $f_i(s') = g(s') + w * h_i(r)$, where r is the parent of s' and w is the weight used for the current WA* search. For example, the cost given to $c \in C$ in open list o_i is given by

$g(c) + w * h_i(s)$. This technique is called *delayed heuristic evaluation* and has been shown to be effective in planning. The states in C' are then added to each of the lists in O^p. For every $o_i^p \in O^p$, states in o_i^p are ordered using the same cost function as o_i. However, notice that o_i and o_i^p do not contain the same states as o_i contains all generated but not expanded states, while o_i^p only contains preferred children.

When selecting which open list to remove a state from, the strategy in use is to alternate between all lists in $O \cup O^p$. The alternation is supplemented with a preferred children open list bonus. Whenever a state is seen such that for at least one of the heuristics it is the state with the lowest heuristic value seen so far, the open lists in O^p are all given a bonus of j state expansions. This means that the alternation will be restricted to only the open lists in O^p until each has expanded j nodes (or more if additional bonuses are accrued during this phase), at which point alternation will continue among all lists in $O \cup O^p$.

Heuristics Two heuristics were used in the version of the LAMA planner entered in IPC 2008: the landmark-count heuristic and a variation of the FF heuristic. These will be denoted as LM and FF^+, respectively. Both heuristics were also used for helpful action generation.

While the LM heuristic was one of the major advances introduced in the LAMA planner, the heuristic was used as is in ArvandHerd and so interested readers are referred to the journal paper on LAMA (Richter and Westphal 2010). However, instead of using FF^+, ArvandHerd uses two related heuristics. To explain why, we briefly describe FF^+.

Just as in FF_{FD}, LAMA's version of the FF heuristic computes a plan for the relaxed problem. This plan yields two obvious heuristics. The first, denoted by FF_{size}, is given by the number of actions in the relaxed plan just as is done in FF_{FD}. This heuristic is intended to capture the expected depth of the solution from the current state. The second, denoted by FF_{cost}, is given by the sum of the cost of the actions in the relaxed plan and is designed to capture the expected cost of the solution from the current state. FF^+ is given by the sum of FF_{size} and FF_{cost} as a way to balance between the two heuristics. Note, as Fast Downward and LAMA compute the relaxed plan differently, the values of FF_{FD} and FF_{size} are often different, as are the set of generated helpful actions.

In our experiments, we found that coverage was increased if, instead of using FF^+, we used both FF_{cost} and FF_{FD} as a way to balance between these metrics. This means that three heuristics are used in the version of LAMA used in ArvandHerd: LM, FF_{cost}, and FF_{FD}. However, only FF_{FD} and FF_{cost} were used to generate helpful actions.

Any-time Planning Once a solution is found with LAMA, the search is restarted from the initial state with a lower weight value. The previous best solution found is then used to prune all future searches. Changing the weight introduces *diversity* into the search which helps the planner avoid making the same early mistakes it has made previously. In the version of LAMA used in ArvandHerd, the first iteration runs greedy best-first search which means open lists are ordered by heuristic values alone. This is then followed by iterations with weights 10, 5, and 2, followed by 4 iterations with a weight of 1, and a final iteration with a weight of 0. A similar strategy has been shown to significantly outperform other forms of any-time planning (Richter, Thayer, and Ruml 2010). The WA* iterations have been further diversified effectively by randomizing the order in which generated children of the same parent are added to any one open list. This causes ties between children of the same state to be broken differently in different iterations.

The caching of heuristic values, helpful actions, and the best path found for each state in the closed list has also been shown to increase the speed of LAMA since many heuristic values will not need to be re-computed during future iterations (Richter, Thayer, and Ruml 2010). This feature was not part of LAMA as submitted to IPC 2008, but has been added to LAMA as used in ArvandHerd.

The ArvandHerd Architecture

For the sequential satisficing multi-core track of IPC 2011, 4 processors are allotted for each planner. As both Arvand and LAMA are built on top of Fast Downward, ArvandHerd is run from a single binary. When problem-solving begins, this binary spawns threads for different members of the portfolio. However, before this can begin, the planner first requires a *translation* from PDDL to a SAS+-like formalism, and a *knowledge compilation* step that builds data structures necessary for the LM heuristic. We have not parallelized these components and simply use this portion of the original LAMA code as is. For more information on this process, see the work on LAMA (Richter and Westphal 2010) or the work on Fast Downward (Helmert 2006) on which this process is based.

Once the translation and knowledge compilation stages are complete, one of the processors is assigned to run LAMA while the other three are each given one of the four Arvand configurations to run. Most of the communication between the processors is limited to those running Arvand. Specifically, the three processors share a walk pool and a single UCB configuration selection system. When a processor has completed a search episode, it submits the corresponding trajectory to the shared walk pool, and gets a new trajectory in return, or the empty trajectory if the activation level has not yet been reached. The processor then submits the reward for its current configuration to the UCB system and in return is given a configuration to use in its next search episode. This sharing of the UCB system among the processors running Arvand allows them to more quickly identify strong configurations than they would be able to with independent UCB systems. The walk pool, for which both the activation level and size are set to 100, is shared for similar reasons. LAMA is also given the ability to add walks to the solution pool, though in the submitted planner it only adds solution trajectories.

So as to maintain the correctness of the walk pool and the configuration learner, each system uses a lock that limits access to one processor at a time. As the search episodes dominate the Arvand execution time and LAMA is not expected to find solutions very often, there is little synchronization or contention overhead caused by sharing these resources.

The final shared value is the cost of the best solution found by any planning method thus far. This value is used to prune LAMA's WA* search.

Plan Improvement with Aras

While Arvand usually performs well in terms of coverage, it often finds low quality solutions. To address this issue, the Aras plan improvement system was created (Nakhost and Müller 2010). Aras involves two phases: *action elimination (AE)* and *plan neighbourhood graph search (PNGS)*. AE involves a scan of the current solution and the removal of unneccesary actions. For PNGS, a *plan neighbourhood graph* is built around the current solution using a breadth-first search. The plan neighbourhood graph is then searched for a shorter path between the start and any goal states.

The execution of Aras alternates between iterations of AE and PNGS until some time or memory limit is hit. However, instead of rebuilding the neighbourhood graph on each new PNGS iteration, the previous bread-first search is simply continued so as to grow the neighbourhood graph.

In ArvandHerd, whenever a solution is found by any processor, an instance of Aras is created and run on the current solution. If the initial solution was found by Arvand, Aras is given a 60 second time-limit. If the initial solution was found by LAMA, Aras is given a 40 second time-limit. This limit is lower for LAMA since that planner already has a fairly effective plan improvement scheme.

Recall that LAMA uses the cost of the best solution found by any method for pruning. Such pruning is ineffective for Arvand which instead only uses the best cost of a solution found strictly with Arvand as a bound. This is because bounds given by LAMA or Aras solutions are often too tight for Arvand in which case Arvand is unable to find any new solutions. As such, it was generally found to be more effective to create a diverse set of plans with Arvand and improve them with Aras, than to force Arvand to create low cost plans directly by using the global bound.

Memory Management

As the memory requirements of Arvand are limited to space for the current trajectory, the best random walk seen thus far, the walk pool, and the UCB configuration selection, Arvand is expected to almost never hit the 6 GB memory limit given to planners for IPC 2011. This is not the case for Aras and LAMA. As such, these processes need to be prevented from exhausting all the memory given to the planner, thereby crashing the whole system, and preventing further search by the processors running Arvand . To address this problem, the PNGS phase of each Aras instance is limited to using only 500 MB, and the total memory of the open and closed lists in LAMA is set as 2.7 GB. If the Aras limit is hit, Aras quits and returns the best solution found thus far. If the LAMA limit is hit, the current search iteration is ended and the open lists are emptied. The next iteration of LAMA then begins with the possibility that the diversity introduced by changing the weight and tie-breaking may avoid the mistakes made on the previous iterations. If the final 0-weight iteration also runs out of memory, the processor running LAMA will run another copy of Arvand instead.

Conclusion

We have described the main features of the ArvandHerd parallel planner which uses a portfolio containing the Arvand and LAMA planners. Due to the use of the porfolio, ArvandHerd is expected to have strong coverage, while the use of Aras and LAMA's any-time strategies should lead to good solution quality.

Acknowledgments

We would like to thank Sylvia Richter for allowing us to use the LAMA planner in the ArvandHerd portfolio, and Malte Helmert for giving us access to the Fast Downward code. We would also like to acknowledge the support of NSERC and Alberta Ingenuity.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47(2-3):235–256.

Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds. 2010. *Proceedings of the 29th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*. AAAI.

Helmert, M.; Do, M.; and Refanidis, I. 2008. IPC 2008 Deterministic Track.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)* 14:253–302.

Nakhost, H., and Müller, M. 2009. Monte-Carlo Exploration for Deterministic Planning. In Boutilier, C., ed., *IJCAI*, 1766–1771.

Nakhost, H., and Müller, M. 2010. Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement. In Brafman et al. (2010), 121–128.

Nakhost, H.; Müller, M.; Valenzano, R.; and Xie, F. 2011. Arvand: The Art of Random Walks. *IPC 2011 Deterministic Track Planner Reports*.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2010. Improving Local Search for Resource-Constrained Planning. Technical Report TR 10-02, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

Richter, S., and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research (JAIR)* 39:127–177.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The Joy of Forgetting: Faster Anytime Search via Restarting. In Brafman et al. (2010), 137–144.

Valenzano, R. A.; Sturtevant, N. R.; Schaeffer, J.; Buro, K.; and Kishimoto, A. 2010. Simultaneously Searching with Multiple Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agent Search Algorithms. In Brafman et al. (2010), 177–184.

AYALSOPLAN: Bitstate Pruning for State-Based Planning on Massively Parallel Compute Clusters

Juhan Ernits, Charles Gretton, Richard Dearden
School of Computer Science
University of Birmingham, UK
{j.ernits,c.gretton,r.w.dearden}@cs.bham.ac.uk

Abstract

Many planning systems operate by performing a heuristic forward search in the problem state space. In large problems that approach fails, exhausting a computer's memory due to the burden of storing problem states. Moreover, it is an open question exactly how that approach should be parallelized to take advantage of modern multiple-processor computers and the proliferation of massively parallel compute clusters. This extended abstract proposes an answer to this second question, while also going some way to addressing the memory problems.

We present AYALSOPLAN, our entry in the Multi-Core Track of the 2011 International Planning Competition (IPC-2011). Our approach is to run many independent and incomplete state-based searches in parallel. Our approach deliberately exploits hashing collisions to limit the set of states an individual search can encounter. Also, none of the parallel searches store all expanded states, each corresponding to a memory efficient state-based reachability procedure, albeit incomplete. As soon as a search determines reachability, the parallel processing ceases, and a single-core computer can efficiently construct the plan.

Because the 2011 IPC evaluation environment of the Sequential Multi-Core Track is *not* a massively parallel computer, and moreover because it imposes a *very limited timeout*, we have limited expectations regarding how AYALSOPLAN might be ranked in that evaluation. Therefore, this extended abstract commits some space to presenting empirical data we collected when evaluating our approach on our local cluster, without any runtime restrictions – i.e., searches can only fail when memory is exhausted. It is in that setting that we demonstrate the positive characteristics of our approach.

1. Introduction

Most of the fastest modern planning systems – including LAMA (Richter and Westphal 2010), the winner of the Sequential Satisficing track of the 2008 IPC – implement a best-first search of the state space. Those searches operate by maintaining an *open list* of states that have been visited but not completely expanded, and a *closed list* of states which have been visited and completely expanded. The storage burden associated with keeping track of visited states is a major hindrance to the scalability of modern systems. This is the case for both *frontier* variants of best-first search (Korf et al. 2005), and more classical implementations. Indeed, we are unaware of any planning system that can solve all IPC benchmark problems, including the big ones, given unlimited processing time and reasonable limitations on available memory.

Bitstate hashing (Courcoubetis et al. 1992; Holzmann 1998) is a memory-efficient technique for keeping track of visited states in state-based searches. The technique tracks the visitation status of a state using a single element, usually a single bit, in an integer indexed array. When a state s is visited during search, the evaluation of a hash function at s maps that state to an index in the array. That s has been visited is recorded by the status of the indexed bit.[1] Bitstate hashing thus trades storage required to record visited states against the probability of *collisions*, which occur when two different states are indexed to the same array entry and therefore cannot be distinguished. Bitstate hashing in a planning context has been explored previously in (Edelkamp 2002; Edelkamp and Jabbar 2005). In this abstract we introduce the related technique of *bitstate pruning*, initially proposed for model checking (Ernits et al. 2006; Ernits 2005), to the planning setting. Bitstate pruning deliberately exploits the collisions of bitstate hashing to dynamically limit the set of states the search can encounter. Bitstate pruning can reduce the memory requirements of search at the expense of completeness. Also in a model checking setting, more-recently a technique that utilises deliberately undersized bitstate hash tables was proposed to alleviate the processing (resp. storage) burden of computing the heuristic value of states (resp. keeping track of states). In detail, Kupferschmid et al. (2006) proposed treating states that hash to the same entry of the undersized array to be of equal heuristic value.

Bitstate pruning as an approach can be applied in any planner that uses state space search to prove goal reachability – constructively or otherwise – while maintaining a collection of visited states. Moreover, the incompleteness of the resulting search can be mitigated by running multiple searches in parallel, each using a different array size. Our competition entry, AYALSOPLAN, is based on an implementation of bitstate pruning in LAMA. In order to help us demonstrate the effectiveness of bitstate pruning

[1] Some variants use multiple bits in multiple arrays according to a set of state hashing functions.

in planning, we have also implemented it in the straightforward satisficing state-based search of AYPLAN (Robinson, Gretton, and Pham 2008). In the remainder of this abstract, when addressing a specific implementation, we write AYALSOPLAN^{lm} if the base search procedure is LAMA, and AYALSOPLAN^{ay} for the AYPLAN implementation.

When exploiting the parallelisation that bitstate pruning makes possible on a cluster of computers, both AYALSOPLAN^{ay} and AYALSOPLAN^{lm}are parallel satisficing planners that consistently solves larger problems than the procedures of their respective base systems, AYPLAN and LAMA. Here, it is important we clarify that our evaluation does not impose a timeout, and therefore when we speak of scalability, we do so where planning failure occurs solely due to memory exhaustion – i.e., given all the time in the world, the parallel incomplete searches of AYALSOPLAN^{ay} can solve larger problems than the serial satisficing procedure of the base planner AYPLAN. Moreover, we also find that the parallel incomplete searches of AYALSOPLAN^{ay} can often solve larger and more difficult problems than LAMA with a 2GB memory limit and unlimited time. In this work we have not exhaustively evaluated AYALSOPLAN^{lm} because the many heuristics and search optimisations employed in LAMA obscure the results, and pose a massive burden on our limited cluster resources, if we have to evaluate all the varieties of LAMA. Finally, we believe that our evaluation using AYALSOPLAN^{ay} already makes a clear empirical case for bitstate pruning for planning on massively parallel architectures.

This extended abstract is organised as follows: In the next section we provide a brief introduction to the use of search in planning algorithms. In Section 3, we then present the bitstate pruning approach and show how the probability of a search reaching a state can change as the size of the hashtable changes. In Section 4, we discuss the specifics of how we have implemented bitstate pruning in a number of planning systems, and present an empirical evaluation of one of those systems in Section 5. In Section 6, we summarise our contribution, making some concluding remarks.

2. Best-First Forward Search

In the early 90s implementations of best-first search exhausted "the available memory on most machines in a matter of minutes" (Korf 1992). Nowadays it continues to be considered a memory intensive approach (Korf et al. 2005), nonetheless the approach underlies a majority of good planning systems. Indeed, the dominant satisficing planning systems are based exactly on a best-first search of the state space, the more successful approaches typically employing a variant of A^* (Hart, Nilsson, and Raphael 1972). This is evidenced by the successes at recent IPCs of such systems. These include LAMA, SGPLAN versions 4 and 5 (Hsu et al. 2006; Chen, Hsu, and Wah 2004),[2] FASTDOWNWARD (Helmert 2006), FASTFORWARD (Hoffmann and Nebel 2001), and HSP (Bonet and Geffner 2001). Overall, LAMA, the base procedure for our competition entry, is one of the more recent and scalable procedures in this vein.

Although the recent success of best-first search in a planning setting might partly be attributed to the relatively vast quantities of memory available on modern computers, it can mostly be attributed to three important developments. First, that of heuristics for the satisficing case, such as the FF-heuristic $h^{\mathtt{ff}}$ (Hoffmann and Nebel 2001), the causal-graph heuristic $h^{\mathtt{cg}}$ (Helmert 2006), and the landmark-counting pseudo heuristic $h^{\mathtt{lm}}$ (Richter and Westphal 2010);[3] second, the development of planning-specific preprocessing algorithms, such as relevance analysis methods (e.g., (Haslum and Jonsson 2000; Bacchus and Teh 1998)) and plangraph analysis (Blum and Furst 1997), and representational optimisations, such as compilations to *multi-valued* setting and the related hierarchical decompositions of planning tasks (Helmert 2006); third, search tricks, such as *deferred(/lazy) heuristic evaluation* and *preferred operators* that have been shown to drastically improve the efficiency of some heuristic search techniques in many planning benchmarks (Richter and Helmert 2009).

We provide a brief sketch of state-based best-first forward search as it typically appears in the discussed planning procedures, since we will use the terms later. The search can be described in terms of a bound ranking function from states to numbers $f : \mathcal{S} \to \mathbb{N}$, two container data-structures *open* and *closed*, and a search graph. Here, the evaluation of the ranking function at a state, $f(s)$, maps each state to a numeric value, thereby ranking states. Structure *open* contains states encountered by the search, forward from the starting-state s_0, whose successors have not all been evaluated – i.e, there are actions whose effects on states in *open* have not been evaluated. The *closed* structure contains states that were previously in *open*, and for which all successor states have been evaluated. Typically we say that states in *closed* have been "expanded", and that states in *open* are "unexpanded" (or in some searches "partially expanded"). The search graph has one vertex for each state occurring in either *open* or *closed*, and a directed edge (s,s') labelled with actions whose expansion at s induced a state transition to s'. Here, we use the notation s to refer both to the state $s \in \mathcal{S}$ and its corresponding vertex.

At the commencement of search *open* is a singleton containing s_0 and *closed* is empty. The search proceeds interleaving the selection of a state from *open* to expand and its expansion. This process executes until a goal state is reached during an expansion, or otherwise until all promising states have been expanded – e.g., in the case that the goal is not reachable according to the search constraints. State selection is done greedily according to the given evaluation function $f(s) = c(s) + \beta h(s)$. Here, $c(s)$ is the (sometimes approximated) length of the shortest path from s_0 to s, while $h(s)$ estimates the value of expanding s. The factor β, usually 1, determines how greedy the search is with respect to h.

[2] For the underlying search procedure SGPLAN uses: *metric*-FASTFORWARD, MCDC (a variant of *Metric*-FASTFORWARD), and LPG. Only the first two can be characterised as a best-first forward search. LPG is a local search procedure for planning inspired by the Boolean SAT(isfiability) procedure WALKSAT.

[3] Nowadays these heuristics are often used in combination in a so-called *multi-heuristic* setting (Helmert 2006).

Once a state s is selected it undergoes expansion. For one or more actions available at s, the search evaluates the successor states of s, adds them to *open* if they are not already contained in either *open* or *closed*, and updates the search graph to reflect node additions and/or altered connectivity. If this step exhausts the action possibilities at s, then s is moved from *open* to *closed*. If h is inadmissible, when expanding a state it might be that a better, less costly path to a *visited* successor s is discovered, and therefore that a better approximation of $c(s)$ is discovered. Many searches propagate that better estimate through the search graph. For example, in order to emphasise a laziness in computation, in some descriptions a state is said to be *reopened* – e.g. see (Hansen and Zhou 2007) – in the sense that when a better approximation of $c(s)$ is found, s is added again to *open*, and the c-value change starts to propagate when the reopened state s is again chosen from *open* for expansion.

During the expansion of a state the search establishes whether or not evaluated successor states are new. The details of how this is achieved are very important to our contribution. In practice, all encountered states (i.e., elements in *open* and *closed*) are stored in a sorted associative container,[4] or more commonly a hash table. When a successor state is considered during expansion, its membership in that container is tested to decide if a new state should be added to *open*, and how the details of the search graph should be altered. In many planning benchmarks solvers fail on large problem instances because storage of encountered states exhausts system memory.

Nowadays it remains an open question how best to exploit modern distributed computing environments to achieve better scalability and efficiency in state-based searches. In particular, how best to trade-off available processing and memory resources, avoiding exhaustion of system memory before search yields a solution.

3. Bitstate Pruning

Let us consider a best-first search that utilises bitstate hashing to distinguish between new and already visited states. At the beginning of search an array of bits H is initialised to contain zeros. Whenever a state s is added to *open* a bit in the hash array at the address of $\texttt{hh}(s)$ is set, $H[\texttt{hh}(s)] := 1$. Here, let hh be a state hashing function that satisfies the standard *uniform hashing assumption* for analysis purposes. When expanding the actions of a state s in *open*, a successor s' is added to *open* iff $H[\texttt{hh}(s')] \neq 1$.

A hash collision occurs when several states hash into the same address in H. In the limit as $|H|$, the size of the hash array, goes to infinity, we can invoke a special hh that hashes each distinct state to a distinct bit in that array. However, since in practice we are constrained by available memory resources, $|H|$ is small and hash collisions occur. In previous applications of bitstate hashing several measures are taken to reduce such effects. For example, in Bloom filters (Bloom 1970), each state uses several bits in the hash table, and the addresses for a state are calculated by multiple hash functions. That approach decreases the probability of collisions if the hash table contains a small number of entries. Other analyses of bitstate hashing (Holzmann 1998; Dillinger and Manolios 2004; Kuntz and Lampka 2004) have also been concerned with reducing the collision/omission probability. (Holzmann 1998) proposes a sequential multihash principle that performs hashing repeatedly using independent hash functions. The overall effect is to reduce the probability of collisions occurring at the same places and thus avoiding omissions.

In our setting hash collisions are valuable, as we use them to reduce the number of states explored by a given instance of best-first search. In more detail, suppose the number of reachable states n in the search graph is much larger than $|H|$. Then the probability of collisions is 1 and states are dropped by a search. That problem can be mitigated by using sequential multihashing, but rather than reducing the state drop-rate to increase the exhaustiveness of a search, we use a sequential multihash idea to increase the probability of reaching a goal state early in multiple independent searches. We call the overall approach *bitstate pruning*.

To increase the probability of finding a goal according to bitstate pruning, it is necessary to either repeat the search with a different hash function, change the search policy,[5] or change the size of H. As such repetitions are independent and involve no communication, they can be performed in parallel. Essentially, we can leverage an abundance of independent processing units to quickly (wall time) find a good hash function and the corresponding plan.

In our approach $\texttt{hh}(s)$ is a combination of two things: a hash function that takes the bitvector of a state as input and produces a hash value of some size, typically 32 or 64 bits, and the use of the modulo function (mod) to calculate the address of the bit in H. Thus, changing the size of $|H|$ provides an easy way of changing the hash function.

A desirable side effect of bitstate pruning is that it imposes a limit to the *open* data structure: there can never be more states in *open* than there are bits in H. The exhaustion of memory due to a large number of states in *open* is one of the reasons why search with very large bitstate hash tables fails. Thus, by pruning the search graph slightly differently under each instantiation of bitstate pruning, we trade memory for CPU, admittedly with some repeated exploration of problem states.

Example

The Three Integer Problem has states consisting of the values of three integer variables and has three actions a_1, a_2 and a_3 that each increment one of those variables. The search graph for this problem is shown in Figure 1 expanded up to search depth 2.

We index states s_i in a sequence that corresponds to depth-first exploration of the state space.

Let us assume that we start exploring the state graph in Figure 1 using depth-first search, a depth limit of 2 actions,

[4] This is the case in LAMA.

[5] In iterative deepending, we can change the search depth, and in A^* we can alter β, h, and even alter how c is (approximately) evaluated.

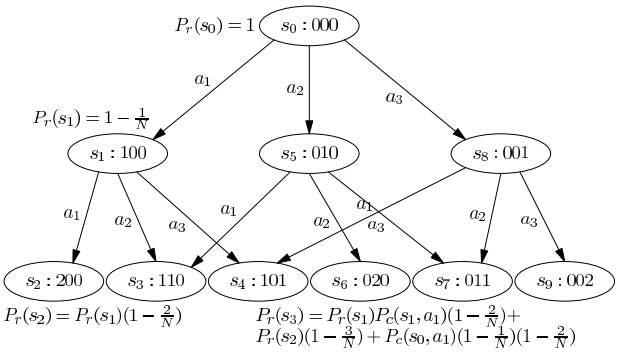

Figure 1: Search graph for the three integer problem. $P_r(s_i)$ denotes the probability of reaching state s_i, and $P_c(s_i, a_i)$ denotes the probability of hash collision when executing action a_i at state s_i. N is the size of the bitstate hash table.

and a bitstate hash table of 2 bits. The probability of reaching the initial state, written $P_r(s_0)$, is 1. The probability of reaching s_1 is $P_r(s_0)(1 - 1/|H|) = 1/2$. When the size of the bitstate hash table is only 2, s_2 is not reachable, because to reach s_2, s_1 must be reached and then the hash table is full. As a result all actions from s_1 yield a collision. Although s_2 is not reachable, it is possible to reach s_5 with probability $P_r(s_0)P_c(s_0, a_1)(1 - 1/|H|) = 1/4$ – i.e. $P_c(s_0, a_1)$ us the probability of a collision when expanding a_1 at s_0, therefore overall the expression gives the probability of reaching s_0 times the probability of action a_1 yielding a collision, times the probability of a collision not occuring in s_5. Using a similar reasoning it is possible to reach s_8 with the probability of $P_r(s_0)P_c(s_0, a_1)P_c(s_0, a_2)(1 - 1/2) = 1/8$. It should be noted that in the given example there is also a $1 - 1/2 - 1/4 - 1/8 = 1/8$ probability that neither s_1, s_5 or s_8 is reached with a bitstate hash table size 2. With the bitstate hash table size 3 the probability of reaching any node in the search graph in Figure 1 using a depth-first search policy becomes nonzero.

4. Our Approach

We develop a satisficing planning approach, AYALSOPLAN, that can leverage the processing resources of cluster computing environments to obtain better scalability according to the bitstate pruning scheme just described. Our competition entry is based on AYALSOPLAN^{lm}, an implementation of this approach using LAMA, and our experimental evaluation is predominantly based on an implementation using AYPLAN. AYALSOPLAN uses multiple searches, each of which implement bitstate pruning for a distinct array size in what is otherwise a *frontier search* procedure (Korf et al. 2005). In order to construct a plan AYALSOPLAN operates in two phases: the first performs parallel plan existence searches, then if a plan exists, the second constructs a plan by repeating a successful incomplete search on a single core, this time storing all the encountered states. It is worth noting that because the frontier searches are independent, they can be run in parallel, in sequence, or a combination thereof. Therefore, in practice one can use several different cluster resources for planning.

Each executed search in the first phase corresponds to a *frontier search* (FS), a best-first search that uses only a fraction of the memory used by ordinary best-first searches of a state space. In detail, FS deletes states designated to *closed*, implicitly removing these from the search graph. Consequently FS is a sound approach to obtaining a proof of plan existence,however it is not constructive, because there is no data from which to extract a plan directly once a goal state is reached. Existing varieties of FS-based systems are rendered constructive by using them according to a divide-and-conquer query strategy, or otherwise by keeping *closed* on a secondary (slow) storage device. In this respect AYALSOPLAN differs from existing FS variants. AYALSOPLAN *deletes* states designated for *closed*, but also, according to bitstate pruning, an instance of search forbids multiple states which hash to the same entry of a bitarray from being considered. This implies two important consequences beyond the scalability obtained by exploiting bitstate pruning in multiple parallel instantiations of AYALSOPLAN. First, as with any FS variant, AYALSOPLAN uses relatively little memory when performing plan existence proofs. Second, a plan can be extracted relatively quickly during our post-processing phase using relatively little memory, by using the $|H|$ and search depth limit from a successful bitstate pruning FS search.

Overall, our contribution is in a similar vein to systems such as HDA* (Kishimoto, Fukunaga, and Botea 2009), which also tackle the A^* memory consumption problem in a parallel setting. HDA*-like systems use a more-or-less brute-force approach, distributing state storage over a cluster of multiple independent networked machines. A key advantage of our approach is that we require no inter-process communication. Also, because we perform a frontier search, we have diminish memory requirements, and improved scalability with the number of processors.[6]

5. Experimental Results

To demonstrate the efficacy of bitstate pruning, we compare AYALSOPLAN^{ay} with AYPLAN, a straightforward implementation of best-first search that stores all the visited states in *open* and *closed* explicitly. In this evaluation we also include the performance results of a August 2010 version of LAMA using the IPC-6 run script (without WA* iteration) as a reference. That system represents a state-of-the-art domain independent system for most of the benchmark problems we have considered. Finally, we also present preliminary experimental results using AYALSOPLAN^{lm}.

It remains to discuss a few planning specific details of AYALSOPLAN^{ay} that reduce the burden on search. Based on AYPLAN, AYALSOPLAN^{ay} incorporates a preprocessing phase that employs a number of computationally cheap problem analysis techniques. In particular, following (Haslum and Jonsson 2000), when grounding domain operators we omit from consideration actions whose precondi-

[6]Clarifying, this comment is not to be interpreted in the language of "speedup factors", rather, it is a comment about being able to solve larger problems given many CPU-cores, each with limited memory and unlimited time in which to solve a problem.

Table 1: Number of problems solved by of AYPLAN, AYALSOPLAN^{ay}, and LAMA. For some problems AYALSOPLAN^{ay} proves plan existence, however a plan cannot be extracted in phase-2 given the 1GB memory limit. In that case we report two figures: (a) outside parenthesis, the number of problems for which a plan could be extracted, and (b) in parenthesis, the number of problems for which the existence problem was solved.

Domain	AYPLAN	AYALSOPLAN^{ay}	LAMA
transport	9	30	30
pipes-tankage	23	43(44)	39
elevators	16	30	26
peg solitaire	30	30	30
scanalyzer	27	27	30
openstacks	14	17(18)	30
sokoban	12	15 (25)	25

tions are statically false. We further reduce the size of the set of ground operators and state propositions by performing *static relevance* testing as described in (Bacchus and Teh 1998). Although recent versions of AYPLAN implement a number of useful planning heuristics, in AYALSOPLAN^{ay} and AYPLAN we rank states in *open* according to how many goal propositions they satisfy. In many cases we find that other heuristics can be detrimental to performance of bitstate pruning on massively parallel machines in the benchmarks we have considered – This usually occurs because a heuristic encourages many of the parallel searches to be uniform, therefore the processing resources are not exploited for coverage. In Figure 4 this problem is indicated for the case of PIPES-TANKAGE P23, where as the bitarray becomes large the probability of AYALSOPLAN^{lm} searches failing increases.

Our evaluation compares the planners on several domains: The IPC 2004 PIPES-TANKAGE domain that poses an NP-Hard satisficing problem (Helmert 2006), and several of the IPC 2008 domains. In evaluation all AYPLAN-based processes were limited to use maximum 1GB of memory. LAMA was run on a single core of a computer with Intel quad core CPU Q9650 and 4 GB of RAM. AYPLAN-based processes were run on a compute cluster with 20 quad-core Xeon E5345 CPUs totalling 80 CPU cores with 1GB memory per core. In no case do we impose any time limit. The default search depth for AYALSOPLAN^{ay} in all cases was limited to the minimum of 100000 and $|H|$.

Table 1 gives a summary of the results across the domains. Each row shows the number of problems solved (i.e. satisficing plans extracted) in the domain by each planner. The numbers in parenthesis in the AYALSOPLAN^{ay} column indicate the number of plan existences found as in some cases the plan extraction step exceeded the 1GB memory limit. Figure 2 summarizes the hash table sizes that were required to find plans. In many of the domains the $|H|$ values were in the order of tens or hundreds of thousands. In those cases, it is even feasible to AYALSOPLAN^{ay} serially on a single core. Figure 3 reports plan costs that were obtained by AYPLAN, AYALSOPLAN^{ay} and LAMA. Here, we are reporting the costs of the first solution found by each planner.[7] In the PIPES-TANKAGE case the plan cost is equivalent to plan length. For that domain AYALSOPLAN^{ay} usually produces better initial plans than AYPLAN. In the PEG SOLITAIRE case, the plan qualities of first plans obtained are also of good quality. We should not that it would be a simple matter to implement a cost cutoff for AYALSOPLAN^{ay} in a manner similar to our maximum depth cutoff.

The results in Table 1 show that in all cases AYALSOPLAN^{ay} made it possible to solve more problems than AYPLAN, and in some cases, like PIPES-TANKAGE and ELEVATORS even more than LAMA. In SOKOBAN AYALSOPLAN^{ay} was able to detect plan existence in more cases than AYPLAN and LAMA, but we were only able to extract plans for 15 problem instances during phase-2 processing. In the SCANALYZER domain AYALSOPLAN^{ay} and AYPLAN found solutions to all problems that could get through AYPLAN preprocessing.

A selection of the results (easier problems solved by all planners are omitted) for PIPES-TANKAGE domain are given in more detail in Table 2. As we described above, AYALSOPLAN^{ay} uses frontier search to further decrease its memory requirements. On termination a plan is constructed using a second phase, that performs an ordinary search parametrised by the array size that yields a solution. For results presented in Table 2, the time and memory recorded for AYALSOPLAN^{ay} represent the time (memory) to run AYALSOPLAN^{ay} in FS mode to prove the existence of a plan, in addition to the time (memory) required to extract a plan in the second phase – by running a plan extraction instance of AYALSOPLAN^{ay} with the size of the hash table discovered by a successful instance of FS search. Space limitations mean that we cannot include results from all the domains we tested in Table 2, however we present PIPES-TANKAGE to highlight some of the interesting findings. The first four and last two columns show respectively the performance of AYPLAN, AYALSOPLAN^{ay} and LAMA in terms of time and memory requirements. A dash in the memory requirements column indicates that after that amount of time the planner ran out of memory without producing a plan. Thus, AYPLAN failed on ten of the 14 problems in the table, LAMA failed on three, and AYALSOPLAN^{ay} solved all of them. The three columns following AYALSOPLAN^{ay} show the performance of the plan existence (frontier search) part of AYALSOPLAN^{ay}. The hashtable size column shows the bit length of the hash array required to find a plan, the memory column is the total memory required by AYALSOPLAN^{ay} to detect plan existence on this size array, and the time column shows the average runtime for runs of AYALSOPLAN^{ay} with array sizes close to the presented bitstate hash table size that led to a plan. The reason we choose this measure is that the runtime is typically much faster when a plan is found than when one is not found, so these times represent the time for an exhausive search given a hashtable of the largest size that needs to be evaluated. A reasonable approximation to

[7] None of the costs are guaranteed to be optimal.

Table 2: Memory and time requirements for finding satisficing solutions for some of the problems in the PIPES-TANKAGE domain using AYPLAN, AYALSOPLAN^{ay} and LAMA. AYALSOPLAN^{lm}could find solutions to problems P23, P26 and P28.

Problem	AYPLAN		AYALSOPLAN		AYALSOPLAN existence			LAMA	
	time (s)	memory (MB)	time (s)	memory (MB)	time (s/iter)	memory (MB)	hashtable size (bits)	time (s)	memory (MB)
P16_NET2_B14_G6_T80	1747	-	10.78	89	2.69	81	31441	6	32
P17_NET2_B16_G5_T20	26.4	191	2.44	38	.1	38	3066	5	20
P18_NET2_B16_G7_T60	786	-	5.25	78	.1	78	1356	3.24	23
P19_NET2_B18_G6_T60	23.65	165	7.06	113	.17	112	2169	5.70	14
P20_NET2_B18_G8_T90	1698	-	14.49	166	.13	166	4843	12.60	26
P21_NET3_B12_G2_T60	596	-	2.74	42	0.03	42	832	2.86	6
P22_NET3_B12_G4_T60	15.2	94	3.59	42	0.323	42	5607	4.2	23
P23_NET3_B14_G3_T60	1451	-	9.69	115	1.16	115	9410	800	-
P24_NET3_B14_G5_T60	1451	-	13.0	115	2.54	115	12711	9.0	43
P25_NET3_B16_G5_T60	1717	-	407	574	151	326	510642	46.7	77
P26_NET3_B16_G7_T70	1928	-	191	506	97.9	308	384214	2280	-
P27_NET3_B18_G6_T70	743	-	112	301	81.4	233	223881	11.6	82
P28_NET3_B18_G7_T70	739	-	371	879	184	505	904270	1150	-
P29_NET3_B20_G6_T70	1221	-	1065	1560	517	827	1428472	22.1	117

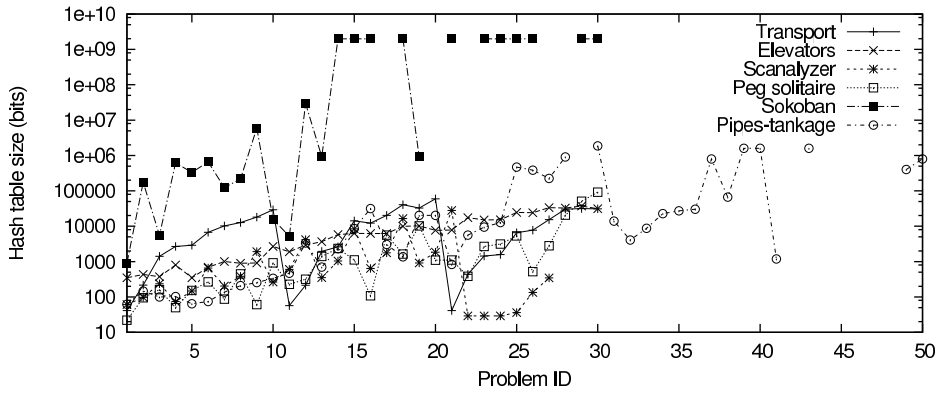

Figure 2: Hash table sizes in bits required by AYALSOPLAN^{ay} to find a plan in the corresponding domain.

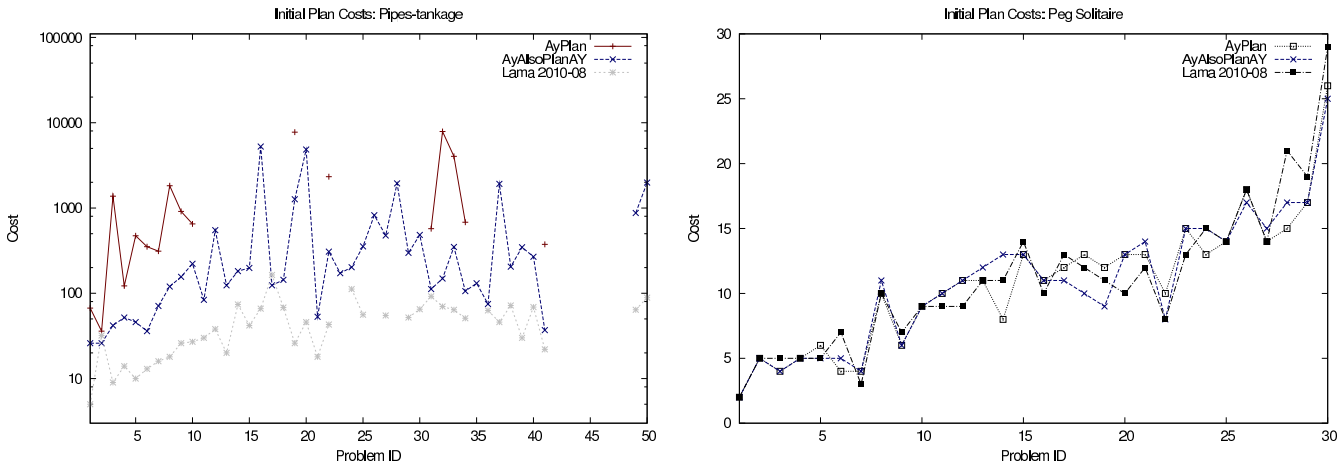

Figure 3: Initial plan costs of plans found by AYPLAN, AYALSOPLAN^{ay}, and LAMA.

the total time required to find a plan (distributed over all the parallel processors) is the time per iteration multiplied by the hashtable size and divided by two and divided by the number of processors we have available. In addition, it is easy to see that the smaller the $|H|$ the faster the search.

Though very preliminary, in Figure 4 we explore the behaviour of AYALSOPLAN^{lm} in PIPES-TANKAGE problem P23, a problem that LAMA is not able to solve and that AYALSOPLAN^{lm} solves very quickly – in the best case in 8 seconds while using 20 MB of memory at $|H| = 6834$. Not surprisingly, on the lower graph, we show that the maximum memory consumed by search instances grows linearly with the bitarray size $|H|$. The data also demonstrates the difficulty of predicting the time a search at some $|H|$ will take. Most importantly, from the data-points we have, it is clear that LAMA's search guidance has a negative impact in this problem instance, stopping goal states from being discovered at larger $|H|$ values.

6. Final Remarks

We describe an approach to state-based planning in massively parallel systems that corresponds to an application of bitstate pruning in domain independent satisficing planners. Our approach instantiates multiple independent and incomplete searches in parallel on separate CPU-cores, each of which has limited memory resources. Each individual search can be characterised as an *incomplete* variant of the *one bit per state* approach described in (Korf et al. 2005). Given sufficient processors, in the limit as the number of processes goes to infinity the overall search is sound and exhaustive, one of the searches eventually finding a goal state (if reachable).

We have implemented our approach using both LAMA and AYPLAN as base planners, the former corresponding to our entry, AYALSOPLAN, at IPC-2011. In this extended abstract we have performed an empirical evaluation of AYALSOPLAN^{ay}, our implementation of bitstate pruning using AYPLAN. That evaluation is over several important IPC benchmarks, and demonstrates that given an abundance of processor resources each with limited memory resources, the technique of bitstate pruning can outperform the same planner without it by a significant margin where planning failures only occur due to memory exhaustion (resp. a timeout). The relative memory efficiency of AYALSOPLAN allows it to solve very large planning problems, some of which cannot be solved by good serial systems, such as LAMA.

AYALSOPLAN can be used in the same way as AYPLAN and LAMA for iterative plan refinement after the first plan for a problem has been discovered. Indeed, our competition entry AYALSOPLAN^{lm} uses the entire evaluation period to iteratively improve the plan prescribed in finality. Our experiments suggest that the costs of plans produced by AYALSOPLAN^{ay} are not significantly worse than those produced AYPLAN, in fact, in the PIPES-TANKAGE domain the initial plans by AYALSOPLAN were better.

Finally, it is worth noting that when LAMA's search guidance is very useful, the performance of AYALSOPLAN^{lm} can be worse than that of the base planner. Therefore, our entry in the Multi-Core Track runs the August 2010 of LAMA in one thread.

References

Bacchus, F., and Teh, Y. W. 1998. Making forward chaining relevant. In Simmons, R. G.; Veloso, M. M.; and Smith, S., eds., *AIPS*, 54–61. AAAI.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13(7):422–426.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Chen, Y. X.; Hsu, C. W.; and Wah, B. W. 2004. SGPlan: Subgoal partitioning and resolution in planning. In Proc. *IPC4*.

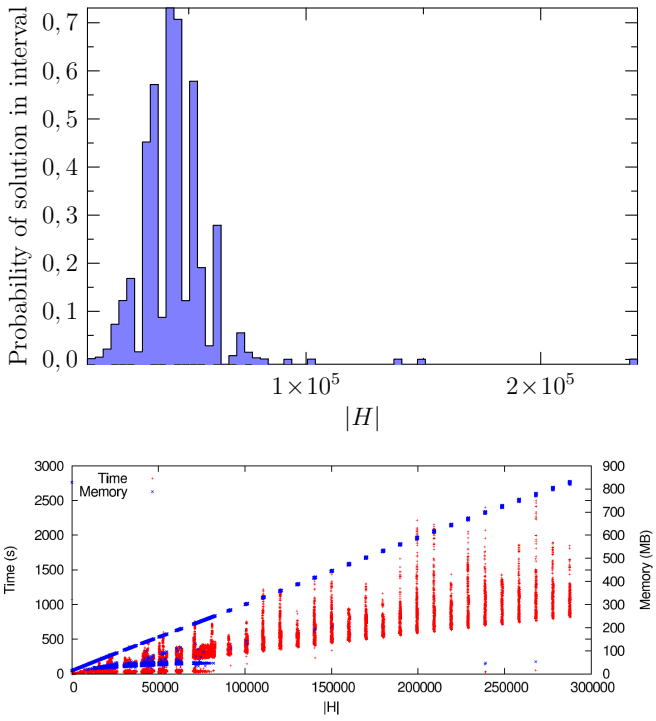

Figure 4: Above: Probability of finding a plan at a given hash table size $|H|$ in P23_NET3_B14_G3_T60 by AYALSOPLAN^{lm}. Below: Time and memory consumption for either finding a plan at a $|H|$ value or for exhausting the reachable state space in problem P23_NET3_B14_G3_T60 by AYALSOPLAN^{lm}.

Courcoubetis, C.; Vardi, M. Y.; Wolper, P.; and Yannakakis, M. 1992. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design* 1(2/3):275–288.

Dillinger, P. C., and Manolios, P. 2004. Bloom filters in probabilistic verification. In Hu, A. J., and Martin, A. K., eds., *FMCAD*, volume 3312 of *Lecture Notes in Computer Science*, 367–381. Springer.

Edelkamp, S., and Jabbar, S. 2005. Accelerating external search with bitstate hashing. In *KI05 (German Conference on AI) 19. Workshop on New Results in Planning, Scheduling and Design*, 7pp.

Edelkamp, S. 2002. Memory limitations in artificial intelligence. In Meyer, U.; Sanders, P.; and Sibeyn, J. F., eds., *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, 233–250. Springer.

Ernits, J.-P.; Kull, A.; Raiend, K.; and Vain, J. 2006. Generating tests from EFSM models using guided model checking and iterated search refinement. In Havelund, K.; Núñez, M.; Rosu, G.; and Wolff, B., eds., *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 85–99.

Ernits, J. 2005. Memory arbiter synthesis and verification for a radar memory interface card. *Nordic Journal of Computing* 12(2):68–88.

Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds. 2009. *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *J. Artif. Intell. Res. (JAIR)* 28:267–297.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1972. *Correction* to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.* (37):28–29.

Haslum, P., and Jonsson, P. 2000. Planning with reduced operator sets. In *In AIPS*, 150–158. AAAI Press.

Helmert, M. 2006. New complexity results for classical planning benchmarks. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *ICAPS*, 52–62. AAAI.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *J. Artif. Int. Res.* 14(1):253–302.

Holzmann, G. J. 1998. An analysis of bitstate hashing. *Form. Methods Syst. Des.* 13(3):289–307.

Hsu, C. W.; Wah, B. W.; Huang, R.; and Chen, Y. X. 2006. New features in SGPlan for handling preferences and constraints in pddl3.0. In Proc. *IPC5*.

Kishimoto, A.; Fukunaga, A. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In Gerevini et al. (2009).

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

Korf, R. E. 1992. Linear-space best-first search: Summary of results. In *AAAI*, 533–538.

Kuntz, M., and Lampka, K. 2004. Probabilistic methods in state space analysis. In Baier, C.; Haverkort, B. R.; Hermanns, H.; Katoen, J.-P.; and Siegle, M., eds., *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*, 339–383. Springer.

Kupferschmid, S.; Hoffmann, J.; Dierks, H.; and Behrmann, G. 2006. Adapting an AI planning heuristic for directed model checking. In Valmari, A., ed., *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 35–52.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In Gerevini et al. (2009).

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)* 39:127–177.

Robinson, N.; Gretton, C.; and Pham, D.-N. 2008. Co-plan: Combining SAT-based planning with forward-search. In Proc. *IPC6*.

Parallel Heuristic Search Fast Foward

Moisés Martínez Muñoz
Department of Computer Science, University Carlos III de Madrid,
Avenida de la universidad 30, 28911 Leganés, Spain

Abstract

Parallel computing involves the use of several processes working together on a single set of code as the same time. It is shown that this offer some advantages that can be applied in the currently planners with a minimal numbers of changes. In this paper we presented an experimental planner called PHSFF (Parallel Heuristic Search Fast Forward), which concurrent programming has been applied in the heuristic search process for the algorithm Enforce Hill Climbing (EHC).

Introduction

In recent years, parallel computing is a resource used to optimize algorithms or solve problems that had not been solved with traditional methods. In many cases the use of this type of programming involves the modification partial or total code of algorithms, although in some cases it is possible to use the sequential algorithms by performing a set of small changes to obtain a high yield.

Parallel computing is the concurrent use of multiple processors or cores to do computational work. In sequential programming, a single processor executes the program instructions in a step-by-step manner. Some operations, however, have multiple steps that do not have dependencies with each others and can therefore be splited into multiple tasks to be executed simultaneously. For instance, adding a number to all the elements of a matrix does not require that the result obtained from summing one element be acquired before summing the others elements. Elements in the matrix can be made available to several processors and the sums performed simultaneously, with the results available much more quickly than if the operations had all been performed serially.

To applicate parallelization in an algorithm, it is necessary find a task t, that can be divided in n (this is usually the number of cores o CPUs) independent tasks $t_0.....t_{(n-1)}$. In this case each task can be executed in different cores, but in some cases may appear dependencies among tasks (shared variables). These dependencies are resolved through critical sections, where threads access shared resources.

There currently exists different libraries, extensions for languages and systems for specific parallel computers, as the MPI library (MPI standar library, 1997) provides the send and receive operations needed for a CSP-like message passing model. OpenMP (OpenMp standar library, 1997) offers extensions to different sequential languages that can follow a particular model of parallel computation, reflects a PRAM-like shared memory.

Planning is a good environment to obtain benefits using the new processors with multiple cores with shared-memory. This is possible because the algorithms and techniques that are used in automated planning can be very easily adapted to use the advantages of this type of programming.

Related Works

In the last years, several approaches to parallel search and planning has been proposed. Parallel Retracting A* (PRA*) (Evett et al, 1995) which is based on A* (Nilson et al, 1968), simultaneously tackles the problem of duplicate state and state distribution among the threads. In this algorithm each thread maintains open and closed lists and a hash function distributes the states to the corresponding processor. The main advantage is that state duplicate detection but produces a significant overhead in the state distribution process. Transposition-table driven work scheduling in Distributed Search (TDS) (Romein et al, 1999), a parallel version of IDA* (**?**). Parallel Frontier A* with Delayed Duplicate Detection (Niewiadomski, Amaral, and Holte 2006) uses a strategy based on intervals computed by sampling to distribute the work among several workstations. In 2009 (Kishimoto et al, 2009) Hash Distributed A* (HDA*) that combines ideas from previous parallel algorithms, the hash-based work distribution strategy of PRA* and the asynchronous communication strategy of TDS. Finally Adaptive K-Parallel Best-First Search (Vidal et al, 2010) where KBFS is parallelized using OpenMP.

Parallel Multi-Heuristic Search Fast Forward

We now introduce the main features of the planner PMHSFF (Parallel Multi-Heuristic Search Fast Foward). This approach is a modification of the FF planner (Hoffmann et al, 2001) (Hoffmann, 2005), where the heuristic calculation process is executed for multiple nodes in the same time

for the algorithm EHC. To make these modifications in the planner, we used the generic library in C for threads.

Initialization of global and local data
In the first stage the planner detect the number of available cores in the computer. Then, creates the control variables for each thread that will be writer during the heuristic calculation process.

Parallelizing the main loop
The algorithm EHC design in FF is based on the commonly used hill-climbing algorithm for local search, but differs in that breadth-first search forwards from the global optimum is used to find a sequence of actions leading to a heuristically better successor if none is present in the immediate neighborhood. In the parallel version of this algorithm for the first node the heuristic value is compute in sequence. Then the process select a number n (n have to be less or equal to the number of available cores) of successors of the node selected from the open list and computed the heuristic value at the same time (one for each thread). Next choose the best node with lower heuristic value. It was possible to parallelize the process of generation of successors but it may involve the occurrence of bottleneck with a large number of thread, increasing in some cases the execution time rather than decrease. This parallelization allows us to avoid bottlenecks, because we maintain the largest number of items shared (reader variables) and the variables or structures frequently used (writer variables) are replicated for each thread. In this case the delete relaxation (shared structure) is used to compute the heuristic value simultaneously by all nodes in each iteration. Through parallelization is possible to obtain two advantages over the sequential process.

- Reduce the execution time, computing the heuristic value in multiple nodes at the same time.
- Increase the size of the search tree in order to find other solutions with better quality that have not been selected by the sequential algorithm.

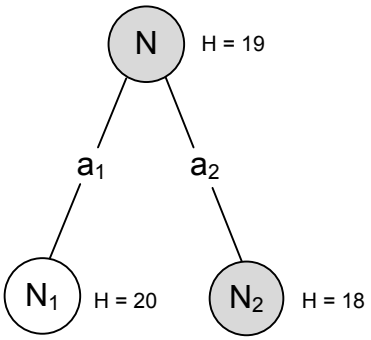

Figure 1: Sequential search process example

Figure 1 shows an example of the search process in sequential mode. In this case the search algorithm expands only two nodes, because the heuristic value of the second successor less than the parent node. The time using for this process is show in the equation (1), where t_g is the generation time of a node and t_h is the heuristic value calculation time.

$$t_s = 2 * (t_g + t_h) \tag{1}$$

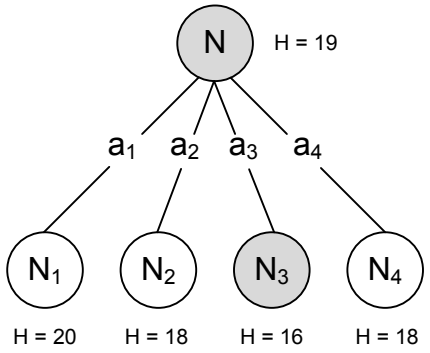

Figure 2: Parallel search process example

Figure 2 shows an example of the search process in parallel mode with four threads. In this case the parallel algorithm has generated sequentially four nodes (one per thread). Then it is computed the heuristic value of the nodes simultaneously. The time for this process can be calculate with the equation (2), where t_g is the node generation time , t_h is the heuristic value calculation time and t_{sinc} it the synchronization time among threads to ensure completion of all tasks.

$$t_p = 4 * t_g + (t_h + t_{sinc}) \tag{2}$$

Equation 3 show that computing the heuristic value of two nodes in sequential process is more expensive than compute the heuristic value of n nodes in parallel process. This property is possible because we have been prevented by the bottlenecks for the use of shared items.

$$t_h < (t_h + t_{sinc}) < 2t_h \Rightarrow t_{sinc} << t_h \tag{3}$$

However in automated planning in some cases, distributing the work among several cores could not be beneficial. For example, if the selected nodes are always on the left of the search tree (first node generated by the algorithm), use a large number of threads may increase the execution time and do not offer advantages over the sequential algorithm.

Conclusions

In this paper, we presented a simple parallelization over a currently planner for the International Planning Competition 2011. This techniques offers a temporary reduction in the search process. Furthermore allows to expand the searched space in order to find other solutions with different cost. The

application of this technique in planning offers great opportunities that should be investigated. For future works we plan to use simultaneous search algorithms or better heuristic calculation techniques obtaining a set of values for each node in order to make our heuristic more informed.

References

M. P. Evett, J. A. Hendler, A. Mahanti, and D. S. Nau. Pra: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 2:133–143, 1995.

P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Science and Cybernetics*, 2:100–107, 1968.

J. Hoffmann. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research*, pages 685–758, 2005.

J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, pages 253–302, 2001.

A. Kishimoto, A. Fukunaga, and A. Botea. Scalable, parallel best-first search for optimal sequential planning. *In Proceedings of the Nineteenth International Conference on Automated Scheduling and Planning, Thessaloniki, Greece.*, 19:201–208, 2009.

J. W. Romein, A. Plaat, H. E. Bal, and J. Schaeffer. Transposition table driven work scheduling in distributed. *In Proceedings of the Sixteenth International Conference on Artificial Intelligence*, 16:725–731, 1999.

V. Vidal, L. Bordeaux, and Y. Hamadi. Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. *In Proceedings of the Twentieth International Conference on Automated Scheduling and Planning*, 20, 2010.

OpenMP, 1997. A proposed standard API for shared memory programming. `http://www.openmp.org`.

MPI: A Message-Passing Interface Standard, 1995. `http://www.mcs.anl.gov/research/projects/mpi/`.

CUDA: Reference Manual, 2005. `http://www.cs.ucla.edu/~palsberg/course/cs239/papers/CudaReferenceManual_2.0.pdf`.

The LMTD Planner
On the Discovery and Utility of Precedence Constraints in Temporal Planning

Yanmei Hu[1], Dunbo Cai[2] and Minghao Yin[3]

Northeast Normal University, Changchun, China[1, 3]

huym260@gmail.com[1] , ymh@nenu.edu.cn[3]

Wuhan Institute of Technology, Wuhan, China[2]

dunbocai@gmail.com[2]

Abstract

LMTD is a satisfied planning system based on heuristic search. Based on the work of Eyerich et al (2009), it extends the precedence constraints contexts heuristic (h^{pcc}) (Cai et al. 2009) to a temporal and numeric setting. Its core feature is the rules, which are derived from landmarks and used to account precedence constraints among comparison variables and logical variables. It builds on the TFD Planning System, using multi-valued state variables and any time search. The planner will continue to search for plans of better quality until the limited search time is due or the remaining search space is empty.

Introduction

LMTD is a temporal planning system based on heuristic state space search. It builds on the TFD (Eyerich et al, 2009), inheriting the general structure of TFD. PDDL tasks with binary state variables are translated to *SAS+* formulism with multi-valued state variables, and comparison variables are introduced for considering numeric resources. A search architecture is designed to search solution in the search space. Every search state is time stamped, due to temporal planning considering temporal dependencies. A heuristic named *temporal precedence constrains contexts* (h^{tpcc}) is derived from landmarks to guide the search, in the spirit of temporal h^{cea} notated as h^{tcea} (Eyerich et al, 2009) and h^{pcc} (Cai et al. 2009). h^{tpcc} considers the precedence constraints over both comparison variables and logical variables, while h^{tcea} considers dependencies among fluents and h^{pcc} only suit for fluents.

Structure of Planner

LMTD consists of three separate programs:

1. the translator (written in Python). This part is used to translate temporal PDDL tasks to temporal *SAS+* tasks. Here we directly use the translator in TFD.
2. the knowledge compilation module (written in C++). In this part, a number of data structures, including data transformation graphs (DTG) and causal graph (CG), are built from the temporal *SAS+* task representation generated by the translator. These data structures play a central role in the generation of landmarks and search. The more details about knowledge compilation are referred to (Helmert, 2004). To handle numeric resources, comparison variables are introduced. In these data structures, comparison variables are also considered. The more details about comparison variables are referred to (Eyerich et al, 2009).
3. the search engine (also written in C++). Using the data structures generated by the knowledge compilation module, the search engine attempts to find a plan using heuristic search with some enhancements, such as the use of *preferred operators* and *deferred heuristic evaluation* (just like that in Fast Downward). LMTD applies a greedy best-first search as the search algorithm and use the *temporal precedence constraints context heuristic* (h^{tpcc}) to guide the search (in the next section, we will introduce the heuristic in detail). Once a plan solution is found, it searches progressively for better solutions until the search is terminated.

To solve a planning task, the three programs are called in sequence; they communicate via text files.

Temporal Precedence Constraints Context Heuristic

Like in TFD, for using the heuristic to guide search, Instant actions are extracted to approximate the durative actions. And for the numeric variables, they do not directly occur in conditions or in the goals but only influence them via comparison variables, so it is sufficient to consider these comparison variables and logical variables (details are referred to Eyerich et al (2009)). In h^{tcea} , notice that other conditions is evaluated in the context state satisfying the pivot condition. That's mean for an atom, we first achieve the pivot condition of the chosen rule and get a context state, and then the remaining conditions are satisfied in the gotten context state. However, in many cases, it is not

reasonable to evaluate conditions in this order. Here we follow the example in Eyerich et. al (2009). Assuming that there are three locations l_0, l_1 and l_2, robot r_1 is at l_1 with a water tank which has capacities of $c = 150$ units and is initially empty $w = 0$, additionally, the robot can only refill its tank at l_0 and there is a path between every two locations. Now it is required to water flower f_1 at location l_1 and flower f_2 at location l_2 with each has current water levels $h_i = 0$ and needs to be watered until levels $n_i = 10$ is reached ($i = 1, 2$). For water some flower at l_j, the rule could have the form "$f_i = unwatered, at(l_i), w - (n_i - h_i) \geq 0 \rightarrow f_i = watered$" ($n_i - h_i = 10$, so the water tank must be fill in at least 10 units water). Hence the estimated cost of $f_1 = watered$ based on h^{tcea} is $d_{01} + 10$ (d_{ij} is the distance between d_i and d_j). While it is more reasonable if we first achieve the condition $w - (n_1 - h_1) \geq 0$, and then achieve $at(l_1)$ in the context state satisfying $w - (n_1 - h_1) \geq 0$. We can obtain a more precise estimated cost $10 + 2 * d_{01}$. Therefore, in this section, we define the h^{tpcc} heuristic function, extending h^{tcea} to capture the order information discussed above. Here we first give out an important notation and introduce the equation of h^{tpcc}, then we give the highlight to how to obtain the precedence constraints which indicating the orders over conditions.

Context Functions

Here we borrow Cai et al.'s notation, each instant action is transformed to a set of rules, and rules are considered to satisfy some condition. Context function is a partial function $ctx : R \times P \mapsto P$, where $ctx(r, q) = p$ indicates that the context of condition $q \in Z_r$ should be the state that results from achieving $p \in Z_r$. Given a rule r, assume that we have constructed the context function for conditions Z_r (how to construct it? We will leave this for later), then a directed and acyclic graph corresponding to the context functions can be build, which explicitly show the precedence constraints between conditions. Each node indicates one condition in Z_r, and there is an edge from p to q if $ctx(r, q) = p$. Moreover, each node has at most one parent to ensure that every condition in Z_r just has one context. So we can get that the corresponding graph is a forest,
and $Z_r^L := \{y \mid y \in Z_r, not \text{ ex. } y' \in Z_r \text{ s.t } ctx(r, y') = y\}$ denotes the leaves, $Z_r^R := \{y \mid y \in Z_r, ctx(r, y) \text{ is undef.}\}$ denotes the roots.

The definition of h^{tppc}

By introducing the precedence constraints over conditions Z_r, we derive h^{tpcc} from h^{tcea}, and

$$h^{tpcc}(x \mid s) = \begin{cases} 0 & \text{if } x \in s \\ \min_{r: Z_r \rightarrow x} (c(s') + \sum_{y \in Z_r} h^{tpcc}(y \mid s^{tc}(y, r, s))) & \text{if } x \notin s, \\ & \text{var}(x) \notin V_c \\ \min_{\substack{r: Z_r \rightarrow v \circ v' \in \\ prom(x,s)}} (c(s') + \sum_{y \in Z_r} h^{tpcc}(y \mid s^{tc}(y, r, s))) & \text{if } x \notin s, \\ & \text{var}(x) \in V_c \end{cases} \quad (1)$$

This equation has the similar structure with h^{tcea} and naturally be extended. In our heuristic function we don't just limit the context state to the pivot condition, but also consider the context state corresponds to the other conditions. Each condition y is evaluated in the resulted state after achieving its direct parent condition in the landmark graph. The resulted state is projected by the function $s^{tc}(s, r, y)$. This function is a map from three sets: states, rules and conditions to the sets of states. In this function, r is applied to achieve x from the corresponding context state s, y is specified to achieve in the new state obtained by the function itself. Similarly, we have the following formal result:

$$s^{tc}(s, r, y) = \begin{cases} s & \text{if } ctx(r, y) \text{ is undef} \\ s[s^{tc}(y', r, s)][r'] & \text{if } ctx(r, y) = y' \end{cases} \quad (2)$$

In the first case in equation (2), that's if $ctx(r, y)$ is undefined, then the context state of y is the same as the original state s applying r. While in the second case, that's if $ctx(r, y) = y'$, then the context state of y is the state that results from achieving y', in corresponding context state indicated by $s^{tc}(s, r, y')$, captured by iterating from its respective root $y_0 \in Z_r^R$. s^{tc} also consider the changes by the rule r' applied to achieve y'. Moreover, what is the state after achieving x? Here we need to consider two things. One is that for achieve x, we should achieve the conditions of the chosen rule ("best rule") r^m, according to equation (1), in the order obtained in advance and indicated by ctx. Recall that the graph indicating the precedence constraints for conditions of each rule is a forest, so the result must be the state after achieving some leaf $y_0 \in Z_r^L$ (the leaf is the last one to achieve) in its respective context state, denoted as $s[s^{tc}(y_0, r, s)]$. However, there may be several leaves in the forest, which one we should pick up? Since they have no order constraints to each other, we can arbitrarily pick one up and treat it as s' in equation (1). The other is that incorporating the changes made by r^m itself. This can be solved by capturing its "side effects" as in h^{cea} (Helmert and Geffner, 2008). Therefore, the final state corresponding to x is $s[s'][r^m]$.

We embed the h^{tpcc} into the best-first-search algorithm, and use it to guide the search process. For every time stamped state met in search, we using the h^{tpcc} to evaluate the cost to the goals, and choose the state having smallest heuristic cost as the expanded state and to generate a new search state till the goals are achieved.

Obtaining Temporal Precedence Constraints

Recently, landmarks attract a large number of researchers, and it is well known that the current heuristic estimators based on them works well in the planning, eg the inadmissible heuristic h^{LM} in the LAMA planner (Richter, Helmert and Westphal, 2008), the admissible heuristics h^{LA}, h^{L} and h^{LM-cut} (Helmert and Domshlak, 2009). For a given propositional planning task, Porteous, Sebastia and

Hoffmann (2001) define that landmarks are propositions that must be true at some point in every solution plan. Also, they propose an algorithm called LM^{RPG} to extract landmarks and their orderings from the relaxed planning graph of a planning task. Richter, Helmert and Westphal (2008) firstly derive landmarks to *SAS+* tasks and propose a landmark-based heuristic h^{LM} mentioned above. Herein, we derive landmarks to temporal and numeric planning task, and apply them to generate the precedence constraints over rule conditions.

Definition 1 landmark

Let $\prod = < V, s_0, s_*, A, O >$ be a temporal planning task.

A logical landmark is an atom associate logical variable that must be true at some point in every plan of $\prod$.

A comparison landmark is a atom associating comparison variable that decided by the numeric expression which must hold at some point in every plan of $\prod$.

The logical landmark is as same to that in LAMA. While for explaining the comparison landmark, we return back to the example mentioned above. For achieve the goal that flower f_1 is watered until its level $n_1 = 10$, it's easy to know that the numeric expression $w - (n_1 - h_1) \geq 0$ as the condition of the action water f_1 must hold at some point in every plan. Patrick et. al introduce auxiliary variables to represent the numeric condition, see Fig. 1. aux_3 is a comparison variable, the figure visualize the comparison axiom of aux_3 and also show the dependency graph of aux_3 and numeric variables aux_2, aux_1. $aux_3 = aux_2 \geq 0$ express the condition from the angle of comparison variable and $aux_2 \geq 0$ must be true at some point in every plan. Hence $aux_3 = true$ is a comparison landmark.

The precondition $w - (n_1 - h_1) \geq 0$ must holds.

$aux_3 = aux_2 \geq 0$

$aux_2 = w - aux_1$

$aux_1 = n_1 - h_1$

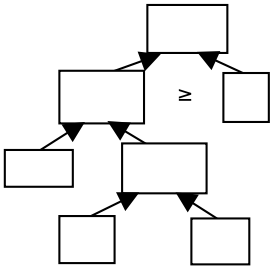

Figure 1: visualization of numeric and comparison axioms.

How do we apply the landmarks to generate the precedence constraints for temporal planning task? Firstly, we build a landmark graph in which nodes indicate landmarks and arcs indicate orderings between landmarks (to guarantee landmark graph acyclic, possible cycles are broken). Secondly, according to the landmark graph adding some rules, we extract the temporal precedence constraints and explicate them as context functions.

To build the landmark graph, we borrow the methods from Hoffmann et .al (2003) and Richer et .al (2008).Following is the briefly explain, and the details refer to the original paper: 1. Set the goals as the initial landmarks, backtrack the relaxed graph to get other landmarks and orderings (If A is a landmark in step i of relaxed graph, and the operators in step i-1 that achieve A has a common precondition B, then B is a candidate landmark which is ordered before A). In addition, one-step lookahead and domain transition graph are applied to find further landmarks. Disjunctive landmarks are considered too 2. A sufficient criterion is applied to eliminate non-landmarks: each fact A is tested by removing all achievers of A from the original task, and then check whether the task is still relaxed solvable (since Hoffmann et al. (2003) show that deciding if a fact is a landmark and deciding orderings between facts are PSPACE-complete, the detection of landmarks and orderings is approximate, so there is no guarantee that the generated landmarks are sound). 3. Further orderings is introduced. For two landmarks A and B, if achieving A must delete B, or if achieving A can decrease the cost of achieving B, then we add the order $A \rightarrow B$.

To obtain precedence constraints, we consider each rule r appearing in the heuristic cost computation and extract ctx for Z_r by the following several rules:

Rule 1 $p, q \in Z_r$ and p, q are landmarks, if p is directly orders before q in landmark graph, then $ctx(q, r) = p$.

Rule 2 $p, q \in Z_r$, p is a comparison fact and q is a logical fact. If $\exists r \in prom(p, s) \wedge \exists x \in Z_r \wedge \text{var}(x) = \text{var}(q)$, then $p <_d q$ (p directly orders before q).

According to Rule 1, we traverse the topology order for Z_r from Landmark Graph, if $p \in Z_r$ is directly order before $q \in Z_r$, then $ctx(q, r) = p$. But if there are more than one facts (in Z_r) directly order before q, we arbitrarily choose one. However, how to embed the comparison facts into the topology order? Rule 2 is applied to solve this question for obtaining more information about temporal precedence constraints. To explain this, we consider the example of watering f_1. $r: f_1\_watered = 0, \text{var}\_p = l_1, aux_3 = 1 \rightarrow f_1\_watered = 1$ is chosen to be the "best rule" (transformed from the propositional form $r: f_1 = unwatered, at(l_1), w - (n_1 - h_1) \geq 0 \rightarrow water\ f_1$, the detail of transforming is referred to Helmert (2004)) and the current state is $s = \{\text{var}\_p = l, aux_3 = 0, f_1\_watered = 0, w = 0, h_1 = 0\}$. We can know that $fill\ tank\ l_0$ (fill the water tank of robot at location l_0) is an action that can change the value of aux_3 since it can increase w, so $r': \text{var}\_p = l \rightarrow w = 150 \in prom(aux_3 = 1, s)$. Supposing we set $\text{var}\_p = l_1 < aux_3 = 1$, when achieve $aux_3 = 1$, $\text{var}\_p = l_1$ is deleted due to $\text{var}\_p = l$'s addition, and according to Koehler and Hoffmann's (2000) goal orderings (if achieve A should delete B, then A < B), $aux_3 = 1 < \text{var}\_p = l_1$ holds, contradictory to our suppose. Hence, we should set $aux_3 = 1 <_d \text{var}\_p = l_1$.

As we known that when we detect the rules to obtain the "best rule" in equation (4), we will meet some facts that are not in landmark graph. That's mean not all the conditions of actions are landmarks. How do we capture the orderings between those facts? We draw on the previous works on goal orderings (Koehler, Hoffmann, 2000). For logical

facts we just directly follow the methods in Koehler and Hoffmann, and leave comparison facts to Rule 2 for simple.

Implementation and Discussion

We implement h^{tpcc} on top of the code of TFD (Eyerich et al, 2009) and LAMA (Richter et al. 2008). Additionally, we consider the estimated cost of comparison variable just like that in h^{tcea} for simple. To evaluate it, we test some benchmarks used in the temporal satisficing track of IPC 2008 and obtain some primary results. From the primary results, we can see the potential power of h^{tpcc} on the improving of solutions, although it works worse on several benchmarks, which is mostly due to our currently very rough implementation. Therefore, we consider h^{tpcc} as a promising heuristic and will improve and perfect our implementation in the future work.

References

Chih-wei, Hsu., and Benjamin W, Wah. 2008. *The SGPlan planning system in IPC-6*. http://manip.crhc.uiuc.edu/Wah/papers/C168/C168.pdf.

Dunbo, C., Hoffmann, J., and Helmert, M. 2009. *Enhancing the context-enhanced additive heuristic with precedence constraints*. In Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009), Thessaloniki, Greece, September 19-23 2009.

Helmert, M. 2004. *The fast Downward planning system*. http://www.informatik.uni-freiburg.de/~helmert/pub.

Helmert, M., and Geffner, H. 2008. *Unifying the causal graph and additive heuristics*. In Proc. ICAPS 2008:140-147.

Helmert, M., and Domshlak, C. 2009. *Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?* 19th International Conference on Automated Planning and Scheduling (ICAPS 2009), Thessaloniki, Greece, September 19-23 2009.

Hoffmann, J., Porteous, J., and Sebastia, L. 2003. *Ordered landmarks in planning*. Journal of Artificial Intelligence Research 22:215–278.

Koehler, J., and Hoffmann, J. 2000. *On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm*. JAIR 12:338–386.

Patrick, E., Robert, M., and Gabriele, R. 2009. *Using the context-enhanced additive heuristic for temporal and numeric planning*. In Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009):130-137. AAAI Press 2009.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. *On the extraction, ordering, and usage of landmarks in planning*. In Cesta, A., and Borrajo, D., eds., Pre-proceedings of the Sixth European Conference on Planning (ECP'01), 37–48.

Sebastia, L.; Onaindia, E.; andMarzal, E. 2006. Decomposition of planning problems. AI Communications 19(1):49–81.

Richter, S., Helmert, M., and Westphal, M. 2008. *Landmarks Revisited*. Proceedings of the Twenty-Third AAAI conference on Artificial Intelligence.

A Planner to Solve Temporally-Expressive Problems and to Exploit Floating Solutions-Plans

F. Maris **P. Régnier**

IRIT, Université Paul Sabatier
118 route de Narbonne
31062 Toulouse, cedex 9, France.
{maris, regnier}@irit.fr

Introduction

TLP-GP [Maris, Régnier, 2008], is a planner based on the use of a simplified planning graph and a Disjunctive Temporal Problem (DTP) solver. It can solve problems expressed in a language whose temporal expressivity is greater than that of PDDL2.1 since preconditions can be required and effects can occur on any temporal interval relative to the start-time of an action. TLP-GP can also take into account, in a natural way, exogenous events and temporally extended goals which must be true over a certain period of time. It is complete for the temporally expressive sublanguages of PDDL2.1.

The TLP-GP Algorithm

TLP-GP [Maris, Régnier, 2008] uses a similar method to that used by the family of BLACKBOX [Kautz, Selman, 1999] and GP-CSP [Do, Kambhampati, 2001] planners. The planning graph is built until the goal are obtained, following the classic algorithm for the atemporal case, without the calculation of mutexes. TLP-GP then looks for a solution-plan searching backwards in the planning graph, using a Disjunctive Temporal Problem (DTP) solver. To achieve this, it places temporal constraints between actions and uses an agenda which is different to that used by TGP.
Instead of a placing actions and propositions on a discrete time-line, TLP-GP uses real-valued temporal variables on which constraints are defined. At each back-chaining step, this set of constraints is fed into a solver which verifies global consistency. This operation is repeated until a solution is obtained. In case of failure, another level is added to the planning graph and the back-chaining process is restarted.

Representation Language

Actions are represented by 4-tuples (<action-name>, <preconditions>, <effects>, <duration>). <effects> and <preconditions> are sets of propositions each with a corresponding temporal label. This label represents an interval, defined relative to the start-time of the action, during which a precondition must be verified or an effect is produced. A label [t,t] containing a single value t is written simply as [t]. <duration> represents the duration of the action. We use $\tau_s(A)$ to denote the temporal variable corresponding to the start-time of the action A. We introduce two imaginary and instantaneous actions A_I and A_G which correspond, respectively, to the initial state and the achievement of the goal. The start and end times of a plan are given by $\tau_I = \tau_s(A_I)$ et $\tau_G = \tau_s(A_G)$.

Example: consider the action (A, {$a_{[-1,\ 2]}$, $b_{[0]}$}, {$\neg a_{[3]}$, $c_{[5,7]}$}, 5). If $\tau_s(A)$ is the start-time of A, its duration is 5, the proposition a must hold between $\tau_s(A)$-1 and $\tau_s(A)$+2, b must hold at instant $\tau_s(A)$, ¬a appears at $\tau_s(A)$+3 and c appears at $\tau_s(A)$+5 and remains true at least until $\tau_s(A)$+7.

In the simpliest version of TLP-GP, each precondition or effect p have an associated interval over which p must be true and not p false. This corresponds to "over all" preconditions in PDDL2.1, extended to effects by [Cushing & al., 2007.b]. However, to represent real-word domains, we have implemented a more expressive language in which we can represent the fact that a precondition or effect p must be true (and not p false) during a minimal duration d anywhere within an interval [a,b]. Our language also allows the user to disassociate p and not p by stipulating, for example, that p must be true at the end of an interval [a,b] over all of which not p cannot be established. It is also possible, in a natural way, to represent external events or goals which have a duration. The former can be easily coded as effects of the dummy action A_I and the latter as preconditions of the dummy action A_G.

```
(:durative-action fire-kiln2
   :parameters (?k - kiln20)
   :duration (= ?duration 20)
   :condition (over all (energy ))
   :effect (and
        (somewhere [start (+start 2)] (ready ?k))
        (over [(+ start 2) end[ (ready ?k))
        (at end (not (ready ?k)))))
```

```
(:durative-action bake-ceramic1
   :parameters (?p - piecetype1 ?k - kiln)
   :duration (= ?duration 15)
   :condition (over all (ready ?k))
   :effect (and
        (over [start end[ (not (baked ?p)))
        (over [start end[ (baking ?p))
        (at end (not (baking ?p)))
        (somewhere [(- end 5) end] (baked ?p))))

(:durative-action treat-ceramic1
   :parameters (?p - piece)
   :duration (= ?duration 4)
   :condition (and (over all (baking ?p)))
   :effect (and (minimal-duration 3 anywhere
                      [start end] (treated ?p))))
```

In the "temporal-machine-shop-2-3" domain the action *fire-kiln* is described using (somewhere [start (+ start 2)] (ready ?k)) to express the fact that the kiln will be ready at an unknown instant between start and start + 2. The expression (over [(+ start 2) end[(ready ?k)) is used to enforce the kiln to be ready up to the end. Moreover, the time wich is necessary to bake a ceramic is not completely known and can be represented using (somewhere [(- end 5) end] (baked ?p)). A ceramic can be treated anywhere between the start and the end of the action *treat-ceramic*.

Expansion of the Temporal Planning Graph

Unlike the planners TGP [Smith, Weld, 1999] or LPGP [Long, Fox, 2003], TLP-GP uses a planning graph without mutexes and constructed in an atemporal manner, without taking into account, to start with, either the duration or the start-time of actions. Conflicts between actions, including mutual exclusions, are managed entirely during the solution extraction phase by a constraint satisfaction system. This minimal usage of the planning graph means that TLP-GP does not have the same restrictions as other planners: decision epochs or time-line which limits the completeness to temporally simple problems or a number of levels which is too large to allow the practical resolution of temporally expressive problems (LPGP). The method we have chosen consists in entrusting a large part of the work to a DTP solver. This method has the double advantage of producing floating temporal plans and considerably increasing the expressivity of the representation language that can be used. This strategy turned out to be very effective in the context of classical planning (cf. the SATPLAN'04 system, successor to BLACKBOX [Kautz, Selman, 1999], which is still the winner in the optimal-planner category of the IPC'06 competition).

Since an action can produce and destroy the same proposition at different times, we also store in the graph the negations of propositions. Finally, unlike other temporal planners based on GRAPHPLAN [Blum, Furst, 1995], the levels are not linked to a fixed time scale. Level 0 contains the dummy action A_I, which has no preconditions and produces all the propositions of the initial state, together with the corresponding effect arcs. In our running example [Maris, Régnier, 2008], we omit level 0 since I is empty. For each level $n \geq 1$, we apply all those actions whose preconditions are all present at level n-1 and we add the corresponding precondition arcs to the graph. We then add the effects of these actions at level n together with the corresponding effect arcs. The graph is built in this way level by level until all the goals appear. Finally, we build an extra level containing the dummy action A_G, which has no effects and has all the goals as its preconditions, and we add the corresponding precondition arcs.

The extraction algorithm is then called. Each arc in the planning graph has a corresponding temporal label according to its type:

- **Precondition arc** (proposition → action): the label represents the interval, relative to the start of the action, over which the precondition must be verified.
- **Effect arc** (action → proposition): the label represents an interval, relative to the start of the action, at the start of which the effect appears and during which it remains true. After the end of this interval, there is no guarantee that it still holds.

Extraction of a Floating Solution-Plan

Once the planning graph has been extended to a level at which all goals are present, TLP-GP searches backward for a solution-plan. To this end, it places temporal constraints between actions and, instead of assigning actions and propositions to fixed time points, it uses mutually constraining real-valued temporal variables. At each step, the set of constraints is fed to a disjunctive temporal constraint satisfaction system which checks its satisfiability. This operation is repeated until a solution is found. In case of failure, an extra level is added to the planning graph and the back-chaining procedure restarted. The simplified extraction algorithm is given below. The search for a solution requires two data structures: *Agenda* and *Constraints*.

Agenda is a set of lists, one for each proposition. For a proposition p, the corresponding list Agenda(p) is composed of intervals of the form $[\tau_s(A)+\delta_1, \tau_s(B)+\delta_2]$, over which p must be true. When the interval consists of a single value, we denote it by $[\tau_s(A)+\delta]$. Two types of temporal intervals can be added to Agenda(p):

- Intervals which correspond to a causality relationship between actions (if the proposition p is produced by an action A in order to fulfill a precondition of an action B, p must remain true until used by B).
- Intervals which correspond to the appearance of the effects of a selected action (appearance of p).

Constraints is a list of disjunctions of (at most two) binary temporal constraints between time points:

- Constraints corresponding to causality relationships representing the fact that a precondition must be produced (by an action A at time $\tau_s(A)+\delta_1$) before it is required (by an action B at time $\tau_s(B)+\delta_2$). Such constraints are not, in

fact, disjunctions since they are of the form $\tau_s(A)+\delta_1 \leq \tau_s(B)+\delta_2$.

- Constraints imposing the non-intersection of the two intervals $[\tau_s(A)+\delta_1, \tau_s(B)+\delta_2]$ and $[\tau_s(C)+\delta_3, \tau_s(D)+\delta_4]$ over which a proposition and its negation hold. In the most general case, these constraints are disjunctive since they are of the form $(\tau_s(B)+\delta_2 \leq \tau_s(C)+\delta_3) \bullet (\tau_s(D)+\delta_4 \leq \tau_s(A)+\delta_1)$. The inequalities may or may not be strict depending on the type of intervals under consideration (open, closed or mixed). In most cases, this constraint simplifies to a non-disjunctive constraint.

The set of constraints created by TLP-GP therefore constitutes a disjunctive temporal problem (DTP). The problem of solving or verifying the consistency of a DTP is NP-hard [Dechter, Mieri, Pearl, 1991] but the performance of the algorithms to solve these problems are regularly improved. The default ordering heuristics used in the choice of subgoals to be established and the choice of actions to establish them is the following: priority is given to subgoals which appear (for the first time) in the highest levels of the graph and, among actions which establish them, priority is given to those actions which appear (for the first time) in the lowest levels of the graph.

Extraction algorithm

```
Goals ← Pre(A_G);
For every effect e • Eff(A_I):
   Add an interval I to Agenda(e) for the
   apparition of the proposition e;
End For;
While Goals ≠ •
   For every proposition p • Goals:
      Goals ← Goals – p;
      Select (* Backtrack point *), using the
      heuristic, an action A to produce p;
      Goals ← Goals • Pre(A);
      Add a precedence constraint between A and
      B, the action whose p is a precondition;
      Add an interval I to Agenda(p) to maintain
      the precondition p;
      For every interval I in Agenda(¬p):
         Add a constraint to forbid the
         overlapping of I and I';
      End For;
      For every effect e of A, e ≠ p:
         Add an interval I to Agenda(e) for the
         apparition of e;
         For every interval I' in Agenda(¬e):
         Add a constraint to forbid the
         overlapping of I and I';
         End For;
      End For;
      Check the consistency of constraints (call
      to the DTP solver);
      In case of failure, return to the back-
      track point to select another action A;
   End For;
End While;
If constraints is satisfiable
   Then return the floating solution-plan
   (selected actions and constraints)
   Else there is no solution in this level;
End If;
End.
```

Use of TLP-GP

Generalities

TLP-GP is implemented in OCaml 3.09.2. TLP-GP uses the SMT (Sat Modulo Theory) solver MathSat[1] 3.4 which is known to perform well on DTP (cf. results of the SMT-COMP'06[2] competition). The archive contains the source code of TLP-GP and statically linked binaries of MathSat for Linux. Use "make" to build the binaries of TLP-GP. If you want to perform the program on an other operating system, you have to ask the authors for the good binaries of MathSat.

The command line to run TLP-GP:

```
./tlp-gp domain.pddl problem.pddl
```

A graphic interface in Java is also available to run TLP-GP and to handle and store floating solutions-plans.

Experimental results

We compared TLP-GP with two state-of-the-art planners capable of solving temporally expressive problems: the LPGP [Long, Fox, 2003] planner which is an extension of GRAPHPLAN, and the partial-plan space planner VHPOP [Younes, Simmons, 2003]. Given that the previous temporal benchmarks were inappropriate, since temporally simple, we drew up several new temporally expressive benchmarks. All these benchmarks can be found at the adress which is given below[3].

The first test domains extend the problem of [Cushing et al., 2007.a, figure 3] in three different ways. The domain, tms-k-t-p (temporal machine shop, [Cushing et al., 2007.a]) is inspired by a real-world application. It concerns the use of *k kilns*, each with different baking times, to bake *p* ceramic *pieces* (*bake-ceramic*) of *t* different *types*. Each of these types requires a different baking time. These ceramics can then be assembled to produce different structures (*make-structure*). The resulting structures can then be baked again to obtain a bigger structure (*bake-structure*). The "cooking" domain allows us to plan the preparation of a meal, as well as its consumption by respecting constraints of warmth. Problems cooking-carbonara-n which we used for this test allow us to plan the preparation of n dishes of pasta. The concurrency of actions is required to obtain the goal because it is necessary that the electrical plates works so that water and oil are hot to cook pasta and bacon cubes. It is also necessary to perform this baking in parallel to serve a hot dish during its consumption.

[1] http://mathsat.itc.it/
[2] http://www.smtcomp.org/2006/
[3] http://tlpgp.free.fr/

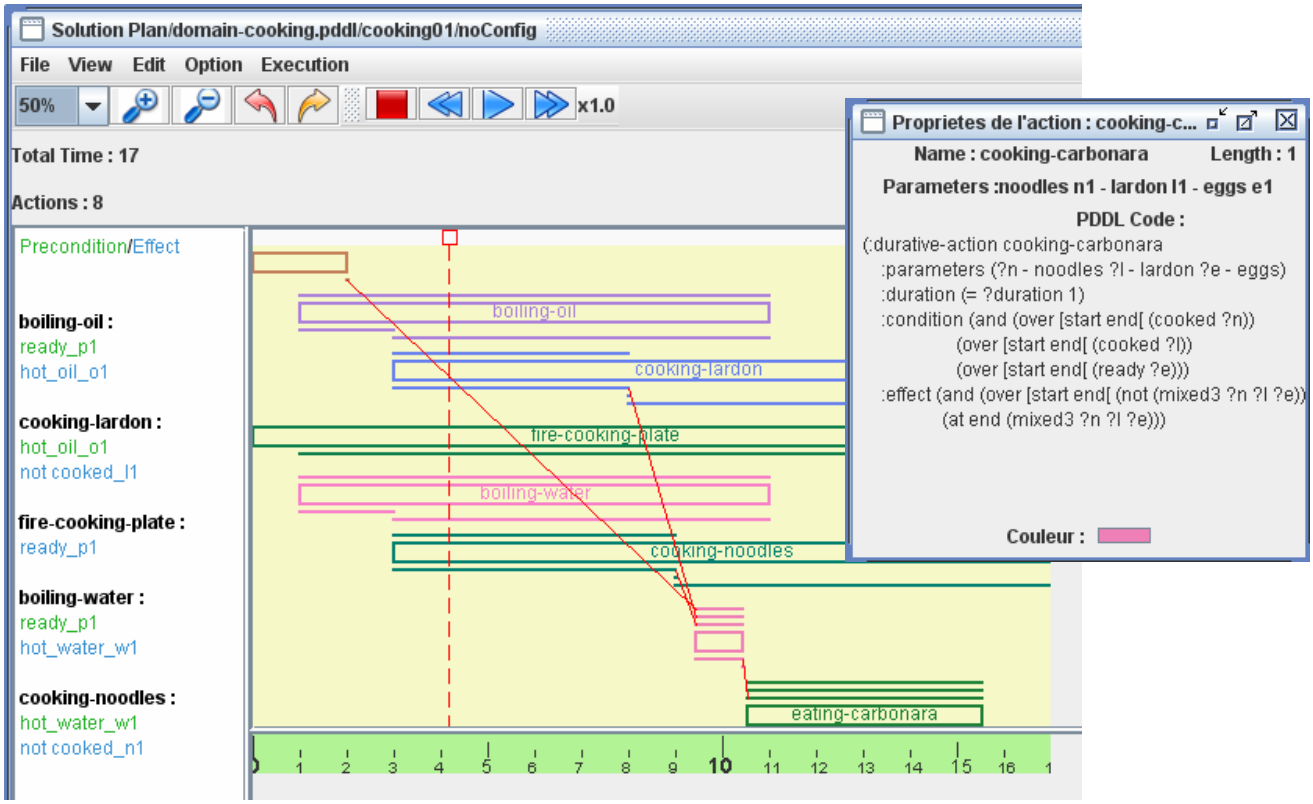

Figure 1 – The graphic interface of TLP-GP (solution of problem cooking-carbonara-01)

As the expressivity of LPGP and VHPOP is less than that of TLP-GP, we tested simplified, but nevertheless temporally expressive, versions of these benchmarks. On these benches TLP-GP clearly outperforms LPGP as well as VHPOP. The results can be found at the address given below[4].

For the "cooking-carbonara-01" problem, TLP-GP finds a floating solution-plan and returns an example of valid static plan providing possible start times. The floating plan can be recovered using the files default.smt and plan.txt which contain the constraints and variables assignments. The graphic interface of TLP-GP allows us to exploit directly floating solutions-plans (Figure 1).

Conclusion

We have presented TLP-GP, a planner which can solve temporally expressive problems in a language whose expressivity is greater than PDDL2.1. No compromise needs to be made in terms of completeness in order to achieve this expressivity. On temporally expressive benchmarks, TLP-GP performed much better then two state-of-the-art planners (LPGP and VHPOP) capable of solving the same types of problem. These results indicate the possibility of representing and solving problems which are closer to real-world applications. The production of a floating solution-plan rather than a single fixed solution also allows for a greater flexibility during the execution of the plan.

[4] http://tlpgp.free.fr/

References

[Blum, Furst, 1995] A.Blum, M.Furst, "Fast Planning Through Planning Graph Analysis", IJCAI, 1995.

[Cushing & al., 2007.a] W.Cushing, S.Kambhampati, Mausam, D.S.Weld, "When is temporal planning really temporal ?", IJCAI, 2007.

[Cushing & al., 2007.b] W.Cushing, S.Kambhampati, K.Talamadupula D.S.Weld, Mausam, "Evaluating temporal planning domains", ICAPS, 2007.

[Dechter, Mieri, Pearl, 1991] R.Dechter, I.Mieri, J.Pearl, "Temporal constraint networks", AI, 49: 61-95, 1991.

[Do, Kambhampati, 2001] M.B.Do, S.K Kambhampati, "Planning as Constraint Satisfaction: Solving the Planing Graph by compiling it into CSP", AI, 132, 2001.

[Kautz, Selman, 1999] H.Kautz, B.Selman, "Unifying SAT-based and Graph-based Planning", IJCAI, 1999.

[Long, Fox, 2003] D.Long, M.Fox, "Exploiting a graphplan framework in temporal planning", ICAPS, 2003.

[Maris, Régnier, 2008] F.Maris, P.Régnier, "TLP-GP: Solving Temporally-Expressive Planning Problems", TIME, 2008.

[Smith, Weld, 1999] D.E.Smith, D.Weld, "Temporal planning with mutual exclusion reasoning", IJCAI, 1999.

[Younes, Simmons, 2003] H.L.S.Younes, R.G.Simmons, "VHPOP: Versatile Heuristic Partial Order Planner", Journal of AI Research, 20, 2003.