Alien: Return of Alien Technology to Classical Planning

Masataro Asai
guicho2.71828@gmail.com

Abstract

Recently, a modern Classical Planning framework Fast Downward (Helmert 2006) is the go-to framework in the planning community. Despite its huge contribution, the framework have several design problems: (1) code quality and extensibility, (2) preprocessing speed, (3) low-level performance. In this IPC submission, we present a new classical planning framework, Alien, with emphasis on preprocessing and low-level performance. While we do not believe this library will get as popular as Fast Downward (the only expected users of this framework are us), we believe some of the design choices might influence future planners. Tl;dr: C/C++ is too slow.

1 Introduction

In recent years (post-2000), classical planning solvers are written in languages such as C or C++, assuming that the resulting binary automatically achieves almost-optimal low-level performance thanks to the compiler improvement. This is not true, mainly due to the limitations in these languages that they cannot optimize the low-level instruction sequences for the problem instance at hand – They apply the same, fixed instruction sequence that iterates/recurses over data structures, to different data. This behavior is similar to a byte-code interpreter, which is reading and interpreting a data structure loaded on main memory instead of directly running the assembly instructions that achieve the same behavior. The approach is slower than the native compilation even if the interpreter itself is compiled by GCC or Clang.

The choice of these languages also carries a significant burden on the extensibility & composability of the resulting solver with external services such as web servers, debuggers, visualizers etc. While it is possible to connect a planner to these systems, it is typically done via a coarse-grained API such as command line options and standard I/O, and is not easily "pluggable": It does not allow users to attach knobs at every corners, unless a significant modification is performed on the source code. While such flexibility might be achieved by interpreted programming languages such as Ruby or Python, we cannot sacrifice the low-level performance for a computationally intensive task like Classical Planning. To achieve flexibility and speed, one should use a flexible compiled language.

Two examples of such lack of extensibility are the adaptation of Fast Downward for parallel processing (Jinnai and Fukunaga 2017) or external memory search (Lin and Fukunaga 2018). In the former case, they had to implement process-level parallelism via MPI because the open/closed lists in Fast Downward are assuming single-threaded execution. In the latter case, not only the state database needs to be rewritten, but also the interface to heuristic functions and other pieces should be modified because they are tightly coupled to the state database.

Finally, Fast Downward uses python-based grounding process (PDDL-SAS+ translation) which becomes slow on certain instances e.g. very large instances with repetitive structures (Asai and Fukunaga 2014) or a problem instance automatically generated from images using neural networks (Asai and Fukunaga 2018). To even start solving the problem one should improve the performance of grounding processes for actions and state variables.

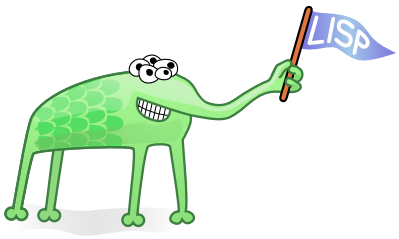

Figure 1: A **Lisp alien**. "To most programmers, Lisp seems like an entirely alien language at first- (...) this strangeness is not an arbitrary obstacle, but a necessary adjustment that imparts great power to programmers that would otherwise be unattainable. The alien Lisp mascot and quirky logo designs are designed to accentuate the awesome (and, to most people, alien) power that Lisp languages have- At the same time, they show how fun Lisp programming tends to be and that Lisp has wide appeal far beyond the stuffy academia it is sometimes wrongly associated with." (Barski 2007)

We tackle these issues by using ANSI Common Lisp programming language (Fig. 1) combined with B-Prolog (Zhou 2012), a modern high-performance Prolog solver with tabling predicates (Van Gelder, Ross, and Schlipf 1991) for preprocessing / grounding process. Common Lisp addresses

the first two issues of low-level performance and extensibility. B-Prolog addresses the third issue by its heavily optimized implementation.

2 PDDL Preprocessing

2.1 Prolog the Language

Prolog is a logic programming language that describes a program with a set of horn clauses. In Prolog, a program consists of rules and facts, both described in first-order logic terms. A rule looks like `<Head> :- <Body>.`, where `Head` is a term, and `Body` consists of several *subgoal* terms $\texttt{<sg>}_1$, $\texttt{<sg>}_2$.... A fact is a rule without body, and can be written as `<Head>.`.

A term can be a number, an atom (e.g. `cat`), a variable (e.g. `X`) including wildcards (`_`), or a compound form `predicate(arg1, arg2...)` where each $\texttt{arg}_i$ is also a term.

To *achieve* a subgoal `predicate(arg1, arg2...)`, Prolog interpreter performs a process called *Unification*. It first looks for a rule/fact in the program whose head has the same predicate and has the same instantiated arguments (arguments that are numbers/atoms) as the subgoal. Next, it assigns a value to each unassigned variable, using existing assignments as much as possible, while it also enumerates all combinations when no existing assignment is available. For each such combination of value assignment, it tries to achieve every subgoals, hence the Prolog interpreter performs a depth-first search on the subgoals. When there is no matching rule in the program, it backtracks and tries another combination of assignments. There is a top-level clause called `initialization` and Prolog tries to achieve this on launch.

Tabling While the default depth-first search method is good for most querying purposes, it has a limitation that sometimes the program does not halt, or good performance is not achieved due to the many re-evaluation of the same subgoal. To address this issue, an alternative semantics called Well-Founded Semantics (Van Gelder, Ross, and Schlipf 1991) was proposed, in which the program is allowed to use *tabled predicates*. When the program declares a certain predicate to be *tabled*, results of achieving compound terms of the same predicate are memoized into a table and it succeeds without recursion when the same subgoal is tested next time.

A related subset of Prolog called Datalog is a much smaller subset. Instead, Prolog + Tabling is an extension of Prolog with Datalog-like efficiency.

High Performance Modern Prolog Prolog language is standardized as ISO-Prolog (Covington 1993) and many Prolog implementations (commercial / open sourced) with various focuses are available. Prolog interpreters typically process a program with a virtual machine called Warren's Abstract Machines (Warren 1985, WAM), which is heavily tuned for optimized execution of unification. While the most popular implementation is SWI-Prolog (Wielemaker et al. 2012), it's focus is the large feature set rather than the performance. In our project, we use B-Prolog (Zhou 2012) which has shown the best performance in our internal testing. B-Prolog is an originally commercial implementation which is now in public domain, and it supports tabled predicates. Typically, B-Prolog is faster than SWI by around x2, but for some corner cases by up to x120 faster (transport agl14 p01, SWI:242s, BProlog:4.4s, translate.py: 13s w/o invariant synthesis).

2.2 Preprocessing

In our planner, we reused the formal definitions described in Helmert (2009) for grounding facts/actions. We use binary formalizm instead of SAS formalism for simplicity, and thus does not perform mutex invariant synthesis, while this is future work.

The program is entirely written in Common Lisp (CL). After opening a PDDL input file, a parser written in CL parses the PDDL input, then programatically constructs a Prolog program using `cl-prolog2` (Asai 2017) library written by the author. The library makes it easy to use Prolog as a domain-specific solver by transpiling S-expressions (the same format used in PDDL: parentheses and symbols) into Prolog expressions, writes them to a file, runs a Prolog interpreter, then extracts the output.

3 Search Component

In this section, we describe the search component and the language used to implement the program, ANSI Common Lisp. In the search component, we generally follow the advice from Burns et al. (2012).

3.1 ANSI Common Lisp

ANSI Common Lisp (ANSI CL) is a *specification* of Common Lisp language, similar to C++14 or C++17. Just as C++14 has various implementations (GCC / Clang / MSVC), so does CL (SBCL / CCL / ECL / ABCL). Just as C++14 does not forbid implementing a C++ interpreter, ANSI CL does not specify if it is interpreted or compiled. **Due to historical mishaps, many misunderstand that lisp implementations are interpreters; In fact most CL implementations compile programs to native instruction sequences.** Alien uses Steel Bank Common Lisp (sbcl), the current fastest Common Lisp compiler on x86_64 environment.

Objects in Common Lisp are strongly typed and functions can be *optionally* statically typed via *declaration*. Typed functions typically compiles to a native code that is as good as code compiled by GCC. Similar to many other languages, or like `auto` keyword in C++, there is type inference mechanism and programmers should declare only a subset of variables. As typing is optional, programmers can choose to neglect it for non-performance-sensitive code, sacrificing speed for agile development.

A notable feature of ANSI CL is the inclusion of `compile` function in the standard library. That is, programmers are allowed to *compile a new code in runtime*, which allows us to generate code specifically optimized for the given problem instance / successor function / state representation / heuristic function.

Compilation of Common Lisp Programs Compilation of a Common Lisp program is quite different from those of traditional programming languages, and it allows flexibility and ultimate low-level performance.

In traditional programming languages, initially (1) there is a textual representation of the program in a file, (2) the compiler emits a compiled binary for a file, (3) the linker links the object files to produce an executable, (4) which is loaded onto main memory and the CPU runs the instructions. While it is possible to compile an additional source code while the program is running and load the object file from the running program, the process would be quite complicated. Thus, most programs are compiled off-line and is considered a fixed entity during execution.

Common Lisp has several differences from this paradigm. First, it is inherently interactive: There is a process that is always running in the background, and the compilation, linking, execution are all performed by this process.

Secondly, the compiler is separated into two controllable phases. A textual program is first parsed into a nested single-linked list structure, because the entire program is written in S-expression (Fig. 2). The compiler then compiles the linked list into an instruction sequence. Due to this separation, *programmers can systematically create a linked-list representing a certain program* and compile/execute it.

Common Lisp:

```
(defun factorial (x)
  (declare ((unsigned-byte 64) x))
  (if (= 0 x)
      1
      (* x (factorial (- x 1)))))
```

Equivalent C:

```
uint factorial(uint x){
    if (0 == x){
       return 1;
    }else{
       return x * factorial(1-x);
    }
}
```

Figure 2: Comparison of factorial implementation with Common Lisp and C.

3.2 Escaping GC for Close List

Common Lisp programs uses Garbage Collection (GC) for memory management as its inherently interactive nature allows the lifetime of certain objects to be unknown. However, GC is an expensive process: It should sweep over the entire memory, collecting and freeing the unreferenced objects. While it is acceptable to have many dead objects in performance-insensitive code such as preprocessing, object allocation inside a core inner loop is problematic, as it invokes GC and slows the entire program execution.

To completely avoid the problem of GC in inner loop, we store Close List in a large, separate memory array independently allocated by `malloc` and not managed by Lisp GC. This design choice is acceptable because in forward state-space search, close-list and other data structures are persistent, and need to be freed only when the program exits.

3.3 States and Per State Information

Memory layout of the Close List is determined after preprocessing and command line option parsing.

In Alien, states have binary representation. Unlike SAS formalism, the representation itself is not densely compressed. However, bit packing performed by FD is trivial in our case. Also, when a state has N propositional variables, it consumes exactly N bits in Close List, with a slight overhead of shifting some bits after reading the data from the array.

Since the layout is computed after the option parsing, per-state information such as heuristic cache or g-value can be placed right after each packed state, i.e. array-of-structures. This improves memory locality compared to Fast Downward, which has a separate array for each `PerStateInformation<T>`, i.e. structure-of-arrays (SoA) representation.

The number of bits consumed for such data is also optimized. For example, the maximum value of FF heuristics is bounded by the number of operators (maximum depth of an RPG), thus the cache takes exactly $\lceil \log |O| \rceil$ bits where $|O|$ is the number of operators.

3.4 Successor Generator as Assembly Sequence

Fast Downward uses Successor Generator (SG) to represent a successor function (Fig. 3). SG is a decision-tree whose internal node represents a precondition of an action and each node has multiple outgoing edges, one for each value in the domain of SAS variable, as well as a single don't-care edge. When generating a successor, the program recurses over this data structure, following the correct branch depending on the value of the variable in the current state, as well as following the don't care edge afterwards.

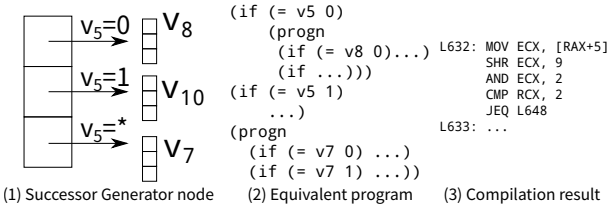

Figure 3: Successor Generator and its corresponding program and the compiled binary

While this achieves a better performance compared to a naive method which checks applicability of an action one by one, Alien improves it by converting a SG into a nested if-else program that is subsequently compiled into an X86_64 instruction sequence.

This approach has two advantages over the recursion to a SG. First, it creates a function that is loaded onto L1 instruction cache rather than a L1 data cache, minimizing the data cache usage. Second, it enables every built-in CPU features including pipelining, out-of-order execution and branch prediction. Thirdly, when building a program

for a SG, it could *merge* several preconditions into a single `if` statement which compiles to a single word-size test op, when their indices are within a 64bit boundary (Fig. 4).

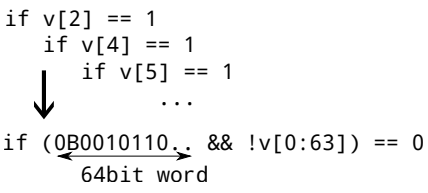

Figure 4: Packing nearby conditions into a single condition.

One issue with this compilation is that it takes time when the SG is large, and that the function may not fit in instruction cache. In the internal testing, compilation time increases quadratically to the number of branches. Therefore, we set a limit on the number of compiled decision nodes, and from the tip node that is not compiled, we process the remaining variables with a standard SG. The limit is heuristically determined to be 1000 nodes, which roughly keeps the function size within 20kB. For reference, Intel Haswell processor has 32kB of L1 instruction cache. Axiom evaluators and conditional effects are compiled similarly.

3.5 Heuristics and Other Search Code

After preprocessing, Alien recompiles the heuristic functions and search algorithms (e.g. eager best first search) being used. This optimizes the program by inlining the information such as the state size / number of operators.

We did not have time to implement various heuristic functions, and we have only FF (Hoffmann and Nebel 2001) heuristics based on RPG, as well as the novelty metric (Lipovetzky 2017). The planner which entered the competition is almost the same as BWFS presented in (Lipovetzky 2017).

3.6 Low-Level Performance

In our preliminary testing with blind search, Alien showed a better low-level performance compared to Fast Downward (Table 1).

In IPC2014 Agile track setting, Alien with eager FF heuristics (54 instances solved) slightly outperforms Fast Downward with FF heuristics with eager evaluation (44 instances solved).

problem	Fast Downward	Alien
sokoban p01	180095	307500
cavediving p01	255159	410255
citycar p01	178950	200229
parkprinter p01	264629	273645

Table 1: Node generation per second for Fast Downward and Alien on four easy problem instances.

4 Conclusion

We present Alien planner, which contains a new approach to write a preprocessor and the base search algorithm. While the framework is still immature, there are some notable design decisions that may also influence future planners. Future work includes the implementation of more heuristic functions, invariant synthesis, and connection to external services such as web services, online notebook or machine learning.

References

Asai, M., and Fukunaga, A. 2014. Fully Automated Cyclic Planning for Large-Scale Manufacturing Domains. In *Proc. of the International Conference on Automated Planning and Scheduling(ICAPS)*.

Asai, M., and Fukunaga, A. 2018. Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. In *Proc. of AAAI Conference on Artificial Intelligence*.

Asai, M. 2017. Cl-Prolog2 - Common Interface to the ISO Prolog implementations from Common Lisp. `github.com/guicho271828/cl-prolog2`.

Barski, C. 2007. Public domain lisp logo set. `lisperati.com/logo.html`.

Burns, E. A.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing Fast Heuristic Search Code. In *Proc. of Annual Symposium on Combinatorial Search*.

Covington, M. A. 1993. ISO Prolog: A Summary of the Draft Proposed Standard. `fsl.cs.illinois.edu/images/9/9c/PrologStandard.pdf`.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.(JAIR)* 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. *J. Artif. Intell. Res.(JAIR)* 14:253–302.

Jinnai, Y., and Fukunaga, A. 2017. On hash-based work distribution methods for parallel best-first search. *Journal of Artificial Intelligence Research* 60:491–548.

Lin, S., and Fukunaga, A. 2018. Revisiting Immediate Duplicate Detection in External Memory Search. In *Proc. of AAAI Conference on Artificial Intelligence*.

Lipovetzky, N. 2017. Best-First Width Search: Exploration and Exploitation in Classical Planning . In *Proc. of AAAI Conference on Artificial Intelligence*.

Van Gelder, A.; Ross, K. A.; and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)* 38(3):619–649.

Warren, D. 1985. An abstract Prolog instruction set. *SRI Technical Note*.

Wielemaker, J.; Schrijvers, T.; Triska, M.; and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12(1-2):67–96.

Zhou, N.-F. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming* 12(1-2):189–218.