Large-Scale Optimal PDDL3 Planning with MIPS-XXL

Stefan Edelkamp[1], Shahid Jabbar[2], and Mohammed Nazih[3] *
Computer Science Department
University of Dortmund, Dortmund, Germany

Introduction

State trajectory and preference constraints are the two language features introduced in PDDL3 (Gerevini & Long 2005) for describing benchmarks of the 5^{th} international planning competition. *State trajectory constraints* provide an important step of the agreed fragment of PDDL towards the description of *temporal control knowledge* and *temporally extended goals* They assert conditions that must be met during the execution of a plan and are often expressed using quantification over domain objects.

We suggest to compile the state trajectory and preference constraints into PDDL2 (Edelkamp 2006). Trajectory constraints are compiled into Büchi automata that are synchronized with the exploration of the planning problem, while preference constraints are transformed into numerical fluents that are changed upon violation. An internal weighted best-first search is invoked that tries to find a solution. Once a solution is found, the solution quality is inserted in the problem description and a new search is started using earlier solution cost as the minimization parameter. If the internal search fails to terminate with in a specified amount of time, we switch to a cost-optimal external breadth-first search procedure that utilizes harddisk to store the generated states.

Compilation of State Trajectory Constraints

State trajectory constraints impose restrictions on plans. Their semantics can best be captured by using a special kind of automata structure called as Büchi automata. Büchi automata has long been used in automata-based model checking (Clarke, Grumberg, & Peled 2000), where both the model to be analyzed and the specification to be checked are modeled as nondeterministic *Büchi automata*. Syntactically, Büchi automata are ordinary automata, but with a special acceptance condition. Let ρ be a run and $inf(\rho)$ be the set of states reached infinitely often in ρ, then a Büchi automaton accepts, if the intersection between $inf(\rho)$ and the set of final states F is not empty. In automata-based model-checking, a specification property is falsified if and only if there is a non-empty intersection between the language accepted by the Büchi automata of the model and of the *negated* specification.

For trajectory constraints, we need a Büchi automaton for the model and one for each trajectory constraints, together with some algorithm that validates if the language intersection is not empty. By the semantics of (Gerevini & Long 2005) it is clear that all sequences are finite, so that we can interpret a Büchi automaton as a non-deterministic finite state automaton (NFA), which accepts a word if it *terminates* in a final state. The labels of such an automaton are conditions over the propositions and fluents in a given state. During the exploration, we simulate a synchronization of all Büchi automata.

To encode the simulation of the synchronized automata, we devise a predicate `(at ?n - state ?a - automata)` to be instantiated for each automata state and each automata that has been devised. For detecting accepting states, we include instantiations of predicate `(accepting ?a - automata)`.

As we require a tight synchronization between the constraint automaton transitions and the operators in the original planning space, we include *synchronization flags* that are flipped when an ordinary or a constraint automaton transition is chosen.

Compilation of Preferences

For preference p we include numerical fluents `is-violated-p` to the grounded domain description. For each operator and each preference we apply the following reasoning. If the preferred predicate is contained in the *delete list* then the fluent is increased, if it is contained in the *add list*, then the fluent is decreased, otherwise it remains unchanged[1].

*All three authors are supported by the German Research Foundation (DFG) projects *Heuristic Search* Ed 74/3 and *Directed Model Checking* Ed 74/2.

[1]An alternative semantic to (Gerevini & Long 2005) would be to set the fluent to either 0 or 1. For rather complex propositional or numerical goal conditions in a preference condition, we can use *conditional effects*.

For preferences p on a state trajectory constraint that has been compiled to an automaton a, the fluents `(is-violated-a-p)` substitute the atoms `(is-accepting-a)` in an obvious way. If the automata accepts, the preference is fulfilled, so the value of `(is-violated-a-p)` is set to 0. In the transition that newly reaches an accepting state `(is-violated-a-p)` is set to 0, if it enters a non-accepting state it is set to 1. The `skip` operator also induces a cost of 1 and the automaton moves to a dead state.

External Exploration

For complex planning problems, the size of the state space can easily surpass the main memory limits. Most modern operating systems provides a facility to use larger address spaces through *virtual memory* that can be larger than internal memory. For the programs that do not exhibit any *locality of reference* for memory accesses, such general purpose virtual memory management can instead lower down their performances.

Algorithms that explicitly manage the memory hierarchy can lead to substantial speedups, since they are more informed to predict and adjust future memory access. In (Korf & Schultze 2005) we see a complete exploration of the state space of 15-puzzle made possible utilizing a 1.4 Terabytes of secondary storage. In (Jabbar & Edelkamp 2005) a successful application of external memory heuristic search for LTL model checking is presented.

The standard model (Aggarwal & Vitter 1988) for comparing the performance of external algorithms consists of a single processor, a small internal memory that can hold up to M data items, and an unlimited secondary memory. The size of the input problem (in terms of the number of records) is abbreviated by N. Moreover, the *block size* B governs the bandwidth of memory transfers. External-memory algorithms are evaluated in terms of number of I/Os, where each block transfer amounts to one I/O.

It is convenient to express the complexity of external-memory algorithms using a number of frequently occurring primitive operations: *Scanning*, *scan(N)* with an I/O complexity of $\Theta(\frac{N}{B})$ that can be achieved through trivial sequential access; *Sorting*, *sort(N)* with an I/O complexity of $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ that can be achieved through external *Merge* or *Distribution Sort*.

Cost-Optimal External BFS

An implicit variant of Munagala and Ranade's algorithm (Munagala & Ranade 1999) for explicit BFS-search in implicit graphs has been coined to the term *delayed duplicate detection* for *frontier search*. It assumes an undirected search graph. Let $\mathcal{I}$ be the initial state, and N be the implicit successor generation function. Figure 1 displays the pseudo-code for external BFS exploration incrementally improving an upper bound U on the solution quality. The state sets corresponding to each layer are represented in form of files.

Procedure Cost-Optimal-External-BFS
$U \leftarrow \infty$; $i \leftarrow 1$
$Open(-1) \leftarrow \emptyset$; $Open(0) \leftarrow \{\mathcal{I}\}$
while $(Open(i-1) \neq \emptyset)$
  $A(i) \leftarrow N(Open(i-1))$
  forall $v \in A(i)$
    if $v \in \mathcal{G}$ **and** $Metric(v) < U$
      $U \leftarrow Metric(v)$
      $ConstructSolution(v)$
  $A'(i) \leftarrow$ *remove duplicates from* $A(i)$
  for $l \leftarrow 1$ **to** *loc*
    $A'(i) \leftarrow A'(i) \setminus Open(i-l)$
  $Open(i) \leftarrow A'(i)$
  $i \leftarrow i+1$

Figure 1: Cost-Optimal External BFS Planning Algorithm.

The search frontier denoting the current BFS layer is tested for an intersection with the goal, and this intersection is further reduced according to the already established bound.

Layer $Open(i-1)$ is scanned and the set of successors are put into a buffer of size close to the main memory capacity. If the buffer becomes full, internal sorting followed by a duplicate elimination scanning phase generates a sorted duplicate-free state sequence in the buffer that is flushed to disk. A sets in the pseudo-code corresponds to temporary sets.

In the next step, *external merging* is applied to merge the flushed buffers into $Open(i)$ by a simultaneous scan. The size of the output files is chosen such that a single pass suffices. Duplicates are eliminated while merging. Since the files were sorted, the complexity is given by the scanning time of all files. One also has to eliminate the previous layers from $Open(i)$ to avoid recomputations. The number of previous layers that have to be subtracted are dependent on the *locality*(loc) of the graph. In case of undirected graphs, two layers are sufficient. For directed graphs, we suggest to calculate this parameter by searching for a sequence of operators that when applied to a state produces no effect. Such a sequence can be computed by just looking at all possible sequences of operators. The length of the shortest such sequence dictates the locality of a planning graph. The process is repeated until $Open(i-1)$ becomes empty, or the goal has been found.

The I/O Complexity of External BFS for undirected graph can be computed as follows. The successor generation and merging involves $O(sort(|N(Open(i-1))|) + (\sum_{l=1}^{loc} scan(|Open(i-l)|)$ I/Os. However, since $\sum_i |N(Open(i))| = O(|E|)$ and $\sum_i |Open(i)| = O(|V|)$, the total execution time is $O(sort(|E|) + loc \cdot scan(|V|))$ I/Os.

In an internal non memory-limited setting, a plan is constructed by backtracking from the goal node to the start node. This is facilitated by saving with every

node a pointer to its predecessor. However, there is one subtle problem: predecessor pointers are not available on disk. This is resolved as follows. Plans are reconstructed by saving the predecessor together with every state, by using backtracking along the stored files, and by looking for matching predecessors. This results in a I/O complexity that is at most linear to the number of stored states.

In planning with preferences, we often have a monotone decreasing instead of a monotonic increasing cost function. Hence, we cannot prune states with an evaluation larger than the current one. Essentially, we are forced to look at all states. In order to speed up the external search with a compromise on the optimality, we can apply a procedure similar to beam-search where we can limit our search to expand only a small portion of the best nodes within each layer. On competition problems, we have managed to have good accelerations through this approach.

Implementation

We first transform PDDL3 files with preferences and state trajectory constraints to grounded PDDL3 files without them. For each state trajectory constraint, we parse its specification, flatten the quantifiers and write the corresponding LTL-formula to disk.

Then, we derive a Büchi-automaton for each LTL formula and generates the corresponding PDDL code to modify the grounded domain description[2]. Next, we merge the PDDL descriptions corresponding to Bčhi automata and the problem file. Given the grounded PDDL2 outcome, we apply efficient heuristic search forward chaining planner *Metric-FF* (Hoffmann 2003). Note that by translating plan preferences, otherwise propositional problems are compiled into metric ones. For temporal domains, we extended the *Metric-FF* planner to handle temporal operators and timed initial literals. The resulting planner is slightly different from known state-of-the-art systems of adequate expressiveness, as it can deal with disjunctive action time windows and uses an internal linear-time approximate scheduler to derive parallel (partial or complete) plans. The planner is capable of compiling and producing plans for all competition benchmark domains.

Due to the numerical fluents introduced for preferences, we are faced with a search space where cost is not necessarily monotone. For such state spaces, we have to look at all the states to reach to an optimal solution. The issue then arises is if it is possible to reach an optimal solution fast. We propose to use a branch-and-bound like procedure on top of the best-first weighted heuristic search as offered by the extended *Metric-FF* planning system. Upon reaching a goal, we terminate our search and create a new problem file where the goal condition is extended to minimize the found solution cost. The search is restarted on this new problem description. The procedure terminates when the whole state space is looked at. The rationale behind this is to have improved guidance towards a better solution quality. If internal search failed to terminate within a specified amount of time, we switch to external BFS search.

[2] `www.liafa.jussieu.fr/~oddoux/ltl2ba`. Similar tools include *LTL→NBA* and the never-claim converter inherent to the SPIN model checker.

Conclusions

We propose to translate temporal and preference constraints into PDDL2. Temporal constraints are converted into Büchi automata in PDDL format, and are executed synchronously with the main exploration. Preferences are compiled away by a transformation into numerical fluents that impose a penalty upon violation. Incorporating better heuristic guidance, especially, for preferences is still an open research frontier.

Search is performed in two stages. Initially, an internal best-first is invoked that keeps on improving its solution quality till the search space is exhausted. After a given time limit, the internal search is terminated and an external breadth-first search is started.

The crucial problem in external memory algorithms is the duplicate detection with respect to previous layers to guarantee termination. Using the locality of the graph calculated directly from the operators themselves, we provide a bound on the number of previous layers that have to be looked at.

Since states are kept on disk, external algorithms have a large potential for parallelization. We noticed that most of the execution time is consumed while calculating heuristic estimates. Distributing a layer on multiple processors can distribute the internal load without having any effect on the I/O complexity.

References

Aggarwal, A., and Vitter, J. S. 1988. The input/output complexity of sorting and related problems. *Journal of the ACM* 31(9):1116–1127.

Clarke, E.; Grumberg, O.; and Peled, D. 2000. *Model Checking*. MIT Press.

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *ICAPS*. To Appear.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences for PDDL3. Technical Report R.T. 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy.

Hoffmann, J. 2003. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *JAIR* 20:291–341.

Jabbar, S., and Edelkamp, S. 2005. I/O efficient directed model checking. In *Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 313–329.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *AAAI*, 1380–1385.

Munagala, K., and Ranade, A. 1999. I/O-complexity of graph algorithms. In *SODA*, 687 – 694.