Cost-Optimal Planning with Constraints and Preferences in Large State Spaces

Stefan Edelkamp[1], **Shahid Jabbar**[2], **and Mohammed Nazih**[3] *
Computer Science Department
University of Dortmund, Dortmund, Germany
[1]`stefan.edelkamp@cs.uni-dortmund.de`
[2]`shahid.jabbar@cs.uni-dortmund.de`
[3]`mohammed.nazih@uni-dortmund.de`

Abstract

This paper deals with planning in the presence of constraints and preferences as proposed for the 5th International Planning Competition. State trajectory constraints are translated into LTL formulae and are compiled into Büchi automata in PDDL format. Preference constraints are compiled into numerical fluents. Values of these fluents are changed by grounded operator effects upon violation.

We propose two exploration strategies for optimal planning in PDDL3 domains: (i) a best-first branch-and-bound weighted heuristic search; (ii) an external breadth-first search exploration algorithm that exploits secondary memory, such as harddisk, to save the *open* and *closed* lists. We prove an upper bound on the locality of the search in planning graphs that dictates the number of layers that have to be kept to avoid re-openings. For non-optimal planning, we present an external variant of enforced hill climbing.

Introduction

In recent years, AI Planning has seen significant growth in both theory and practice. PDDL (Planning Domain Description Language) provides a common framework to define planning domains and problems. Starting from a pure propositional framework, it has now grown into accommodating more complex planning problems. In Metric planning, we see a numerical extension to the STRIPS planning formalism, where actions can contribute an increase or decrease of numeric variables. The task is then to find a path from an initial state to a state where all goal criteria are fulfilled, *additionally*, the values of a set of numeric variables are minimized (or maximized).

State trajectory and preference constraints are the two language features introduced in PDDL3 (Gerevini & Long 2005) for describing benchmarks of the 5th international planning competition. *State trajectory constraints* provide an important step of the agreed fragment of PDDL towards the description of *temporal control knowledge* (Bacchus & Kabanza 2000; Kabanza & Thiebaux 2005) and *temporally extended goals* (DeGiacomo & Vardi 1999; Lago, Pistore, & Traverso 2002; Pistore & Traverso 2001). They assert conditions that must be met during the execution of a plan and are often expressed using quantification over domain objects. Through the decomposition of metric and temporal plans into happenings, state trajectory constraints also feature higher levels of the PDDL hierarchy (Fox & Long 2003).

Unfortunately, as the planning problems get complicated, the size of the state and the number of states grow significantly too - easily reaching the limits of main memory capacity. Having a systematic mechanism to flush the already seen states to the disk can circumvent the problem. Algorithms that utilize secondary storage devices have seen significant success in single-agent search. 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. (Zhou & Hansen 2004b) proposed structured duplicate detection for external search, where the state space structure was exploited to define a partition on the state space. Among the reported results are applications on STRIPS planning problems.

The paper is structured as follows: We first discuss the planning problem with temporal and preference constraints as proposed for the 5th International Planning Competition. The temporal constraints are compiled into Büchi automata that are synchronized with the exploration of the planning problem, while preference constraints are transformed into numerical fluents. Upon violation, a penalty cost is imposed to the corresponding fluent. We then discuss some implementation details for the realization of this compilation procedure. An overview of the external memory model is presented afterward. Then, we introduce external Breadth-First Search for implicit undirected graphs. Directed graphs are treated in the next section, where we discuss a formal basis to determine the locality of planning graphs which dictates the number

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

of previous layers to look at during duplicates removal to avoid re-expansions (Edelkamp & Jabbar 2006). For non-optimal planning, we discuss an external memory variant of enforced hill climbing.

State Trajectory Constraints

We briefly recall *automata-based model checking*, a common approach for model checking of softwares.

Automata-based Model Checking

In automata-based model checking both the model to be analyzed and the specification to be checked are modeled as non-deterministic *Büchi automata*. Syntactically, Büchi automata are ordinary automata. For accepting *infinite words*, or *runs*, a different acceptance condition is applied. Let ρ be a run and $\mathit{inf}(\rho)$ be the set of states reached infinitely often in ρ, then a Büchi automaton accepts, if the intersection between $\mathit{inf}(\rho)$ and the set of final states F is not empty.

The desired property of the system is specified in some form of temporal logic. We briefly introduce *linear temporal logic (LTL)*. A path in model $\mathcal{M}$ is a sequence of states $\pi = S_0, S_1, \ldots$ and π^i denotes the suffix of π starting at S_i. LTL formulae have the form "Always f", where f is a *path formula*. If p is an atomic proposition then p is a path formula. If f and g are path formulae so are $\neg f, f \vee g, f \wedge g, \mathbf{X}\ f, \mathbf{F}\ f, \mathbf{G}\ f$, and $f\ \mathbf{U}\ g$.

For the *next time* operator $\mathbf{X}$ we have $M, \pi \models \mathbf{X}\ f \Leftrightarrow M, \pi^1 \models f$. For the *until* operator $g\ \mathbf{U}\ f$ we have $M, \pi \models g\ \mathbf{U}\ f \Leftrightarrow \exists 0 \leq k : M, \pi^k \models f \wedge \exists k \leq j : M, \pi^j \models g$, for the *eventually* operator we have $M, \pi \models \mathbf{F}\ f \Leftrightarrow \exists 0 \leq k : M, \pi^k \models f$, and for the *globally* operator we have $M, \pi \models \mathbf{G}\ f \Leftrightarrow \forall 0 \leq k : M, \pi^k \models f$.

Transforming the model and the specification into Büchi automata assumes that the system can be modeled by a deterministic finite state machine, and that the LTL formula can be transformed into an equivalent Büchi automaton. The contrary is not always possible, since Büchi automata are clearly more expressive than LTL expressions (Wolper 1983). Checking correctness is reduced to checking language emptiness. More formally, the model checking procedure validates that a model represented by an automaton $\mathcal{M}$ satisfies its specification represented by an automaton $\mathcal{S}$. The task is to verify if $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S})$. In words: the *language accepted by the model is included in that of the specification.* We have $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S})$ if and only if $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$.

Büchi automata are closed under intersection and complementation (Buchi 1962), so that there exists an automaton that accepts $\overline{\mathcal{L}(\mathcal{S})}$ and an automata that accepts $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})}$.

It is possible to complement a Büchi automaton, *but* the worst-case running time of such a construction is double-exponential in the size of the formula. Therefore, in practice, one constructs the automaton for negation of the LTL formula, avoiding complementation.

The product is *synchronous*, that is, each transition in one automata implies one in the other. The property automaton is non-deterministic, such that both the successor generation and the temporal formula representation may introduce branching to the overall exploration module. The construction assumes that all states in the model are accepting.

Application to Temporal Plan Constraints

In the proposed extension to planning we do not have to negate the property formula. Planning goals already correspond to the negations of properties in model checking. If ordinary goals without temporal modalities are used, we add their satisfaction to the acceptance condition of the model. For state trajectory constraints ϕ, we search for a witness in $L(\mathcal{M}) \cap L(\phi) \neq \emptyset$, where $\mathcal{M}$ is the original plan space.

For the exploration we, therefore, need a Büchi automaton for the model and one for the trajectory constraint, 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. We will illustrate how these conditions can be modeled using planning operators. There are some important observations to be made:

1. It is well known that an NFA can be transformed into an equivalent deterministic one using a power set construction (Hopcroft & Ullman 2000). This DFA, however, can become exponentially large, so that in most cases a simulation of the NFA is preferable.
2. Most state trajectory constraints are universally quantified. The quantified expressions can be unrolled. This is always possible as the scope of the quantified object variables is finite.
3. As the union of the conditions of all outgoing transitions is not always trivial, synchronizing the planning model with the automata of state trajectory constraint may also prune the exploration.

Examples

In PDDL3, the constraint *a fragile block can never have something above it* is expressed as

```
(always (forall (?b - block)
          (implies (fragile ?b) (clear ?b))
```

We call this condition an *always/every* constraint. The LTL formula for two selected blocks a and b is

```
[] ((fragile_a -> clear_a) &&
    (fragile_b -> clear_b))
```

The corresponding automaton is shown in Fig. 1[1]. The

[1]The automata in the figures are constructed automatically using the LTL to Büchi automaton converter. The in-

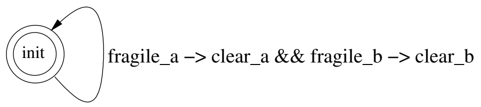

Figure 1: Automaton for the *always/every* constraint with a transition in clause form.

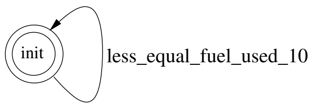

Figure 2: Automaton for numeric constraint.

automaton consists of only one state. Numeric conditions as generated, e.g., by `(always (<= (fuel-used) 10))` do not lead to additional expressiveness as for the translation process they are interpreted as integral propositions to be combined in transition labels. Fig. 2 provides a simple example. When applying such an automata, the construction has to be followed by a backward translation of edges to numeric conditions.

The assertion *each block should be put on the table at least once* corresponds to

```
(forall (?b - block)(sometime (ontable ?b)))
```

called an *every/sometime* constraint. For two blocks a Büchi automaton with respect to the LTL formula `(<> ontable_a) && (<> ontable_b)` is constructed. It is shown in Fig. 3 (top right).

The statement *in some state visited by the plan all blocks are on the table* is expressed as

```
(sometime (forall (?b - block) (ontable ?b))
```

denoted as *sometime/every* constraint. The according LTL formula is `<> (ontable_a && ontable_b)` with a Büchi automata shown in Fig. 3 (top left). It is much simpler than the previous one.

The expression *each truck should visit each city at most once* is given by the constraint

```
  (forall (?t - truck ?c - city)
          (at-most-once (at ?t ?c)))
```

We use a simple instantiation with one truck and one city, yielding the LTL formula (cf. (Gerevini & Long 2005))

```
[](at_truck_a_city_a->
  (at_truck_a_city_a U ([]!at_truck_a_city_a)))
```

The corresponding Büchi-Automaton is displayed in Fig. 3 (bottom). The translation from a PDDL constraint to a Büchi automata is not lossless. This is due to the fact that PDDL constraints are defined over finite runs while Büchi automata are defined over infinite runs. It is not possible to capture the exact semantics

termediate LTL notation satisfies the syntax of *SPIN* model checker; *always* is denoted as `[]` and eventually as `<>`. We avoid hyphens in declaring the propositions as they are misinterpreted by the converter. They are re-introduced when generating the translated PDDL description.

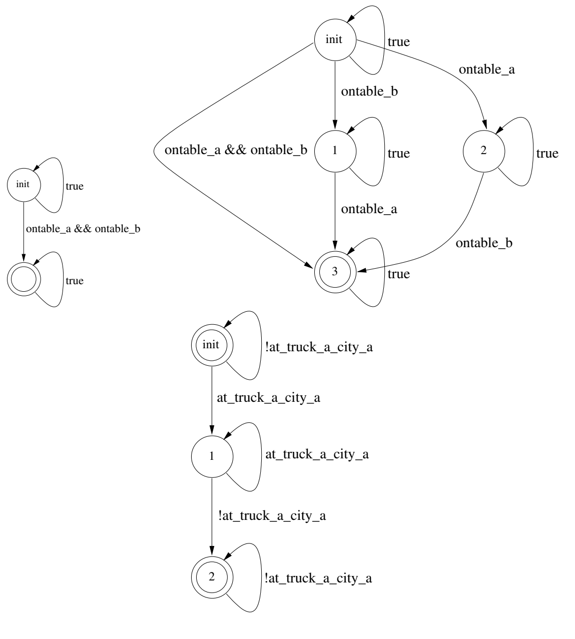

Figure 3: Büchi automata for the *sometime/every* constraint (top left), *every/sometime* constraint (top right) and *exactly-once* constraint (bottom).

of, e.g., `at-most-once` ϕ, when the plan consists of just one action and ϕ holds in the initial state as well as in the goal state. In our previous example, this implies that we will terminate at the state 1 of the automata and will claim that the constraint is not satisfied.

Instead of deriving one automata for both constraints in common, simulating two synchronized automata, one for each constraint, is equivalent. Hence, we observe a trade-off between the size of the automata for one constraint and the maintenance of several concurrent automata.

Preferences

Preferences model soft constraints that are desirable but do not have to be fulfilled in a valid plan. The degree of desirability of satisfying a preference constraint is specified in the plan metric.

Simple Preferences

Simple preferences refer to ordinary propositions (and are included to the planning goal). E.g., if we prefer block *a* to reside on the table during the plan execution, we write `(preference p (on-table a))` with a validity check `(is-violated p)` in the plan objective. Such checks are interpreted as natural numbers that can be scaled and combined with other variable assignments in the plan metric. To evaluate the costs for a given plan, we have to accumulate how often the stated preference condition is violated in the preconditions of the actions in the plan. The according numerical value is substituted in the metric for its evaluation. Quantified preference constraints like

```
(forall (?b - block) (preference p (clear ?b))
```

are flattened to multiple instantiated preference conditions (one for each block), while the inverse expression

```
(preference p (forall (?b - block) (clear ?b))
```

leads to only one constraint.

Preference on Temporal Plan Constraints

Preferences for state trajectory constraints like

```
(preference cleaned
   (forall (?t truck) (always (clean ?t))))
```

can, in principle, also be dealt with automata theory. Instead of requiring to reach an accepting state we *prefer* to be there, by means that not arriving at an accepting state incurs costs to the evaluation of the plan metric using the `(is-violated cleaned)` variable.

Language Compilation

Have the two language extensions enriched the PDDL language or is it possible to translate the new constructs away? Fortunately, we can show how to implement a language compilation from PDDL3 to PDDL2.

Temporal Plan Constraints

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)`. The initial state of the planning problem includes the start state of the automata and an additional proposition if it is accepting. For all automata, the goal includes their acceptance.

Next, we have to specify allowed automata transitions in form of planning actions. This is done by declaring a grounded operator for each automata transition, with the current automaton state and the transition label as preconditions, as well as the current automaton state as the delete and the successor state as the add effect. For transition leading to an accepting state, we include the corresponding automata acceptance proposition to the add effects. 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. An example for a grounded transition is

```
(:action sync-trans-a-0-init-a-0-accept-0
:precondition
     (and (at-a-0-init) (sync-automaton-a-0)
          (in-package1-truck2))
:effect
     (and (accepting-a-0) (not (at-a-0-init))
          (at-a-0-accept-0) (not (sync-automaton-a-0))
          (sync-ordinary)))
```

As said, the size of the Büchi automaton for a given formula can be exponential in the length of the formula[2]. In practice, the size of the automaton is often small compared to the size of the (grounded) model.

Metric Time Constraints

So far we have only seen how to derive automata for the *untimed plan constraints* `sometime`, `always`, `at-most-once`. Fortunately, (Gerevini & Long 2005) show that `sometime-before` and `sometime-after` can be expressed using standard LTL expressions, so these modalities easily fit into the above framework. For *metric time constraints* like `within`, `always-within`, `hold-during`, and `hold-after` we have to restrict actions to the execution time window specified in the constraints. Moreover, these constraints necessarily call for parallel/temporal planning, as they refer to absolute points in time for plan execution.

These expressions can be tackled using *timed initial literals* as already contained in the language PDDL2 (Hoffmann & Edelkamp 2005). Timed initial literals denote fixed *dates* in plan time in which an atom is *true* or *false*. As they are only allowed to be checked in operators' preconditions, they correspond to *action execution time windows*. The modalities `hold-after`, and `hold-during` immediately translate to timed initial literals for the operators, in which the stated conditions are satisfied in the preconditions. If the state formula is disjunctive the planner has to deal with multiple action windows.

For combined metric and temporal modalities as in `within` and `always-within` action execution time window are included in form of additional timed initial literal for the preconditions of the automata's transitions.

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[3].

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.

[2]In the notion of *essentiality* (Nebel 2000), which provides a complexity theory for domain compilations, the compilation is *essential*. Similar essential compilations have been proposed by (Gazen & Knoblock 1997) for ADL to STRIPS and by (Thiebaux, Hoffmann, & Nebel 2005) for domain axioms.

[3]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*.

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[4]. Next, we merge the PDDL descriptions corresponding to Bc̈hi 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 neccessarily 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 fails to terminate within a specified amount of time, we switch to External Breadth-First search (BFS).

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. When the program is executed, virtual addresses are translated into physical addresses. Only those portions of the program currently needed for the execution are copied into main memory; the rest stays on the harddisk. 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.

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

Algorithms that explicitly manage the memory hierarchy can lead to substantial speedups, since they are more informed to predict and adjust future memory access. The standard model 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. Only the number of block reads and writes are counted, computations in internal memory do not incur any cost. The single disk model for external algorithms has been devised by (Aggarwal & Vitter 1988).

It is convenient to express the complexity of external-memory algorithms using a number of frequently occurring primitive operations. Here D represents the number of disks that can be accessed simultaneously.

1. *Scanning*: *scan(N)* with an I/O complexity of $\Theta(\frac{N}{DB})$ that can be achieved through trivial sequential access.
2. *Sorting*: *sort(N)* with an I/O complexity of $\Theta(\frac{N}{DB}\log_{M/B}\frac{N}{B})$ that can be achieved through *Merge* or *Distribution Sort*

External Breadth-First Search in Undirected Graphs

Munagala and Ranade's algorithm (Munagala & Ranade 1999) for explicit Breadth-First Search has been adapted for implicit graphs. The new algorithm is known as *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. The algorithm maintains BFS layers on disk. Layer $\mathit{Open}(i-1)$ 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.

In the next step, *external merging/sorting* is applied to remove duplicates in the flushed buffers. This results in a duplicates-free sorted file corresponding to $\mathit{Open}(i)$. One also has to eliminate $\mathit{Open}(i-1)$ and $\mathit{Open}(i-2)$ from $\mathit{Open}(i)$ to avoid re-expansions; that is, nodes extracted from the external queue are not immediately deleted, but kept until after the layer has been completely generated and sorted, at which point duplicates can be eliminated using a parallel scan. The process is repeated until $\mathit{Open}(i-1)$ becomes empty, or the goal has been found.

The corresponding pseudo-code is shown in Figure 4. A-sets in the algorithm correspond to temporary files. Termination is not shown, but imposes no additional overhead. As with the algorithm of Munagala and Ranade, delayed duplicate detection applies

Procedure External-BFS

```
$Open(-1) \leftarrow \emptyset$, $Open(0) \leftarrow \{\mathcal{I}\}$
$i \leftarrow 1$
while $(Open(i-1) \neq \emptyset)$
    $A(i) \leftarrow N(Open(i-1))$
    $A'(i) \leftarrow$ remove duplicates from $A(i)$
    $Open(i) \leftarrow A'(i) \setminus (Open(i-1) \cup Open(i-2))$
    $i \leftarrow i+1$
```

Figure 4: Delayed duplicate detection in BFS.

$O(sort(|N(Open(i-1))|)+scan(|Open(i-1)|+|Open(i-2)|))$ I/Os. However, since no explicit access to the adjacency list is needed, by $\sum_i |N(Open(i))| = O(|E|)$ and $\sum_i |Open(i)| = O(|V|)$, the total execution time is $O(sort(|E|) + scan(|V|))$ I/Os.

Locality in Planning Domains

How many layers are sufficient for full duplicate detection in general is dependent on a property of the search graph called *locality*. For integer weighted problem graphs, it is defined as the maximum $\max\{\delta(s,u)-\delta(s,v),0\}$ of all nodes u,v, with v being a successor of u and δ the shorted path distance. For undirected graphs we always have that $\delta(s,u)$ and $\delta(s,v)$ differ by at most one so that the locality is 1. The locality determines the *thickness* of the boundary slice of the graph needed to prevent duplicates.

Let l be the graph's locality, and k the number of stored layers. In breadth-first search, when layer m is expanded, all previous layers with g-value smaller than m have been closed, and are known by their optimal g-value. Thus, if a node u at level m is expanded, and its successor v has a shorter optimal distance to s, i.e., $m = \delta(s,v) < \delta(s,u) = m'$, then v must have been encountered earlier in the search, in the worst case at layer $m' = m-l$. The re-generation of v will be avoided if and only if it is contained in the stored layers $m-k \dots m-1$; i.e., if and only if $k \geq l$. This is the basis of the following theorem due to (Zhou & Hansen 2004a)

Theorem 1 *(Locality Determines Boundary) The number of previous layers of a breadth-first search graph that need to be retained to prevent duplicate search effort is equal to the locality of the search graph.*

As a special case, in undirected graphs, the locality is 1 and we need to store the immediate previous layer only to check for duplicates.

The condition $\max\{\delta(s,u)-\delta(s,v),0\}$ over all nodes u,v, with v being a successor of u is not a graph property. So the question is if we can find a sufficient condition or upper bound for it.

Theorem 2 *(Upper-Bound on Locality) The locality of a uniformly weighted graph for breadth-first search can be bounded by the minimal distance to get back from a successor node v to u, maximized over all u. In other words, with Γ representing the set of successors, we have*

$$\max_{u,v\in\Gamma(u)}\{\delta(v,u)\} \geq \max_{u,v\in\Gamma(u)}\{\delta(s,u)-\delta(s,v),0\}$$

Proof: For any nodes s,u,v in a graph the triangular property of shortest path $\delta(s,u) \leq \delta(s,v)+\delta(v,u)$ is satisfied, in particular for s being the start node of the BFS and $v \in \Gamma(u)$. Therefore $\delta(v,u) \geq \delta(s,u)-\delta(s,v)$ and $\max_{u,v\in\Gamma(u)}\{\delta(v,u)\} \geq \max_{u,v\in\Gamma(u)}\{\delta(s,u)-\delta(s,v)\}$. In positively weighted graphs we have $\delta(v,u) \geq 0$ such that $\max_{u,v\in\Gamma(u)}\{\delta(v,u)\}$ is larger than the locality.

As for graphs without self-loops we have $\max_{u,v\in\Gamma(u)}\{\delta(v,u)\} = \max_u\{\delta(u,u)\}-1$, in order to bound the locality we have to look for largest minimal cycles in the graph.

The question then arises is: How can we decide the condition in an implicitly given graph as they appear in action planning? In the following we provide an answer to this question based on the rules or operators involved in a state space. Without loss of generality, we consider STRIPS planning operators in the form of $\langle pre(O) add(O), del(O)\rangle$, representing preconditions, add, and delete lists for an operator O. A duplicate node in an implicit graph appears when a sequence of operators, applied to a state generate the same state again, i.e., they cancel the effects of each other. Hence the following definition:

Definition 1 *(*no-op *Sequence) A sequence of operators $O_1, O_2, \dots, O_k$ is a* no-op sequence *if its application on a state produces no effects, i.e, $O_k \circ \dots \circ O_2 \circ O_1 =$* no-op*,*

This definition provides us the basis to bound the locality of the implicit graphs in the following theorem. It generalizes undirected search spaces, in which for each operator O_1 we find an inverse operator O_2 such that $O_2 \circ O_1 =$ *no-op.*

Theorem 3 *(*no-op *Sequence determines Locality) Let $\mathcal{O}$ be the set of operators in the search space and $l = |\mathcal{O}|$. If for all operators O_1 we can provide a sequence $O_2, \dots, O_k$ with $O_k \circ \dots \circ O_2 \circ O_1 =$* no-op*, where* no-op *is the identity mapping, then the locality of the implicitly generated graph is at most $k-1$.*

Proof: If $O_k \circ \dots \circ O_2 \circ O_1 =$ *no-op* we can reach each state u again in at most k steps. This implies that $\max_u\{\delta(u,u)\} = k$. Theorem 2 shows that $\max_u\{\delta(u,u)\}-1$ is an upper bound on the locality.

The condition $O_k \circ \dots \circ O_2 \circ O_1 =$ *no-op* can be tested in $O(l^k)$ time. It suffices to check that the cumulative add effects of the sequence is equal to the cumulative delete effects. Using the denotation by (Haslum & Jonsson 2000), the cumulative add C_A and delete C_D effects of a sequence can be defined inductively as,

$$C_A(O_k) = A_k \qquad C_D(O_k) = D_k \text{ and,}$$

$$\begin{aligned} C_A(O_1,\dots,O_k) &= (C_A(O_1,\dots,O_{k-1})-D_k)\cup A_k \\ C_D(O_1,\dots,O_k) &= (C_D(O_1,\dots,O_{k-1})-A_k)\cup D_k \end{aligned}$$

```
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 $loc \leftarrow 1$ to $locality$
      $A'(i) \leftarrow A'(i) \setminus Open(i - loc)$
    $Open(i) \leftarrow A'(i)$
    $i \leftarrow i + 1$
```

Figure 5: Cost-Optimal External BFS Planning. $\mathcal{G}$ is the set of goals and U the best goal cost found.

Theorem 3 gives us the missing link to the successful application of external breadth first search in planning. Subtracting k previous layer *plus* the current layer from the successor list in an external breadth-first search guarantees its termination on finite planning graphs.

Cost-Optimal External BFS

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.

Figure 5 displays the pseudo-code for external BFS exploration incrementally improving an upper bound U on the solution length. The state sets that are used are represented in form of files. 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.

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. For memory-limited frontier search, a divide-and-conquer solution reconstruction is needed for which certain relay layers have to be stored in main memory. In external search divide-and-conquer solution reconstruction and relay layers are not needed, since the exploration fully resides on disk.

There is one subtle problem: predecessor pointers are not available on disk. This is resolved as follows. We propose to save predecessor together with every state. Once a goal is found, backtracking to the initial state along the stored files, and by looking for the matching predecessors constructs the whole solution. This results in a I/O complexity that is at most linear to the number of stored states. In the pseudo-codes this procedure is denoted by *ConstructSolution*.

```
Procedure External Enforced Hill-Climbing
    $u \leftarrow \mathcal{I}$
    $h = Heuristic(\mathcal{I})$
    while ($h \neq 0$)
        $(u', h') \leftarrow External\text{-}EHC\text{-}BFS(u, h)$
        if ($h' = \infty$) return $\emptyset$
        $u \leftarrow u'$
        $h \leftarrow h'$
    return $ConstructSolution(u)$
```

Figure 6: External Enforced Hill-Climbing.

```
Procedure External-EHC-BFS$(u, h)$
  $Open(-1, h) \leftarrow \emptyset$, $Open(0, h) \leftarrow u$
  $i \leftarrow 1$
  while ($Open(i-1, h) \neq \emptyset$)
    $A(i) \leftarrow N(Open(i-1, h))$
    forall $v \in A(i)$
      $h' = Heuristic(v)$
      if $h' < h$
        return $(v, h')$
    $A'(i) \leftarrow$ remove duplicates from $A(i)$
    for $loc \leftarrow 1$ to $locality$
      $A'(i) \leftarrow A'(i) \setminus Open(i - loc)$
    $Open(i) \leftarrow A'(i)$
    $i \leftarrow i + 1$
```

Figure 7: External-BFS for External Enforced Hill Climbing. u is the new start state with the heuristic estimate h.

External Enforced Hill Climbing

Enforced Hill Climbing (EHC) is an *enforced* form of hill climbing search. Starting from a start state, a breadth-first search is performed for a successor with a better heuristic value. As soon as such a successor is found, the hash tables are cleared and a fresh breadth-first search is started. The process continues until the goal is reached. Since EHC performs a complete breadth-first search on every state with a strictly better heuristic value, it is guaranteed to find a solution. The following theorem is due to (Hoffmann & Nebel 2001).

Theorem 4 *For directed graphs without dead-ends, Enforced Hill Climbing is complete and guaranteed to find a solution.*

Having external BFS in hand for planning domains, an external algorithm for enforced hill limbing can be constructed by utilizing the heuristic estimates. In Figure 6, we show the algorithm in pseudo-code format for external enforced hill-climbing. The externalization is embedded in the sub-procedure (Figure 7) that performs external breadth-first search for a state with better heuristic estimate. As heuristic guidance, we chose relax plan heuristics (Hoffmann 2003).

Conclusions

In this paper, we discussed a method 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.

We discuss two external algorithms in this paper: Cost-optimal external breadth-first search and external enforced hill climbing search for non-optimal planning. 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.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116:123–191.

Buchi, J. R. 1962. On a decision method in restricted second order arithmetic. In *Conference on Logic, Methodology, and Philosophy of Science*, 1–11.

DeGiacomo, G., and Vardi, M. Y. 1999. Automata-theoretic approach to planning for temporally extended goals. In *ECP*, 226–238.

Edelkamp, S., and Jabbar, S. 2006. Cost-optimal external planning. In *National Conference on Artificial Intelligence (AAAI)*. To Appear.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Gazen, B. C., and Knoblock, C. 1997. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In *ECP*, 221–233.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia.

Haslum, P., and Jonsson, P. 2000. Planning with reduced operator sets. In *AIPS*, 150–158.

Hoffmann, J., and Edelkamp, S. 2005. The deterministic part of IPC-4: An overview. *JAIR* 24:519–579.

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

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

Hopcroft, J. E., and Ullman, J. D. 2000. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Jabbar, S., and Edelkamp, S. 2005. I/O efficient directed model checking. In *VMCAI*. 313–329.

Kabanza, F., and Thiebaux, S. 2005. Search control in planing for termporally extended goals. In *ICAPS*, 130–139.

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

Lago, U. D.; Pistore, M.; and Traverso, P. 2002. Planning with a language for extended goals. In *AAAI*, 447–454.

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

Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *JAIR* 12:271–315.

Pistore, M., and Traverso, P. 2001. Planning as model checking for extended goals in non-deterministic domains. In *IJCAI*, 479–486.

Safra, S. 1998. On the complexity of omega-automata. In *Annual Symposium on Foundations of Computer Science*, 319–237. IEEE Computer Society.

Sistla, A. P.; Vardi, M. Y.; and Wolper, P. 1983. The complementation problem for Buchi automata with applications to temporal logic. *Theoretical Computer Science* 49(2–3):217–237.

Thiebaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of PDDL axioms. *Artificial Intelligence* 168(1–2):38–69.

Wolper, P. 1983. Temporal logic can be more expressive. *Information and Control* 56:72–99.

Zhou, R., and Hansen, E. 2004a. Breadth-first heuristic search. In *ICAPS*, 92–100.

Zhou, R., and Hansen, E. 2004b. Structured duplicate detection in external-memory graph search. In *AAAI*. 683–689.