Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)

Optimal Metric Planning with State Sets in Automata Representation

Björn Ulrich Borowsky and **Stefan Edelkamp***
Fakultät für Informatik,
Technische Universität Dortmund, Germany

Abstract

This paper proposes an optimal approach to infinite-state action planning exploiting automata theory. State sets and actions are characterized by Presburger formulas and represented using minimized finite state machines. The exploration that contributes to the planning via model checking paradigm applies symbolic images in order to compute the deterministic finite automaton for the sets of successors. A large fraction of metric planning problems can be translated into Presburger arithmetic, while derived predicates are simply compiled away. We further propose three algorithms for computing optimal plans; one for uniform action costs, one for the additive cost model, and one for linear plan metrics. Furthermore, an extension for infinite state sets is discussed.

Introduction

Our approach addresses planning problems which are expressible in PDDL 2.2 (Hoffmann and Edelkamp 2005). Due to the limitations of Presburger arithmetics we have to impose that the ranges of all functions are subsets of $\mathbb{Z}$, that all numerical expressions are linear, and that in the preconditions and effects no divisions nor non-integer numbers are used. We also exclude temporal planning, conditional effects and timed initial literals. The restriction to linear expressions is encountered in other metric planners (Hoffmann 2002), too. We present algorithms for three cost models:

- Uniform cost model: all actions have cost 1.
- Additive cost model: all actions have positive cost, which may depend on the state an action is applied to.
- Metric cost model: actions have arbitrary cost which may depend on the state an action is applied to.

In PDDL, additive costs can be modeled using temporal actions that define their durations through single equations. Another way is to use a monotonous problem metrics *total-cost*. When using general metrics, the cost of an action applied to a state s_1 is defined as the value the metric takes in the successor state s_2 minus the value of the metric in s_1.

Since the problem of solving arbitrary problems of the type described above is undecidable (Helmert 2002), our algorithms are not guaranteed to terminate. But if there is an optimal solution, it will be found in a finite amount of time. Alternatives either compute non-optimal plans or convert the problem into finite domain (Hoffmann et al. 2007).

The Presburger arithmetic is a first-order theory over integers. Terms in Presburger arithmetic are sums also of variables and integers. Atomic assertions are equations or inequations over terms. For the combination of atomic assertions common logical operators apply. Numbers can be existentially and universally quantified. The Presburger arithmetic is consistent and decidable, which remains true for the extension to integers. We also allow subtraction and unary $-$. Identical variables in a term can be merged to constant coefficients. Examples for formulas in Presburger arithmetics are

- $73x - 52y + 30z \leq 778 - u$
- $\forall x : ((\exists k : 2k = x) \vee (\exists k : 2k + 1 = x))$
- $(\exists y : x = 3y) \wedge (\exists y : x = 7y)$

A solution to a formula Φ is an assignment of all free variables such that Φ is satisfied. The satisfaction of a formula can be decided in double exponential time complexity with respect to the length of the formula (Fischer and Rabin 1974).

For each formula in Presburger arithmetic a deterministic finite automaton (DFA) can be constructed that accepts exactly the solution set of the formula. This work refers to the algorithm presented in (Wolper and Boigelot 2000).

The paper is structured as follows. First, we illustrate the transformation of formulas into automata. Then we show how to compile planning problems into Presburger arithmetics. We present planning algorithms based on basic automata operations to compute the image and pre-image of a state set with respect to the automaton representation of each action. Starting with BFS to minimize the number of steps, we provide a variant for metric plan cost optimization and a specialized version for additive action cost. Finally, we consider the problem of non-singleton or even infinite initial state sets and introduce the concept of multi plans.

Automata Construction

Consider the formula $\Phi := x = 1 \wedge x' = x - y$. First, automata $A_{x=1}$ and $A_{x'=x-y}$ are created for the atoms. $A_{x=1}$ uses the alphabet $\{0, 1\}$. A sequence read by $A_{x=1}$ is interpreted as the two's complement encoding (most significant

*Thanks to DFG for support in ED 74/3 and 74/2.

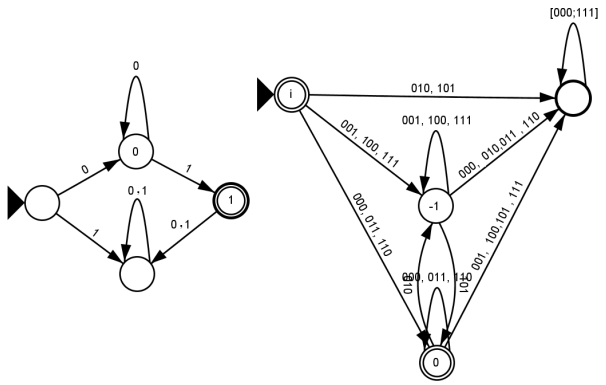

Figure 1: Automata for $x = 1$ (left) and $x' = x - y$ (right).

bit first) of the value x is currently assigned to. Automaton states correspond to values of x, and, while reading a word, $A_{x=1}$ keeps track of the current value by changing its state. The complete automaton is shown in Fig. 1 (state labels are showing corresponding values). If $A_{x=1}$ starts reading bit 1, it moves from the initial state to the overall rejecting state, because all words starting with 1 encode negative values. By reading 0001, the automaton reaches the only accepting state 1, which is correct as this is decimal value 1. Analogously, $A_{x'=x-y}$ uses alphabet $\{0,1\}^3$, where each bitvector position refers to one variable, and each state (except the rejecting one) corresponds to a certain value of $x' - x + y$. $A_{x'=x-y}$ is shown in Fig. 1 (100 means $(1,0,0)$, using variable ordering (y, x, x')). Besides state 0, the initial state is accepting, since the empty bit sequence is interpreted as 0.

Generally, an equation is first rewritten to $a_1x_1 + \cdots + a_nx_n = c$. Then state c is generated. Due to the linearity of the term and the properties of the two's complement, reading a bitvector $(b_1, \ldots, b_n)$ in a state p representing a value v means that the successor state q must represent value $2v + a_1b_1 + \ldots + a_nb_n$. With this relation it is also easy to find the predecessors of a state. Starting in state c, the automaton is built up using a backward construction in order to prevent the construction of states not reachable from the initial state. The resulting automaton is already deterministic. If c is reachable from the initial state, it is also minimal. (For inequations, the construction is similar. The resulting automaton may be nondeterministic, but in this special case, determinization can be done in linear time.) In our example, A_Φ is computed by extending the alphabet of $A_{x=1}$ to $\{0,1\}^3$ (e.g., transition label 1 maps to transition label $\{010, 011, 110, 111\}$), intersecting the extended automaton with $A_{x'=x-y}$ and minimizing the result.

Existential quantification maps to an automata operation called projection. A projection removes the i-th bit from each bitvector, where i is the position referring to the quantified variable. There are possibilities of implementing all these operations efficiently (Boigelot and Wolper 2002).

The worst-case number of states of a minimal DFA recognizing the solution set of a formula Φ is triple exponential in the length of Φ (Klaedtke 2004).

```
(define (domain Boxes)
(:types Block Box - Object)
(:predicates (in ?b - Block ?x - Box))
(:functions (weight ?o - Object) (num ?x - Box))

(:action move
 :parameters (?bl - Block ?from ?to - Box)
 :precondition (in ?bl ?from)
 :effect (and (in ?bl ?to) (not (in ?bl ?from))
  (decrease (num ?from) 1) (increase (num ?to) 1)
   (decrease (weight ?from) (weight ?bl))
   (increase (weight ?to) (weight ?bl)))))
```

Figure 2: Domain *Boxes*.

Representation of Planning Problems

The problem and its domain have to be instantiated. A planning problem in the uniform cost model induces a planning task $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{G})$. The set $\mathcal{S}$ is the state space, $\mathcal{A}$ is the set of actions. The initial state is a set $\mathcal{I} \subseteq \mathcal{S}$ (for ordinary PDDL we have $|I| = 1$), the goal state is a set $\mathcal{G} \subseteq \mathcal{S}$.

If the planning problem uses the additive cost model then it induces a planning task $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{I}, \mathcal{G})$. $\mathcal{S}$, $\mathcal{A}$, $\mathcal{I}$ and $\mathcal{G}$ are defined as before. The set $\mathcal{C} = \{c_A \mid A \in \mathcal{A}\}$ contains for each action $A \in \mathcal{A}$ a linear function $c_A : \mathcal{S} \to \mathbb{Z}^+$ that assigns a positive cost to every reachable state $s \in \mathcal{S}$.

If the metric cost model is used, it is simpler to model the metrics directly rather than modeling action specific cost functions. Hence, the planning task has the form $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{G}, m)$. Function $m : \mathcal{S} \cup \{\bot\} \to \mathbb{Z}$ maps each $s \in \mathcal{S}$ to the value the metric evaluates to in s. We obtain m by rescaling the metric to be integer-valued and a minimization problem. Constant terms are eliminated. The value $m(\bot)$ is the coefficient for *total-cost* in the modified metric. We assume $m(\bot) > 0$.

For each function symbol f of the domain we introduce a state variable x_f and a successor state variable x'_f. Now we are able to express binary comparisons and numeric effects as atoms. If we instantiate a problem referring to domain *Boxes* (Fig. 2) and defining two boxes *Box1*, *Box2* and block *B*, we also obtain the parameterless action *move_B_Box1_Box2*, which can be encoded as a formula Φ:

$$\begin{aligned}\Phi = {} & p_{in\_B\_Box1} = -1 \wedge p'_{in\_B\_Box2} = -1 \wedge p'_{in\_B\_Box1} = 0 \\ & \wedge x'_{num\_Box1} = x_{num\_Box1} - 1 \\ & \wedge x'_{num\_Box2} = x_{num\_Box2} + 1 \\ & \wedge x'_{weight\_Box1} = x_{weight\_Box1} - x_{weight\_B} \\ & \wedge x'_{weight\_Box2} = x_{weight\_Box2} + x_{weight\_B}\end{aligned}$$

Here, in analogy to the functions each predicate is encoded by a pair of variables, where value -1 means *true* and value 0 means *false*. Formula Φ describes a set of state transitions (s_1, s_2) applicable in s_1, where the components of the successor state s_2 are described by the primed variables. Multiplications with a scalar can be expressed as a formula, too. For division it is possible to simulate the behavior of integer divisions as known in C and Java (Borowsky 2007). More complex preconditions and goals are expressed using non-atomic Presburger formulas.

Sometimes a more efficient encoding can be achieved by representing several predicates by one pair of variables. The most important alternative encoding (Helmert 2004) is to partition the set of predicates, such that in each block of the partition, at most one predicate is true in any reachable state (in our example, $\{(in\_A\_Box1), (in\_A\_Box2)\}$ would be one block). Then each predicate is assigned to an index which is unique within the predicate's block. Each block is represented by one pair of variables, where a variable is assigned to the index of the predicate valid in the state to be described (there is another value indicating that no predicate is valid)[1]. If a literal is negated in a goal or a precondition, the formula may characterize not only desired, but also invalid states. For example, $\neg(p_{in\_A\_Box1} = -1)$ includes $p_{in\_A\_Box1} = k$ for all $k \in \mathbb{Z}\backslash\{-1\}$ even if we use only values 0 and -1. Invalid states may occur in the goal state set and as the predecessor state s_1 in state transitions (s_1, s_2). But during exploration, no invalid state will ever be reached as long as the representation of the initial state set does not contain invalid states.

Before creating formulas for the instantiated problem and domain, we compile away all derived predicates. Our approach of doing this assumes that in the instantiated domain, dependencies between derived predicate literals are acyclic. That is, the body of each derived predicate definition does neither directly nor indirectly depend on the positive literal forming the head of the definition. In this case the head is nothing more than a macro for its body. By substituting heads with their bodies in topological order and then replacing all heads within preconditions and the goal, all derived predicates are removed.

Solution sets map to sets of planning states or transitions. The last step is to construct DFAs representing the solution sets of the formulas. Let n be the number of state variables. The state sets $\mathcal{I}$ and $\mathcal{G}$ are represented by automata with the alphabet $\Sigma_{state} := \{0,1\}^n$. As actions induce transition sets, automata representing actions use the alphabet $\Sigma_{trans} := \{0,1\}^{2n}$. We use some fixed variable ordering for Σ_{state}. For $i = 1, 3, \ldots, 2n-1$, the bitvector position i of Σ_{trans} is assigned to the same variable as for the position $(i+1)/2$ of Σ_{state}, and the bitvector position $i+1$ of Σ_{trans} is assigned to the successor state variable corresponding to the state variable at position i of Σ_{trans}.

Planning Algorithms

Next we propose different algorithms for exploring the state spaces and for the construction of plans. The algorithms are inspired by methods for finite state spaces that use BDDs (Cimatti et al. 1997), but considerable extensions are needed. For improved readability, the descriptions of the algorithms do not refer to the automata but to state and transition sets. Intersection, union, complement, subset containment and emptiness checking of sets easily map to corresponding operations on DFAs. In addition to the standard operations, we need two further operations: the computation of the predecessors set and the successors set of a state set with respect to a transition set. For a state set $S \subseteq \mathcal{S}$ and a transition set $T \subseteq \mathcal{S} \times \mathcal{S}$, the predecessors set *pred*(S,T) and the successors set *succ*(S,T) are defined as follows:

$$\textit{pred}(S,T) := \{s_1 \in \mathcal{S} \mid \exists s_2 \in S : (s_1, s_2) \in T\}$$
$$\textit{succ}(S,T) := \{s_2 \in \mathcal{S} \mid \exists s_1 \in S : (s_1, s_2) \in T\}$$

Given automata A_S and A_T representing S and T, *succ*(S,T) can be computed by first extending A_S to alphabet Σ_{trans} (mapping the positions of A_S to positions $i = 1, 3, \ldots, 2n-1$), then intersecting extended A_S with A_T, projecting out all bitvector positions of the obtained automaton that are assigned to state variables and minimizing the automaton afterwards. The automaton for *pred*(S,T) is computed similarly.

Algorithm 1 BFS

Input: Planning task $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{G})$, set I with $\emptyset \neq I \subseteq \mathcal{I}$
Output: A shortest sequential plan and a set $I' \subseteq I$ or 'No plan.'

```
L := R := stack := I;
while L ∩ G = ∅ do
    L' := ∅
    for all A ∈ A do
        L' := L' ∪ succ(L, A)
    if L' ⊆ R then return 'No plan.'
    R := R ∪ L'; stack.push(L'); L := L'
G := stack.pop() ∩ G
return Extract-Plan(R, stack, G)
```

Breadth-First Search

Let $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ be a planning task. We search for a shortest sequential plan which transforms a non-empty subset of $\mathcal{I}$ in a subset of $\mathcal{G}$. The problem graph induced by $\mathcal{R}$ has node set $\mathcal{S}$ and edge set $\mathcal{A}$. The problem graph is infinite, directed and unweighted. A shortest sequential plan corresponds to a shortest path in the problem graph and can be computed via set-based BFS (see Algorithm 1). Under the i-th layer ($i = 0, 1, \ldots$) we understand the set of all states $s \in \mathcal{S}$ reachable from some state $s_0 \in I$ by executing some plan that has *exactly* i steps. Note that for $i \neq j$ the i-th layer and the j-th layer are not necessarily disjoint. For $i = 0, 1, \ldots$, the algorithm computes the i-th layer, one after another. Variable L contains the last completed layer. The already computed subset of the next layer is stored in variable L' simulating the queue for explicit-state breadth-first search. The set R contains all states that are contained in at least one set being assigned to L. The algorithm initializes L with I, such that I is the 0-th layer. The layers are maintained in *stack*, since they are needed in reverse order for solution reconstruction. The next layer is the successors set of L with respect to all applicable actions. The exploration terminates if the current layer contains a goal state.

If Algorithm 1 has reached a goal state, it extracts an optimal sequential plan from the stack by calling Algorithm 2. Additionally, with $G \subseteq \mathcal{G}$ the set of all reached goal states is provided. In this case this is the intersection of the last layer with $\mathcal{G}$. First the algorithm searches for an action A with the property that the set of predecessors of G with respect to A has a non-empty intersection with the previous layer.

[1] See (Borowsky 2007) for a formal definition and examination of predicate encodings.

Algorithm 2 Extract-Plan

Input: Planning task $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{G})$, stack *stack*, $G \subseteq \mathcal{G}$
Output: Sequential plan of length $|stack|$, $I' \subseteq \mathcal{I}$

```
π := []; post := G
while stack ≠ ∅ do
  pre := stack.pop(); post' := ∅
  for all A ∈ 𝒜 do
    post' := pred(post, A) ∩ pre
    if post' ≠ ∅ then; a := A; break
  post := post'; π := [a].π
return (π, post)
```

This action is the last action in the plan. Then the second last action of the plan is computed analogously, where only the predecessors in the currently active layer are considered. The algorithm continues unless the bottom-most layer on the stack has been encountered.

If the set of all reachable states $\mathcal{S}_I := \{s \in \mathcal{S} \mid \exists (s_0, s_1), (s_1, s_2), \ldots, (s_{l-1}, s_l) \in \mathcal{A} : (s_0 \in I \wedge s_l = s)\}$ is finite, or if $\mathcal{S}_I \cap \mathcal{G} \neq \emptyset$, the algorithm terminates.

BFS guarantees that all goal states with the shortest distance l are found before all other goal states.

Algorithm 3 Metric-BFS

Input: Planning task $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{G}, m)$, set I with $\emptyset \neq I \subseteq \mathcal{I}$
Output: An optimal sequential plan and a set $I' \subseteq I$ or 'No plan.'

```
L := R := stack := I; stack' := ∅; b := ∞; G := ∅
while true do
  B := {s ∈ 𝒮 | m(s) < b − m(⊥) · (|stack| − |stack'| − 1)};
  X := L ∩ 𝒢 ∩ B
  if X ≠ ∅ then
    b := MinVal(m, X); stack' := stack; stack'.pop();
    G := X ∩ {s ∈ 𝒮 | m(s) = b}
  L' := ∅
  for all A ∈ 𝒜 do
    L' := L' ∪ succ(L, A)
  if L' ⊆ R then
    if b = ∞ then return 'No plan.';
    else return Extract-Plan(ℛ, stack', G)
  else R := R ∪ L'; stack.push(L'); L := L'
```

Algorithm 3 is a metric variant of BFS. Its loop does not terminate when reaching a goal state, because subsequent layers may contain other goal states which let the metric take a smaller value. Variable b stores the value $m(s)$ of a state $s \in \mathcal{G}$ belonging to the best states reached so far. Every new layer is examined for new goal states which are better than all the goal states reached in older layers. When comparing the values with b, it is important to consider that the original metric may depend on *total-cost*. If this is the case, the monom containing *total-cost* will yield different values for new goal states and goal states discovered in an older layer. If the current layer contains better goal states, then the algorithm calls *MinVal* (see Algorithm 4) which uses a geometric search to determine the minimal value m takes for at least one of these goal states. In contrast to many known approaches, *Metric-BFS* does not require to extend the state space by another dimension representing *total-cost*.

Proposition 1 *For $m : \mathcal{S} \cup \{\perp\} \to \mathbb{Z}$ and S with $\emptyset \neq S \subseteq \mathcal{S}$,* MinVal *computes the smallest value $m(s)$ takes for some $s \in S$, if a smallest value exists.*[2]

Algorithm 4 MinVal

Input: Metric $m : \mathcal{S} \cup \{\perp\} \to \mathbb{Z}$, state set S with $\emptyset \neq S \subseteq \mathcal{S}$
Output: Minimal value $m(s)$ takes for $s \in S$

```
U' := U := m(s) for an arbitrary element s ∈ S;
  D := 128; L := U − D
while S ∩ {s ∈ 𝒮 | m(s) ≤ L} ≠ ∅ do
  U' := L; D := 2 · D; L := U − D
U := U'
while L ≤ U do
  V := ⌊(L + U)/2⌋
  if S ∩ {s ∈ 𝒮 | m(s) ≤ V} ≠ ∅ then U := V − 1
  else L := V + 1
return V
```

In the following, we require that there is always a smallest value w within S. In particular, this is true if $\mathcal{I}$ is finite. If $\mathcal{S}_I$ is finite then *Metric-BFS* will terminate, since case $R = \mathcal{S}_I$ will occur after a finite amount of iterations and in the next iteration condition $L' \subseteq R$ will be fulfilled. If I is finite and $\mathcal{S}_I$ is infinite, however, *Metric-BFS* will not terminate.

Proposition 2 *If there is a plan, then – after a finite amount of iterations –* stack′ *will be assigned to a content from which* Extract-Plan *computes an optimal plan, which uses the smallest number of actions among all optimal ones. If* Metric-BFS *terminates, such a plan will be returned.*

One can also think of manually aborting the planning process and return the best solution found so far.

Dijkstra Algorithm

Now we consider planning tasks of the form $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{I}, \mathcal{G})$. They are specializations of the metric problems examined in the previous section because action cost are restricted to positive values. The problem graph induced by $\mathcal{R}$ can basically be modeled as the one for non-metric problems. But here, edges are assigned to cost. As a transition (s_1, s_2) can be executed by more than one action, we have to model transitions as triplets (s_1, c, s_2), where $c \in \{1, 2, \ldots\}$ is the cost of the transition when applying the action belonging to the transition to state s_1. An optimal plan corresponds to a path from an initial state to a goal state such that the sum of the weights of all the edges forming the path is minimal. The planning problem reduces to a weighted shortest paths problem, which can be solved by an algorithm based on single-source-shortest-paths finding.

The shortest-paths algorithm of Dijkstra (1959) is adapted in Algorithm 5. The priority queue is represented by data structures D and *open*. Set D contains all key-value pairs already inserted into the queue, whereas *open* contains only those pairs which have not yet been retrieved through a *deleteMin* operation. The union of all state sets retrieved through *deleteMin* is stored in R. (The first two rows of the loop correspond to a *deleteMin* operation.) If the extracted state set M contains a goal state, the algorithm proceeds with solution reconstruction. Otherwise R is updated and

[2] Proofs are found in (Borowsky 2007)

successor sets are computed with respect to single actions. In case of non-constant action cost, first M is partitioned into blocks consisting of states with an identical cost. For each block the set consisting of all successors not retrieved from queue earlier is inserted into the priority queue using the appropriate key, if the set is not empty. State sets inserted under different keys are not necessarily disjoint as we do not detect duplicates. That the problem graph is actually a multi graph does not affect the correctness of the algorithm.

Algorithm 5 Dijkstra-Search

Input: Planning task $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{I}, \mathcal{G})$, set $I \subseteq \mathcal{I}$
Output: Optimal sequential plan and set $I' \subseteq I$ or 'No plan.'

$D := \{(0, I)\}$; *open* $:= \{(0, I)\}$; $R := \emptyset$
while *open* $\neq \emptyset$ **do**
  $(t, M) :=$ pair from *open* with smallest key;
  open $:=$ *open*$\backslash\{(t, M)\}$
  if $M \cap \mathcal{G} \neq \emptyset$ **then** $G := M \cap \mathcal{G}$; $k := t$;
    return *Extract-Additive-Cost-Plan*$(\mathcal{R}, D, k, G, R)$
  $R := R \cup M$
  for all $A \in \mathcal{A}$ **do**
    if c_A constant **then**
      $t' := t + c_A(s), s \in \mathcal{S}$;
      $M' :=$ *succ*$(M, A)\backslash R$
      if $M' \neq \emptyset$ **then**
        if $\exists M'' : (t', M'') \in D$ **then**
          $M' := M' \cup M''$;
          $D := D\backslash\{(t', M'')\}$; *open* $:=$ *open*$\backslash\{(t', M'')\}$
        $D := D \cup \{(t', M')\}$; *open* $:=$ *open* $\cup \{(t', M')\}$
    else
      $\mathcal{P} :=$ *Partition*(c_A, M)
      for all $(d, P) \in \mathcal{P}$ **do**
        $t' := t + d$;
        $M' :=$ *succ*$(P, A)\backslash R$
        if $M' \neq \emptyset$ **then**
          if $\exists M'' : (t', M'') \in D$ **then**
            $M' := M' \cup M''$;
            $D := D\backslash\{(t', M'')\}$; *open* $:=$ *open*$\backslash\{(t', M'')\}$
          $D := D \cup \{(t', M')\}$; *open* $:=$ *open* $\cup \{(t', M')\}$
return 'No plan.'

The extraction of a sequential plan makes use of the structures $\mathcal{R}$, D and R, and needs also set G of all goal states retrieved from the queue in the last iteration of the while loop and the corresponding key k (the last value of t). Here, too, the plan is extracted in inverse order of execution. Obviously, the result must be a plan with cost k. Variable t in Algorithm 7 stores the current cost of plan execution, therefore the initial value for t is k. Set M is a superset of the set of all intermediate states generated by the plan at time point t (similar to *post* in Algorithm 2). We have completed plan extraction as soon as we reach cost 0.

For *Dijkstra-Search* it is not difficult to see that if $\mathcal{S}_I$ is finite or if $\mathcal{S}_I \cap \mathcal{G} \neq \emptyset$ the while-loop terminates, and if I is finite and $\mathcal{S}_I$ is infinite, and if $\mathcal{S}_I \cap \mathcal{G} = \emptyset$, then the while loop does not terminate.

Proposition 3 *The call* Extract-Additive-Cost-Plan$(\mathcal{R}, D, k, G, R)$ *invoked in* Dijkstra-Search *computes a sequential plan of cost k transforming a nonempty subset I' of I into a nonempty subset of G.*

Algorithm 6 Partition

Input: Function $d : \mathcal{S} \to \mathbb{Z}^+$, state set $S \subseteq \mathcal{S}$
Output: Set $\{(c_1, P_1), \ldots, (c_n, P_n)\}$ with $c_1, \ldots, c_n \in \mathbb{Z}^+, c_i \neq c_j, i \neq j$, partition $\{P_1, \ldots, P_n\}$ of $S, \forall s \in P_i : d(s) = c_i$

$U := d(s)$ for an arbitrary element $s \in S$; $D := 128$
while $S \cap \{s \in \mathcal{S} \mid d(s) \leq U - D\} \neq \emptyset$ **do** $D := 2 \cdot D$
$L := U - D$; $D := 128$
while $S \cap \{s \in \mathcal{S} \mid d(s) \geq U + D\} \neq \emptyset$ **do** $D := 2 \cdot D$
$U := U + D$; $\mathcal{P} := \emptyset$; $B :=$ *true*
while B **do**
  $B :=$ *false*; $U' := U$; $L' := L$
  while $L' \leq U'$ **do**
    $V := \lfloor (L' + U')/2 \rfloor$
    if $S \cap \{s \in \mathcal{S} \mid L' \leq d(s) \leq V\} \neq \emptyset$ **then**
      $U' := V - 1$; $B :=$ *true*; **else** $L' := V + 1$
  if B **then** $L := V + 1$; $P := S \cap \{s \in \mathcal{S} \mid d(s) = V\}$;
    $\mathcal{P} := \mathcal{P} \cup \{(V, P)\}$
return $\mathcal{P}$

Algorithm 7 Extract-Additive-Cost-Plan

Input: Planning task $\mathcal{R} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{I}, \mathcal{G})$, set D of key-value pairs, key $k \in \mathbb{N}_0$, $G \subseteq \mathcal{G}$, set R of all states
Output: Sequential plan with cost k, $I' \subseteq \mathcal{I}$

$t := k$; $M := G$; $\pi := []$
while $t > 0$ **do**
  for all $A \in \mathcal{A}$ **do**
    $M' :=$ *pred*(M, A)
    if c_A constant **then**
      $t' := t - c_A(s)$ for an arbitrary $s \in \mathcal{S}$
      if $\exists M'' : (t', M'') \in D$ **and** $M' \cap M'' \neq \emptyset$ **then**
        $\pi := [A].\pi$; $M := M' \cap M''$; $t := t'$; **break**
    else
      $M' := M' \cap R$; $\mathcal{P} :=$ *Partition*(c_A, M')
      for all $(d, P) \in \mathcal{P}$ **do**
        $t' := t - d$
        if $\exists M'' : (t', M'') \in D$ **and** $P \cap M'' \neq \emptyset$ **then**
          $\pi := [A].\pi$; $M := P \cap M''$; $t := t'$; **break**
return π

Proposition 4 *If* Dijkstra-Search *terminates the algorithm returns a plan that transforms $I' \subseteq I$ into G with $\emptyset \neq G \subseteq \mathcal{G}$ and minimal cost.*

Sets of Initial States

One property of finite automata is the ability of recognizing infinite languages. Any automaton representing a nonempty solution set of a Presburger formula already recognizes an infinite language, as in the two's complement system the sign bit can be duplicated an arbitrary number of times without changing the numeric value encoded by the bit string. Due to the restriction $|\mathcal{I}| = 1$ all represented state sets $S \subseteq \mathcal{S}$ are finite for a classical PDDL input. To exploit the potential of finite automata, we allow the specification of a set of initial states instead of a single initial state. This is accomplished by using a goal description. We wish to compute a plan, which transforms each initial state into a goal state, being optimal for each initial state. Of course, such a plan rarely exists. For this reason the planning system should compute a finite set of plans, where each plan is an optimal solution for a subset of $\mathcal{I}$. In the ideal case, each

Algorithm 8 Multi-Search

Input: Planning task $\mathcal{R} = (\mathcal{S}, \mathcal{A}, [\mathcal{C},]\mathcal{I}, \mathcal{G}[, m])$
Output: An optimal multi plan for $\mathcal{R}$

```
P := ∅; I := 𝓘
while I ≠ ∅ do
  if [𝓒] then r := Dijkstra-Search(R, I)
  else
    if [m] then r := Metric-BFS(R, I)
    else r := BFS(R, I)
  if r = (π, I') then P := P ∪ {(I', π)}; I := I\I'
  else return P
return P
```

initial state is covered by one of these subsets.

A *multi plan* for $\mathcal{R}$ is a set $P = \{(C_1, P_1), \ldots, (C_n, P_n)\}$ with $\emptyset \neq C_i \subseteq \mathcal{I}$ and the property that for all $s_0 \in C_i$ P_i is a (possibly suboptimal) plan for $\mathcal{R}' = (\mathcal{S}, \mathcal{A}, [\mathcal{C},]\{s_0\}, \mathcal{G}[, m])$. Sets $C_1, \ldots, C_n$ are pairwise disjoint. If $\mathcal{R}' = (\mathcal{S}, \mathcal{A}, [\mathcal{C},]\{s_0\}, \mathcal{G}[, m])$ is unsolvable for all $s_0 \in \mathcal{I} \backslash \bigcup_{1 \leq i \leq n} C_i$ then we say P is *complete*. If P is complete and for all $s_0 \in C_i$ plan P_i is optimal for $\mathcal{R}' = (\mathcal{S}, \mathcal{A}, [\mathcal{C},]\{s_0\}, \mathcal{G}[, m])$ $(i = 1, \ldots, n)$, P is optimal.

In order to obtain an automaton for $\mathcal{I}$ the automaton constructed for the goal description defined in the *:init* block is to be intersected with an automaton representing the state space $\mathcal{S}$. The reason for the additional operation is that the problem of characterizing invalid states through goal descriptions cannot be ignored here because invalid states would be reachable.

All algorithms presented in section already allow arbitrary initial state sets as input and also return a subset $M_1 \neq \emptyset$ of the input state set. If another exploration is started with $\mathcal{I} \backslash M_1$ as the initial state set, a set $M_2 \subseteq \mathcal{I}$ mit $M_2 \cap M_1 = \emptyset$ is returned. By repeating exploration with more and more restricted initial state sets (see Algorithm 8), a multi plan can be computed.

Proposition 5 *The returned multi plan is optimal, if any.*

The result of *Multi-Search* is influenced by the output of the underlying exploration algorithm, which are themselves the output of the plan extraction algorithms. The data structure built up during exploration implicitly contains all optimal plans found. For both extraction algorithms, which plan is extracted depends on the order in which actions are tested. It is possible that there is a plan P_{opt} which transforms each state $s \in I$ into a goal state in an optimal way, whereas for each $s \in I$ there is an optimal plan P_s which only transforms s into one goal state. In case of a bad test order, $|I|$ plans P_s are inserted into the multi plan instead of P_{opt}. In case of $|I| = \infty$ the while loop of *Multi-Search* does not terminate. In order to avoid this behavior and similar phenomena, a random test order should be chosen before the next action is extracted. In our worst-case scenario, the explorations may extract plans P_s first, but P_{opt} is still applicable to the remaining states. As the number of plans already found is always finite (in each exploration the optimal plans found have the same cost) and P_{opt} is chosen with a probability greater than zero, P_{opt} is extracted after and expected time which is finite.

The "meta exploration" offers the possibility of computing deterministic plans even if there is uncertainty about the actual initial state. If an initial state is given later, we just have to determine if and which set C_i contains that state.

Conclusion and Outlook

We have seen approaches for optimally solving PDDL2.2 action planning problems. A distinguished advantage is the processing of infinite state sets. An adaption of A* search is desirable. Currently, only an inefficient implementation exists, but we are developing a more efficient one, which also will support real-valued variables and the integration of different automata libraries, such as the very efficient LIRA (Becker et al. 2007), for instance.

Acknowledgements Thanks to the anonymous reviewers for their suggestions to improve the accessibility of the text.

References

Becker, B.; Dax, C.; Eisinger, J.; and Klaedtke, F. 2007. LIRA: Handling constraints of linear arithmetics over the integers and the reals. In *CAV*.

Boigelot, B., and Wolper, P. 2002. Representing arithmetic constraints with finite automata: An overview. In *ICLP*, 1–19.

Borowsky, B. 2007. Optimal metric planning with state sets in automata representation. Master's thesis, TU Dortmund.

Cimatti, A.; Giunchiglia, E.; Giunchiglia, F.; and Traverso, P. 1997. Planning via model checking: A decision procedure for AR. In *ECP*, 130–142.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Fischer, M. J., and Rabin, M. O. 1974. Super-exponential complexity of Presburger arithmetic. MIT. http://www.lcs.mit.edu/publications/pubs/ps/MIT-LCS-TM-043.ps.

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *AIPS*, 44–53.

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

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

Hoffmann, J.; Gomes, C. P.; Selman, B.; and Kautz, H. A. 2007. SAT encodings of state-space reachability problems in numeric domains. In *IJCAI*, 1918–1923.

Hoffmann, J. 2002. Extending FF to numerical state variables. In *ECAI*.

Klaedtke, F. 2004. On the automata size for Presburger arithmetic. In *LICS*, 110–119.

Wolper, P., and Boigelot, B. 2000. On the construction of automata from linear arithmetic constraints. In *TACAS*, 1–19.