Propagation of PDDL3 Plan Constraints

P@trik Haslum
Australian National University & NICTA Optimisation Research Group
firstname.lastname@anu.edu.au

Abstract

We present a sound, though incomplete, and tractable propagation procedure for PDDL3 trajectory constraints, with the aim of providing an inexpensive unsatisfiability test for sets of such constraints. The propagator is supported by (tractable) methods that derive additional constraints from the problem description. It is applied to compute lower bounds on penalty for problems with soft trajectory constraints (preferences).

Introduction

PDDL version 3 (Gerevini et al. 2009) introduced *trajectory constraints*, a subset of linear temporal logic that can express constraints on the sequence of states visited by the execution of a plan, in addition to the constraint on the end state of the sequence that is imposed by the planning goal. Given a planning problem P and a set of trajectory constraints C, the question we must answer is, is there a plan for P whose induced state sequence satisfies C? This question is clearly as hard as deciding if there is any plan at all for P, i.e., PSPACE-complete.

However, suppose we know that P has a plan, and that we need to check not one set of trajectory constraints but a large number of different sets of constraints. This situation arises, for example, if the trajectory constraints are "soft", i.e., preferences rather than hard constraints, and we are searching for a most preferred subset that is satisfiable w.r.t. P. We should, at least in some cases, be able to infer that a constraint set C is unsatisfiable w.r.t. P without exhaustively searching through all plans for P.

This paper presents an approach to this problem, in the form of a sound but incomplete, and tractable, propagation procedure for PDDL3 trajectory constraints. That is, given a constraint set C, and some information extracted from the problem P, the propagator computes additional constraints that are implied by those given; if it finds an implied contradiction, we know that C is unsatisfiable w.r.t. P. Because the propagator reasons (mostly) not about the problem but only about constraints extracted from it, time complexity scales additively in the size of P and the number of constraint sets to be tested.

This work is motivated by a specific example of the kind of problem described above: The Rovers Qualititative-Preferences domain from the 2006 International Planning Competition. In this domain, the objective is to satisfy a maximum weight subset of preferences over trajectory constraints. For each problem there is a plan that achieves the hard goals, but the complete set of constraints can not be simultaneously satisfied. Identifying subsets of constraints that are contradictory (w.r.t. P) allows computing bounds on the minimum penalty, due to unsatisfied preferences, of any plan (Haslum 2007). My previous approach to testing satisfiability of plan constraint sets was to compile the constraints into the problem and test for unsolvability with the admissible h^m heuristic. The disadvantage of this test is that the complexity of computing h^m depends on the size of the problem (albeit only polynomially, for fixed m). As we will demonstrate, this causes the propagation-based test to scale up much better as the size of the problem grows: for the largest instances, it is three orders of magnitude faster at performing a single test. However, because the compilation-based test is able to exploit properties of the h^m heuristic to amortise computation over several tests, the difference in total runtime is only a factor 2.89 (median). On the other hand, the propagation- and compilation-based methods are complementary, in the sense that both find unsatisfiable sets that the other cannot detect. Thus, lower bounds based on the combined results of both methods are generally best.

PDDL3

PDDL3 (Gerevini et al. 2009) extends PDDL with two new features: Preferences are "soft goals", which may be either normal, final state goals or preferences over trajectory constraints. Trajectory constraints are expressed using a set of five modal operators, which may not be nested. The satisfaction of a constraint is determined by the sequence of states visited by a plan's execution. Each PDDL3 operator corresponds to a particular formula in linear temporal logic (Pnueli 1977), provided a suitable interpretation of LTL over finite state sequences (Bauer and Haslum 2010).

For ease of presentation, we consider here a standard propositional STRIPS model of planning problems, without negation. That is, a planning problem P consists of a set of propositional atoms (V), a set of actions (A), and an initial state s_0. A state (including s_0) is an assignment of truth values to the atoms in V, i.e., a propositional logic model. Each action a is described by its precondition ($\mathrm{pre}(a)$), add ($\mathrm{add}(a)$) and delete ($\mathrm{del}(a)$) effects, which are all sets of

Constraint name	φ	$\vec{s} = s_0, s_1, \ldots, s_n \models \varphi$ iff...
`(at-end α)`	$\mathsf{F}\alpha$	$s_n \models \alpha$
`(always α)`	$\mathsf{A}\alpha$	$\forall i\, s_i \models \alpha$
`(sometime α)`	$\mathsf{E}\alpha$	$\exists i\, s_i \models \alpha$
`(at-most-once α)`	$\mathsf{AMO}\alpha$	$\forall i$ if $s_i \models \alpha$ then $\exists j \geq i\, \forall i \leq k \leq j\, s_k \models \alpha$ and $\forall k > j\, s_k \not\models \alpha$
`(sometime-before α β)`	$\beta\,\mathsf{SB}\,\alpha$	$\forall i$ if $s_i \models \alpha$ then $\exists j < i\, s_j \models \beta$
`(sometime-after α β)`	$\beta\,\mathsf{SA}\,\alpha$	$\forall i$ if $s_i \models \alpha$ then $\exists j \geq i\, s_j \models \beta$
Never α	$\mathsf{N}\alpha$	$\forall i\, s_i \not\models \alpha$
Never β after α	$\beta\,\mathsf{NA}\,\alpha$	$\forall i$ if $s_i \models \alpha$ then $\forall j \geq i\, s_j \not\models \beta$

Table 1: PDDL3 and auxiliary plan constraints.

atoms, interpreted as conjunctions. The action is applicable in a state s iff $s \models \mathrm{pre}(a)$, and applying it leads to a state s' where all atoms in $\mathrm{add}(a)$ are true, all atoms in $\mathrm{del}(a) - \mathrm{add}(a)$ are false, and all other atoms retain their value from s. Note that this definition of a planning problem does not include a goal. PDDL3 has a special trajectory constraint for facts that must hold at the end of a plan execution. Thus, the standard notion of a planning goal is subsumed by the more general condition of satisfying a set of trajectory constraints, defined below.

Every sequence of actions, $\vec{a} = a_1, \ldots, a_n$, from A that is executable from the initial state induces a corresponding sequence of states, $\vec{s} = s_0, s_1, \ldots, s_n$, visited by the execution. We call this an *execution of* P. PDDL3 trajectory constraints are evaluated over state sequences. We write $\vec{s} \models \varphi$, where φ is a trajectory constraint, if $\vec{s}$ satisfies φ. We also write $P \models \varphi$ if every execution of P satisfies φ. Given a planning problem P and a set C of trajectory constraints, we say that C is satisfiable w.r.t. P iff there exists an execution of P that satisfies each constraint in C.

The PDDL3 trajectory constraints and their satisfaction conditions are summarised in Table 1. It also introduces an abbreviation for each constraint, and two auxiliary constraints which will be useful in describing the propagation algorithm. Note that PDDL3 does not allow nesting of modal operators: the formulas α and β are only allowed to be *state formulas*, i.e., Boolean formulas over V. To ensure that implication between state formulas can be decided in polynomial time, we assume that these formulas are single atoms or sets of atoms (i.e., conjunctions). This is, however, not an essential restriction of the propagation algorithm. If support for general formulas is desired, we may either give up tractability and use a complete SAT solver to decide implication, or use some sound but incomplete polynomial-time implication test. The only requirement on the test is that it is closed under transitivity; that is, if the test proves $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, it must also prove $\alpha \rightarrow \gamma$.

Note the asymmetry between the `sometime-before` and `sometime-after` constraints: $\alpha\,\mathsf{SB}\,\beta$ requires α to hold strictly before β is first achieved, while $\alpha\,\mathsf{SA}\,\beta$ requires α to hold at the same time as or after any time that β is true. The propagation procedure currently does not consider SA constraints. Some of the challenges and possibilities of making inferences from such constraints are discussed later.

In the following we will use one additional notation: $\mathsf{D}a$, where a is an action. It is read "disallowed a", and means that a must not appear in any action sequence. It is not a modal operator like in other trajectory constraints, but is implied by those in some situations. For example, if the state sequence must satisfy $\mathsf{N}\alpha$ and $\mathrm{pre}(a) \rightarrow \alpha$, then a can not be part of the corresponding action sequence. We use $\mathsf{D}a$ as a shorthand for stating that some condition that prevents the inclusion of a holds.

Inferring Constraints from the Problem

The propagation procedure works on a set of trajectory constraints, C. However, since the aim is to infer if the constraint set is inconsistent w.r.t. a planning problem, P, we extract certain information from P, which is provided as input to the propagator. This information is (mostly) expressed as additional trajectory constraints.

This is an important design decision. Since the motivation is to perform quick (in-)consistency tests on many different trajectory constraint sets, the complexity of the test should not depend (too much) on the size of P. It is acceptable to perform relatively expensive (though still tractable) computation on P to extract information that is used by the test, but not to repeat this computation for every constraint set that is tested.

Mutual Exclusion Mutual exclusion, or "mutex" for short, holds between two state formulas, α and β, iff there is no reachable state in which both of them are true, i.e., $P \models \mathsf{N}(\alpha \wedge \beta)$. We use the shorthand notation $\mathrm{mutex}(\alpha, \beta)$.

Deciding mutual exclusion in general is as hard as solving the planning problem, but there are numerous methods for computing a sound but incomplete set of mutex relations, using admissible heuristics or invariant-finding methods (e.g. Gerevini and Schubert 1998; Rintanen 2000; Helmert 2006). In the implementation of the propagator we use the pair-wise atom mutex set found by the h^2 heuristic.

Landmarks The concept of landmarks in planning was first introduced by Porteous, Sebastia and Hoffmann (2001), and have been used in many ways since. Informally, a landmark is "something that must happen at some point in any plan". Different varieties of landmarks have been defined, where the "something" is a fact, formula or set of actions.

We consider a landmark relation between state formulas: α *is a landmark of* β iff α must be achieved (strictly) before β in every execution of P. This is precisely the same as saying that $P \models \alpha\,\mathsf{SB}\,\beta$. Fact landmarks, in the usual

sense, are landmarks of the planning goal. Deciding if the landmark relation holds between arbitrary state formulas is again PSPACE-hard, but a sound approximation for single-atom state formulas can be computed in polynomial time by testing relaxed reachability of β in a problem modified by removing α from the add effects of all actions and the initial state. Note that if α is true in s_0 or β is unreachable in the original problem, $\alpha \,\mathsf{SB}\, \beta$ holds trivially. Such trivial relations are ignored.

The Never-After Relation $P \models \beta \,\mathsf{NA}\, \alpha$ iff β does not hold in, and cannot be achieved from, any reachable state where α holds. This is a kind of extended mutex relation: a normal mutex says that α and β cannot be true simultaneously, while $\beta \,\mathsf{NA}\, \alpha$ says that if α ever was true, β can never become true.

A sound but incomplete set of never-after relations between single atoms can be computed by relaxed reachability tests. Let p be an atom and s_p a state in which every atom except those that are mutex with p is true: if q is not relaxed reachable from s_p, then $P \models q \,\mathsf{NA}\, p$. This is similar to Vidal & Geffner's (2004) computation of inter-action distances, but distinguishing only the case of infinite distance (unreachability).

Conditional Constraints Relations between formulas, like landmarks and never-after, are a consequence of lack of choice. Generally, the more alternative ways there are of achieving β, the fewer α's will be landmarks of it. However, if actions are disallowed (because their preconditions or effects contradict some constraint), choices narrow, and new relations that previously did not hold may become valid. For example, suppose there are two alternative plans for getting from `A` to `B`: `(go A C)`, `(go C B)` and `(go A D)`, `(go D B)`. If, however, `(go A C)` is disallowed, there is only one way and `(at D)` becomes a landmark of `(at B)`. If both `(go A C)` and `(go A D)` are disallowed, `(at B)` becomes unreachable. This idea extends also to other types of trajectory constraints.

Definition 1 *$\langle \varphi, \overline{A} \rangle$, where φ is a trajectory constraint and $\overline{A}$ a set of actions, is a* conditional constraint *of P iff φ holds in every execution of P that does not include any action in $\overline{A}$.*

That is, if all actions in $\overline{A}$ become disallowed, then the constraint φ is satisfied by all remaining executions of P. Equivalently,

$$P \models \left(\bigwedge_{a \in \overline{A}} \mathsf{D}a \right) \rightarrow \varphi.$$

We also say that φ holds in P conditional on $\overline{A}$.

We consider two types of conditional constraints: landmarks, i.e., constraints $\alpha \,\mathsf{SB}\, \beta$, and unreachability, i.e., constraints of the form $\mathsf{N}\alpha$. The next proposition provides a method to compute a sound, but not necessarily complete, set of such conditional constraints, with single-atom state formulas, without enumerating subsets of actions. It may be that, for example, conditional never-after constraints also exist in a problem, but we currently do not have an effective method of finding them.

Proposition 2 *Let p be an atom that is false in the initial state, and $\mathrm{Adds}(p) = \{a \mid p \in \mathrm{add}(a)\}$ the set of actions that add p: $\langle \mathsf{N}p, \mathrm{Adds}(p) \rangle$ is a conditional (unreachability) constraint of P.*

Furthermore, for each $q \in \bigcup_{a \in \mathrm{Adds}(p)} \mathrm{pre}(a)$, such that q is not already a landmark of p, let $\mathrm{Reqs}(q) = \{a \mid q \in \mathrm{pre}(a)\}$ be the set of actions whose preconditions include q: $\langle q \,\mathsf{SB}\, p, \mathrm{Adds}(p) - \mathrm{Reqs}(q) \rangle$ is a conditional (landmark) constraint of P.

Proof: Since p is not initially true, some action in $\mathrm{Adds}(p)$ must take place to make it true; hence, if these actions are disallowed, $\mathsf{N}p$ must hold. Furthermore, if all actions in $\mathrm{Adds}(p) - \mathrm{Reqs}(q)$ are disallowed, all remaining actions that add p have q in their precondition. Thus q must be achieved before p. □

Enumerating pairs of propositions p and q gives directly a polynomial-time algorithm for computing conditional landmarks and unreachability, since the sets $\mathrm{Adds}(p)$ and $\mathrm{Reqs}(q)$ are immediate from action definitions.

In the example above, this algorithm will find that `(at D)` is a landmark of `(at B)` conditional on $\{$`(go C B)`$\}$, but it will not find that the same relation is also conditional on $\{$`(go A C)`$\}$.

The Propagation Algorithm

Algorithm 1 presents the main propagation algorithm. Its arguments are a set of trajectory constraints, C, and a set of conditional (landmark and unreachability) constraints, X. C is assumed to contain any non-conditional constraints (landmark and never-after relations) inferred from the problem.

The algorithm first infers state formulas that can never hold in any execution that satisfies C (lines 4–23), repeating a cycle of inferences until a fixpoint is reached, then infers state formulas that must hold, at some point, in any execution (lines 24–25). All inferences are restricted to state formulas that appear in the input. If there is a formula that must hold but cannot, a contradiction has been found. Finally, a separate check for inconsistencies with `at-most-once` constraints is done. This is detailed in Algorithm 2.

The algorithm maintains two data structures: a set D of disallowed actions and a directed graph G over the set of state formulas that combines `sometime-before` relations and implications. That is, there is an edge from α to β in G iff either $\beta \,\mathsf{SB}\, \alpha \in C$ or $\alpha \rightarrow \beta$. Note the direction of the edge in the first case: it is from the "triggering side" of the constraint, i.e., α. Both of these relations are transitive, and the first step in the fixpoint loop (lines 9–11) is to add any missing transitively implied SB relation (this assumes that implications are already transitively closed). Whenever a new constraint $\mathsf{E}\alpha$ or $\mathsf{N}\alpha$ is derived, it is propagated along SB relations and implications. Likewise, when a new SB constraint is derived, existing N constraints are propagated through it.

Algorithm 1 Trajectory Constraint Propagation

1: **procedure** PROPAGATE(C, X)
2: Let $F = \{$state formulas in C and $X\}$.
3: Set $D = \{a \mid \mathsf{A}\alpha \in C$ and $\left(\bigwedge_{p \in \mathrm{del}(a)} \neg p\right) \rightarrow \neg\alpha\}$.
4: **for** each $\mathsf{A}\alpha \in C$ **do**
5: **for** each $\beta \in F$ such that $\mathrm{mutex}(\alpha, \beta)$ **do**
6: ASSERTNEVER(β)
7: Set $G = \langle F, \{(\alpha, \beta) \mid \beta\,\mathsf{SB}\,\alpha \in C$ or $\alpha \rightarrow \beta\}\rangle$.
8: **repeat**
9: **for** each $(\alpha, \beta), (\beta, \gamma) \in G$ **do**
10: **if** $(\alpha, \gamma) \notin G$ **then**
11: ASSERTSB(γ, α).
12: **for** each $\alpha, \beta \in F$ **do**
13: **if** $\alpha\,\mathsf{SB}\,\beta \in C$ and $\beta\,\mathsf{SB}\,\alpha \in C$ **then**
14: ASSERTNEVER(α).
15: **if** $\alpha\,\mathsf{SB}\,\beta \in C$ and $\beta\,\mathsf{NA}\,\alpha \in C$ **then**
16: ASSERTNEVER(β).
17: **for** each $\langle \alpha\,\mathsf{SB}\,\beta, \overline{A}\rangle \in X$ **do**
18: **if** $\overline{A} \subseteq D$ **then**
19: ASSERTSB(α, β).
20: **for** each $\langle \mathsf{N}\alpha, \overline{A}\rangle \in X$ **do**
21: **if** $\overline{A} \subseteq D$ **then**
22: ASSERTNEVER(α).
23: **until** no change.
24: **for** each $\mathsf{E}\alpha \in C$ and $\mathsf{F}\alpha \in C$ **do**
25: ASSERTSOMETIME(α).
26: **if** $\exists\alpha$ such that $\mathsf{E}\alpha \in C$ and $\mathsf{N}\alpha \in C$ **then**
27: **return** *contradiction*.
28: **if** $\exists\alpha, \beta$ s.t. $\alpha\,\mathsf{NA}\,\beta, \beta\,\mathsf{NA}\,\alpha, \mathsf{E}\alpha, \mathsf{E}\beta \in C$ **then**
29: **return** *contradiction*.
30: **if not** CHECKAMO(C, D) **then**
31: **return** *contradiction*.
32: **return** C.

33: **procedure** ASSERTSB(α, β)
34: Add $\alpha\,\mathsf{SB}\,\beta$ to C and (β, α) to G.
35: **if** $\mathsf{N}\alpha \in C$ and $\mathsf{N}\beta \notin C$ **then**
36: ASSERTNEVER(β).

37: **procedure** ASSERTNEVER(α)
38: Add $\mathsf{N}\alpha$ to C.
39: **for** each action a **do**
40: **if** $\mathrm{pre}(a) \rightarrow \alpha$ or $\mathrm{add}(a) \rightarrow \alpha$ **then** Add a to D.
41: **for** each $\alpha\,\mathsf{SB}\,\beta \in C$ **do**
42: **if** $\mathsf{N}\beta \notin C$ **then** ASSERTNEVER(β).
43: **for** each $\beta \in F$ such that $\beta \rightarrow \alpha$ **do**
44: **if** $\mathsf{N}\beta \notin C$ **then** ASSERTNEVER(β).

45: **procedure** ASSERTSOMETIME(α)
46: Add $\mathsf{E}\alpha$ to C.
47: **for** each $\beta\,\mathsf{SB}\,\alpha \in C$ **do**
48: **if** $\mathsf{E}\beta \notin C$ **then** ASSERTSOMETIME(β).
49: **for** each $\beta \in F$ such that $\alpha \rightarrow \beta$ **do**
50: **if** $\mathsf{E}\beta \notin C$ **then** ASSERTSOMETIME(β).

A cyclic SB relation becomes unsatisfiable if any formula in the cycle is ever true. Similarly, a cycle between SB and NA relations implies that the trigger formula can never hold. When a state formula is proven unachievable (i.e., $\mathsf{N}\alpha$ is derived), the set of disallowed actions is updated with actions whose preconditions or add effects imply the formula (subroutine ASSERTNEVER, lines 39–40). The last step in the fixpoint loop (lines 17–22) is to check if the set of disallowed actions triggers any conditional constraint, which may result in further SB or N constraints becoming active.

Proposition 3 *If implications derived between state formulas are closed under transitivity,* PROPAGATE *is correct.*
Proof: The correctness of the CHECKAMO procedure is shown separately in Proposition 4 below. Hence, we consider only other inferences made by PROPAGATE:

Line 3: $\mathsf{A}\alpha$ and $\left(\bigwedge_{p \in \mathrm{del}(a)} \neg p\right) \rightarrow \neg\alpha$ entail $\mathsf{D}a$.

If a appears in the action sequence, then the negation of every atom in $\mathrm{del}(a)$ holds in the state immediately after. If this implies $\neg\alpha$, clearly α does not hold in every state, contradicting $\mathsf{A}\alpha$.

Lines 4–6: $\mathsf{A}\alpha$ and $\mathrm{mutex}(\alpha, \beta)$ entail $\mathsf{N}\beta$. Obvious.

Lines 9–11: Since implications are already transitively closed, both edges (α, β) and (β, γ) cannot be implications; thus at least one of $\beta\,\mathsf{SB}\,\alpha$ and $\gamma\,\mathsf{SB}\,\beta$ is in C. If both are, then for any state sequence $\vec{s}$ satisfying C, if $s_i \models \alpha$ there exists a $j < i$ such that $s_j \models \beta$ (otherwise $\vec{s} \not\models \beta\,\mathsf{SB}\,\alpha$), and therefore there exists a $k < j$ such that $s_k \models \gamma$ (otherwise $\vec{s} \not\models \gamma\,\mathsf{SB}\,\beta$). Thus, if any state satisfies α there must be an earlier state satisfying γ. Hence $\vec{s} \models \gamma\,\mathsf{SB}\,\alpha$.

Suppose $\beta\,\mathsf{SB}\,\alpha$ is in C and $\beta \rightarrow \gamma$. If $s_i \models \alpha$, there is a $j < i$ such that $s_j \models \beta$; by the implication $s_j \models \gamma$ as well. If instead $\gamma\,\mathsf{SB}\,\beta$ is in C and $\alpha \rightarrow \beta$, then if $s_i \models \alpha$, then $s_i \models \beta$ by implication; thus there is a $j < i$ such that $s_j \models \gamma$.

Lines 13–14: $\alpha\,\mathsf{SB}\,\beta$ and $\beta\,\mathsf{SB}\,\alpha$ entail $\mathsf{N}\alpha$. By the transitivity of SB shown above, the cyclic SB relation implies $\alpha\,\mathsf{SB}\,\alpha$. This implies that if α is ever true, it would also have to be true in an earlier state; thus, there can be no first state in which α holds. Hence, α holds in no state. (The cycle also entails $\mathsf{N}\beta$, but this is added by ASSERTNEVER.)

Lines 15–16: $\alpha\,\mathsf{SB}\,\beta$ and $\beta\,\mathsf{NA}\,\alpha$ entail $\mathsf{N}\beta$. Suppose $\vec{s} \models \{\alpha\,\mathsf{SB}\,\beta, \beta\,\mathsf{NA}\,\alpha\}$, and that $s_i \models \beta$ for some state s_i in $\vec{s}$. There is a $j < i$ such that $s_j \models \alpha$ (otherwise $\vec{s} \not\models \alpha\,\mathsf{SB}\,\beta$). But since $\vec{s} \models \beta\,\mathsf{NA}\,\alpha$, this implies $s_k \not\models \beta$ for all $k \geq j$. Hence $s_i \not\models \beta$.

Lines 17–22: If φ holds conditional on $\overline{A}$, then $\left(\bigwedge_{a \in \overline{A}} \mathsf{D}a\right) \rightarrow \varphi$ by definition.

Line 24–25: Goals that must hold in the final state ($\mathsf{F}\alpha$) must also hold at some point in the execution.

Lines 26–27: $\mathsf{E}\alpha$ and $\mathsf{N}\alpha$ are contradictory. This is immediate from their definitions in Table 1.

Lines 28–29: $\alpha\,\mathsf{NA}\,\beta$ and $\beta\,\mathsf{NA}\,\alpha$ entail $\neg\mathsf{E}\alpha \vee \neg\mathsf{E}\beta$. $\alpha\,\mathsf{NA}\,\beta$ and $\beta\,\mathsf{NA}\,\alpha$ means that α and β are mutex, i.e., that there is no reachable state in which both are true. Thus, they cannot be achieved at the same time. If α becomes true at any point,

Algorithm 2 Checking `at-most-once` Constraints

1: **procedure** CHECKAMO(C, D)
2: Set AMO_Acts $= \emptyset$. // AMO_Acts is a set of sets
3: **for** each AMO$\alpha \in C$ **do**
4: **if** $s_0 \models \alpha$ **then**
5: Add ActChF(α) to AMO_Acts.
6: Set $D = D \cup \mathrm{ActChT}(\alpha)$.
7: **else**
8: Add ActChF(α) and ActChT(α) to AMO_Acts.
9: Let Cands $= \{p \mid s_0 \not\models p, \exists \mathsf{E}\alpha \in C : \alpha \rightarrow p\}$.
10: Set sets $= \emptyset$. // sets is a set of sets of sets
11: **for** each $p \in$ Cands **do**
12: Let sets(p) be the smallest $\{A_1, \ldots, A_m\} \subseteq$ AMO_Acts s.t. $(\mathrm{Adds}(p) - D) \subseteq \bigcup_{i=1,\ldots,m} A_i$.
13: Add sets(p) to sets.
14: **for** each $S = \{A_1, \ldots, A_m\} \in$ sets **do**
15: Let $R = \{p \mid \mathrm{sets}(p) = S\}$.
16: Let $G = \langle R, \{(p, q) \mid \exists a \notin D : p, q \in \mathrm{add}(a)\}\rangle$.
17: Let $R' =$ APXINDEPENDENTSET(G).
18: **if** $|R'| > |S|$ **then**
19: **return** *false*.
20: **return** *true*.

β cannot be achieved later, and vice versa. Hence, at most one of Eα and Eβ can be satisfied.

Lines 35–36, 41–42: α SB β and Nα entail Nβ. If $s_i \models \beta$ for any i, there must be a $j < i$ such that $s_i \models \alpha$. This contradicts Nα.

Lines 39–40: Nα and $\mathrm{pre}(a) \rightarrow \alpha$ entail Da; Nα and $\mathrm{add}(a) \rightarrow \alpha$ entail Da. If a appears in the action sequence, $\mathrm{pre}(a)$ holds in the state where it is applied and $\mathrm{add}(a)$ in the state immediately after. If either implies α, this contradicts Nα.

Lines 43–44: Nα and $\beta \rightarrow \alpha$ entail Nβ. Obvious.

Lines 47–48: Eα and β SB α entail Eβ. If $\vec{s} \models \mathsf{E}\alpha$ then $s_i \models \alpha$ for some state s_i in $\vec{s}$. Since $\vec{s} \models \beta$ SB α, this implies that $s_j \models \beta$ for some $j < i$. Hence $\vec{s} \models \mathsf{E}\beta$.

Lines 49–50: Eα and $\alpha \rightarrow \beta$ entail Eβ. Obvious. □

The AMOα constraint states that α may be true in at most one contiguous subsequence of states. That is, if α is true at some point and later becomes false, it may not become true again. The procedure for checking unsatisfiability of this constraint type, shown in Algorithm 2, is based on counting. Each state formula α that appears in an AMO constraint is associated with two sets of actions: $\mathrm{ActChF}(\alpha) = \{a \mid \mathrm{pre}(a) \rightarrow \alpha, \left(\bigwedge_{p \in \mathrm{del}(a)} \neg p\right) \rightarrow \neg\alpha\}$ and $\mathrm{ActChT}(\alpha) = \{a \mid \mathrm{mutex}(\mathrm{pre}(a), \alpha), \mathrm{add}(a) \rightarrow \alpha\}$. Actions in $\mathrm{ActChF}(\alpha)$, when applied, necessarily change the value of α from true to false, and actions in $\mathrm{ActChT}(\alpha)$ change it from false to true. The AMOα constraint implies that at most one action in each of these sets can appear in any plan. (In fact, if α is initially true, no action in $\mathrm{ActChT}(\alpha)$ can appear in the plan.) Next, we find a set of atoms that are not initially true but implied by existing E constraints, i.e., atoms that must be achieved at some point, and such that the set of still allowed actions that add each of them is covered by the union of at-most-once action sets, $\mathrm{ActChF}(\alpha)$ and $\mathrm{ActChT}(\alpha)$, where AMO$\alpha \in C$. These atoms are grouped into sets whose achievers are covered by the same at-most-once action sets, and from each a subset such that no action adds two atoms in the subset is found. This amounts to solving a independent set problem over the (undirected) graph that has the atoms in the set as nodes and an edge between two atoms iff there is a, still allowed, action that adds both. Since finding a maximal independent set is NP-hard, it is solved with an approximation algorithm (Boppana and Halldórsson 1992). If the size of such a set is greater than the number of at-most-once action sets that covers its achievers, we have a contradiction.

Proposition 4 *If* CHECKAMO(C, D) *returns false, no sequence of actions satisfies* $C \cup \{\mathsf{D}a \mid a \in D\}$.

Proof: We first establish that any sequence of actions $a_1, \ldots, a_n$ satisfying AMOα:

(1) contains at most one action from $\mathrm{ActChF}(\alpha)$;
(2) contains at most one action from $\mathrm{ActChT}(\alpha)$; and
(3) if $s_0 \models \alpha$, contains no action from $\mathrm{ActChT}(\alpha)$.

Suppose a_i and a_l ($i < l$) both belong to $\mathrm{ActChF}(\alpha)$. By construction of $\mathrm{ActChF}(\alpha)$, this means $s_{i-1} \models \alpha$, $s_i \not\models \alpha$, $s_{l-1} \models \alpha$, and $s_l \not\models \alpha$. Note that $l - 1 > i$, since s_i and s_{l-1} cannot be the same state. However, since $s_{i-1} \models \alpha$, the AMOα constraint requires that there is a $j \geq i - 1$ such that $s_k \models \alpha$ for all $i - 1 \leq k \leq j$ and $s_k \not\models \alpha$ for all $k > j$. Any choice of $j > i - 1$ violates the first condition, since $s_i \not\models \alpha$. But chosing $j = i-1$ violates the second condition, since $s_{l-1} \models \alpha$ and $l - 1 > i - 1$. This shows (1).

For (2), suppose a_i and a_l ($i < l$) both belong to $\mathrm{ActChT}(\alpha)$. Similar to the previous case, this means $s_{i-1} \not\models \alpha$, $s_i \models \alpha$, $s_{l-1} \not\models \alpha$, and $s_l \models \alpha$. By the same argument as above, this contradicts AMOα.

For (3), suppose $s_0 \models \alpha$ and that a_i belongs to $\mathrm{ActChT}(\alpha)$. This means $s_{i-1} \not\models \alpha$ and $s_i \models \alpha$. Note that $i > 1$, since s_0 and s_{i-1} cannot be the same state. Since $s_0 \models \alpha$ and $s_{i-1} \not\models \alpha$ $s_k \not\models \alpha$ must hold for all $k > i - 1$ for AMOα to be satisfied. But this is contradicted by $s_i \models \alpha$.

Suppose the condition of the **if** statement on line 18, $|R'| > |S|$, is true. From (1) and (2) above, no more than $|S|$ actions from the set $\bigcup_{A_i \in S} A_i$ can appear in any action sequence satisfying C. Also to satisfy C, each atom in R' must be achieved. Since no action adds more than one atom in R', this means at least $|R'|$ actions that add some atom in R' must take place. But all actions that add some atom in R' and that are not disallowed are contained in $\bigcup_{A_i \in S} A_i$. Clearly, no action sequence can contain both at least $|R'|$ and at most $|S|$ actions from this set. □

Proposition 4 refers only to *sequences* of actions. Since PDDL3 constraints are evaluated over the sequence of states visited by a plan, they can, in some situations, be satisfied by a parallel plan even when not satisfiable by any sequential plan, because the parallel plan does not visit the states that occur where parallel actions are interleaved (Gerevini

et al. 2009, Section 2.4.2). However, no two actions in $\mathrm{ActChF}(\alpha)$ can occur in parallel, since they all destroy each other's preconditions. Likewise, no two actions in $\mathrm{ActChT}(\alpha)$ can take place in parallel: Since α holds after applying any action in $\mathrm{ActChT}(\alpha)$, the action must destroy the precondition of every action in $\mathrm{ActChT}(\alpha)$, as otherwise the mutex relation between α and those preconditions would not hold. Hence, contradictions found by CHECK-AMO are valid also if we consider parallel plans.

Evaluation

The trajectory constraint propagator was designed with the problem of computing lower bounds for problems in the Rovers QualitativePreferences domain in mind, so it is natural to test it in this setting. Some limitations of the current implementation (e.g., that all state formulas are atoms, and not considering `sometime-after` constraints) are also due to this particular problem set.

The method previously used to test unsatisfiability of a constraint set (Haslum 2007) was to compile the constraints into the problem, i.e., to create a modified problem P' such that any plan for P' satisfies the constraints, and check unsolvability of the resulting problem with the h^m admissible heuristic (Haslum and Geffner 2000), with $m = 1$ or 2. The compilation is also somewhat specialised for the constraints that appear in the Rovers QualitativePreferences problem set, and for use with the h^m test for unsolvability.

With both tests, unsatisfiable constraint sets are found by simply enumerating and testing subsets of constraints in the problem in order of increasing size, skipping sets that contain a subset already proven unsatisfiable. End-state goals (`at-end` constraints) are included in every test. With the compilation-based test, `sometime` constraints are treated in a special way for efficiency reasons. As a result, the two methods do not test the exact same subsets of constraints.

Compilation of Trajectory Constraints

PDDL3 trajectory constraints can be compiled away, with a polynomial increase in problem size (Gerevini et al. 2009). The compilation used in the proof of this is based on converting the constraint to an automaton, accepting exactly the state sequences that satisfy the constraint, encoding the automaton into the problem, and posing its acceptance as a goal. Variations of this compilation have been used in some planners supporting PDDL3 (e.g. Edelkamp 2006; Baier and McIlraith 2006).

The compilation we have used is simplified, mainly by restricting it to trajectory constraints in which state formulas are single atoms. It is also somewhat tailored to support inference by the h^m heuristic, which is used to detect unsolvability of the compiled problem. We assume that the constraints to be tested are not trivially satisfied or contradicted, e.g., for Ap that p is not initially false and for Ep and p SB q that p is not initially true. The compiled problem, P' is an ordinary planning problem, with an end-state goal. For each constraint, P' is modified as follows:

Fp: p becomes a goal.

Ap: Remove from P' any action with $p \in \mathrm{del}(a)$.

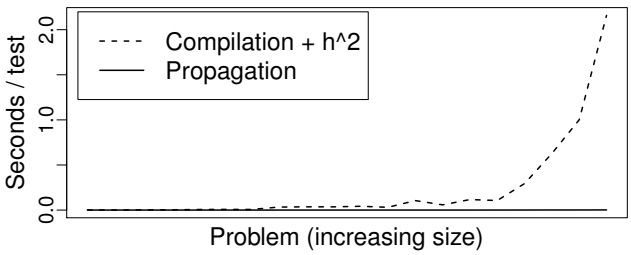

Figure 1: Time per unsatisfiability test, using compilation and the h^2 heuristic and using propagation.

Ep: Add a new atom had-p, and add had-p to the add effects of any action with $p \in \mathrm{add}(a)$. had-p becomes a goal.

p SB q: Add a new atom had-p, and add had-p to the add effects of any action with $p \in \mathrm{add}(a)$. Add had-p to the precondition of any action with $q \in \mathrm{add}(a)$.

AMOp: First, for each action that adds or deletes p, ensure the action is "toggling" w.r.t. p. That is, if the action deletes p, its precondition must include p and if it adds p its precondition must be mutex with p. This property can be enforced by splitting non-toggling actions into two cases (Hickmott et al. 2007). Next, add a new, initially true, atom first-p, and add first-p to the delete effects of any action with $p \in \mathrm{del}(a)$, and to the precondition of any action with $p \in \mathrm{add}(a)$.

Testing a constraint set C with the compilation-based method proceeds in three steps: First, the constraints are compiled to produce problem P'. Second, the h^m heuristic is computed from the initial state of this problem. Third, the problem goals are evaluated with the heuristic. If $h^m(G') = \infty$, where G' is the goal set of P', C is unsatisfiable w.r.t. P.

The first two steps are relatively time-consuming, while the last is very quick. The h^m heuristic approximates the cost of achieving a set (i.e., conjunction) of atoms of size greater than m by the cost of the most expensive subset of size m. Thus, if $|G'| > m$, h^m will only detect unsolvability if there is a m-subset of G' that is also unsolvable. This is the reason for the special treatment of `sometime` constraints: Since these are translated into end-state goals, it is not necessary to perform a separate compilation for each subset of them. Instead, all `sometime` constraints (together with some subset of other constraints) are compiled, and each subset of at most m of them (together with `at-end` goals) tested in the heuristic evaluation step.

Results

There are 20 problems in the test set. The number of soft trajectory constraints varies from 14 to 274, and tends to increase with problem size. The compilation-based test is applied to every subset of at most 2 constraints of types other than `sometime`, and every combination of one of these sets with at most m `sometime` constraints. The propagation-based test is applied to every subset of at most 3 constraints. For the largest problem that is potentially over 20 million tests. However, because sets that contain a subset already proven to be unsatisfiable are skipped, only 3.4 million tests

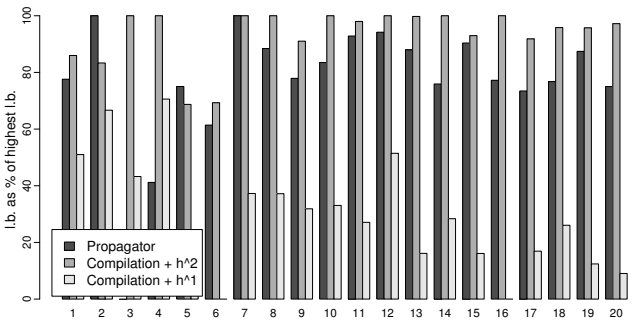

Figure 2: Lower bounds computed from unsatisfiable constraint sets found by the propagation-based and compilation-based tests, as a percentage of the highest lower bound for each problem. The highest lower bound was obtained by combining the results of both tests.

are actually needed for this problem.

The propagation-based test is faster than the compilation-based test with the h^2 heuristic, and although it does grow with problem size, it does so much more slowly. The difference ranges from about 10 times faster to over 1000 times faster on the largest problems. This is shown in Figure 1. However, because of the separate treatment of `sometime` constraints, the compilation-based method performs far fewer tests. As a result, the reduction in total runtime is only a factor that ranges from 1.85 to 4.77 (median 2.89). On two problems, the compilation-based method is faster (by a factor less than 2).

The compilation-based test with h^2 is generally more powerful than the propagation-based method: it finds more unsatisfiable constraint sets for 15 problems, and a higher lower bound for 17 problems. The lower bounds computed from the unsatisfiable sets found by the propagation-based method are, however, not far below: with the exception of two problems, they are above 75% of higher bound. There is, however, also a strong complementarity between the two methods: For 16 of the 20 problems, both methods find some unsatisfiable set that is not found by the other, and for 14 problems, the lower bound computed from the union of unsatisfiable sets found by both methods dominates the bounds produced by either method alone. Figure 2 displays this in detail. The majority of unsatisfiable constraint sets found by propagation but not with compilation and h^2 include at least one `at-most-once` constraint. This is because the CHECKAMO procedure can find contradictions implied by a more than two subgoals, as shown by the following example.

Example 1 Rovers QualitativePreferences problem #6 has, i.a., the following goals[1]:

(1) `(sent-image obj0 col)`
(2) `(sent-image obj0 LR)`
(3) `(sent-image obj1 LR)`

To achieve a `sent-image` goal, some rover must take the image (achieving `(have-image ?rover ?obj ?mode)`), and then send it. The `take-image` action, requires, among other preconditions, `(calibrated ?camera)`, which is also deleted by the action (i.e., a camera must be recalibrated before each photo), and `(supports ?camera ?mode)`.

In problem #6, there are two rovers, `rover0` and `rover1`. `rover0` has two cameras, `cam0` and `cam1`, which support both image modes; `rover1` has one camera, supporting only mode `col`. Thus, landmark analysis infers the constraints

(4) `(sometime-before (sent-image obj0 LR) (have-image rover0 obj0 LR))`
(5) `(sometime-before (sent-image obj1 LR) (have-image rover0 obj1 LR)).`

Now, consider the constraints

(6) `(at-most-once (calibrated cam0))`
(7) `(at-most-once (calibrated cam1))`
(8) `(sometime (have-image rover0 obj0 col)).`

From (2) and (4), and (3) and (5), the propagator derives

(9) `(sometime (have-image rover0 obj0 LR))`
(10) `(sometime (have-image rover0 obj1 LR)).`

Hence, the set Cands in CHECKAMO contains, i.a.,

p_1: `(have-image rover0 obj0 LR)`
p_2: `(have-image rover0 obj1 LR)`
p_3: `(have-image rover0 obj0 col)`

$A_1 = \mathrm{ActChF}(\texttt{(calibrated cam0)})$, the set of actions that change `(calibrated cam0)` from true to false, consists of all `take-image` actions using `cam0`; likewise, $A_2 = \mathrm{ActChF}(\texttt{(calibrated cam1)})$ consists of all `take-image` actions using `cam1`. Since these are the only two cameras on `rover0`, these two sets together contain all actions that add each of the three candidate atoms; thus $\mathrm{sets}(p_i) = \{A_1, A_2\}$ for $i = 1, 2, 3$. Thus, we have $S = \{A_1, A_2\}$, and $R = \{p_1, p_2, p_3\}$. Since no action adds more than one of the candidate atoms, the independent set problem is trivial (the graph has no edges), so $R' = R$. Since $|R'| = 3 > 2 = |S|$, CHECKAMO finds a contradiction.

The propagation-based test also finds a few contradictions involving more than two `sometime-before` constraints. The compilation-based test with the h^1 heuristic is fast, but also much weaker.

Discussion

The design goal for the PDDL3 propagator was to have a sound, though incomplete, test for unsatisfiability of a trajectory constraint set w.r.t. a planning problem, whose time complexity is not strongly related to the size of the problem. This led to a two-stage approach, where relevant information is extracted from the problem in a preprocessing step, and passed to the propagator in the form additional constraints. Results on problems from the IPC 2006 Rovers QualitativePreferences domain confirm that this goal has been largely met.

A limitation of the current propagator is that it makes no use of `sometime-after` constraints. This constraint stands out in that it is the only one that is not finitely satisfiable. For any constraint set C that does not include a `sometime-after` constraint, if C is satisfied by any execution of a problem P, it is also satisfied by a *finite* execution:

[1]The names of some predicates have been changed to shorten them, and to make the example easier to grasp.

Because the number of possible states is finite, in any infinite execution there is an index i such that all states that appear in the sequence appear in the finite sequence up to i, and this finite sequence satisfies all constraints in C. The `sometime-after` constraint does not have this property: If α and β are mutually exclusive state formulas, the constraint set $\{\alpha \,\mathsf{SA}\, \beta, \beta \,\mathsf{SA}\, \alpha\}$ is satisfied by a state sequence that alternates infinitely between states where α and β hold, but it is not satisfied by any finite sequence.

Like SB, the SB constraint is transitive: $\beta \,\mathsf{SA}\, \alpha$ and $\gamma \,\mathsf{SA}\, \beta$ entail $\gamma \,\mathsf{SA}\, \alpha$. It is also propagated by implication, from the right-hand side to the left. Because it is not strict, however, a cycle of SA constraints do not entail that the state formulas in the cycle can never hold. From $\alpha \,\mathsf{SA}\, \beta$ and $\beta \,\mathsf{SA}\, \alpha$ we can only infer that in any finite execution, $\alpha \wedge \beta$ must hold at some point. Applying this inference rule would mean that the set of state formulas handled by the propagation algorithm is no longer restricted to those that appear in its input; in fact, it may grow exponentially. Of course, we could limit inference to checking whether any pair of state formulas that appear in a SA-cycle are mutex.

To the best of my knowledge, this is the first approach aimed specifically at proving the unsatisfiability of PDDL3 trajectory constraints. Most planners have dealt with such constraints by compiling them away. An exception is the work of Bienvenu, Fritz and McIlraith (2006), which deals with preferences over trajectory constraints (more general than those expressible in PDDL3) using progression. Their optimistic evaluation provides a lower bound, but a rather weak one, since it assumes that any constraint that has not been irrecoverably violated by the state sequence so far will be satisfied. Baier, Bacchus and McIlraith (2009) describe an admissible heuristic, based on delete-relaxed plans, for planning with preferences. In combination with compilation it can provide lower bounds for problems with soft trajectory constraints, probably comparable to those obtained using the compilation described above with the h^1 heuristic. Resolution-based proof procedures for the full linear temporal logic have been developed in the area of formal methods (Fisher, Dixon, and Peim 2001). These methods are complete, and hence necessarily of high complexity.

Although the propagator is designed to prove unsatisfiability of trajectory constraints w.r.t. a planning problem, it could potentially also detect unsolvability of a problem without trajectory constraints, by applying it to just the constraints extracted from the problem and the problem's goal.

Acknowledgements NICTA is funded by the Australian Government represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

Baier, J., and McIlraith, S. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proc. 21st National Conference on AI (AAAI'06)*.

Baier, J.; Bacchus, F.; and McIlratih, S. 2009. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence* 173(5–6):593–618.

Bauer, A., and Haslum, P. 2010. LTL goal specifications revisited. In *Proc. 19th European Conference on Artificial Intelligence (ECAI'10)*.

Bienvenu, M.; Fritz, C.; and McIlraith, S. 2006. Planning with qualitative temporal preferences. In *Proc. 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, 134–144.

Boppana, R., and Halldórsson, M. 1992. Approximating maximum independent sets by excluding subgraphs. *BIT* 32(2).

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *Proc. of the 16th International Conference on Automated Planning and Scheduling (ICAPS'06)*, 374–377.

Fisher, M.; Dixon, C.; and Peim, M. 2001. Clausal temporal resolution. *ACM Transactions on Computational Logic* 2(1):12–56.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proc. 15th National Conference on Artificial Intelligence (AAAI'98)*, 905–912.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence* 173(5-6):619–668.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 140–149. AAAI Press.

Haslum, P. 2007. Quality of solutions to IPC5 benchmark problems: Preliminary results. In *ICAPS'07 Workshop on the IPC*.

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

Hickmott, S.; Rintanen, J.; Thiébaux, S.; and White, L. 2007. Planning via Petri net unfolding. In *Proc. 20th International Conference on Artificial Intelligence (IJCAI'07)*, 1904–1911.

Pnueli, A. 1977. The temporal logic of programs. In *Proc. of the 18th Symposium on Foundations of Computer Science (FOCS'77)*, 46–57.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering and usage of landmarks in planning. In *Proc. 6th European Conference on Planning (ECP'01)*.

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *Proc. 17th National Conference on Artificial Intelligence (AAAI'00)*, 806–811.

Vidal, V., and Geffner, H. 2004. Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proc. 19th National Conference on Artificial Intelligence (AAAI'04)*, 570–577.