TPLP **19** (5–6): 1124–1142, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000395

Domain-Independent Cost-Optimal Planning in ASP

DAVID SPIES, JIA-HUAI YOU and RYAN HAYWARD
University of Alberta, Edmonton, Canada

submitted 31 July 2019; accepted 2 Aug 2019

Abstract

We investigate the problem of cost-optimal planning in ASP. Current ASP planners can be trivially extended to a cost-optimal one by adding weak constraints, but only for a given makespan (number of steps). It is desirable to have a planner that guarantees global optimality. In this paper, we present two approaches to addressing this problem. First, we show how to engineer a cost-optimal planner composed of two ASP programs running in parallel. Using lessons learned from this, we then develop an entirely new approach to cost-optimal planning, *stepless planning*, which is completely free of makespan. Experiments to compare the two approaches with the only known cost-optimal planner in SAT reveal good potentials for stepless planning in ASP.

KEYWORDS: Cost-Optimal Planning, Answer Set Programming, CORE-2 ASP Standard.

1 Introduction

Answer Set Programming (ASP) has been celebrated for its elegance and applicability to AI planning (Lifschitz 2002). A planning problem in this context is, given a collection of actions, each with pre-conditions and effects, and fluents which are properties over states, determine whether there is a sequence of actions from an initial state to a final state (Fikes and Nilsson 1971). State-of-the-art ASP planners like PLASP have been developed (Gebser et al. 2011; Dimopoulos et al. 2019).

In cost-optimal planning, each action is associated with a cost, and our objective is to find a plan which minimizes the sum cost of all actions. Any ASP planner can be trivially extended to a cost-optimal planner by adding weak constraints, but only for a given makespan. Such a planner does not guarantee global optimality. Eiter et al. (2003) present an approach to addressing this problem, but make the assumption that a polynomial upper bound on makespan exists and is known in advance.

In the related field of SAT, some work has been done on cost-optimal Partial-Weighted-MaxSat planning with regard to makespan (e.g., (Maratea 2012) and (Chen et al. 2008)). Again, readers should find this somewhat unsatisfactory. After all, finding a plan that is "cost-optimal with regard to makespan" is just a way to sidestep the complication the real problem presents. A makespan is just an internal artifact of the SAT approach to planning. A solution should not depend on the way in which the planner happens to order the actions. Ideally, we want an approach to planning which guarantees a globally optimal solution and makes no mention of makespan.

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

We are aware of only one existing logic-based approach that tackles this much more difficult problem for SAT planning, the paper by Robinson et al. (2010). Inspired by this work, we pursue a separate investigation into globally cost-optimal planning in ASP and develop a two-threaded planner, one thread being a regular planner and the other an *any-goal* planner. While the former computes successive decreasing upper bounds of the optimal cost by iteratively increasing the makespan, the latter computes successive increasing lower bounds by planning in a modified environment where some amount of cheating is allowed. We achieve this by forcing the any-goal planner to "make progress" at each time-step. An optimal plan is obtained when the two bounds meet in the middle, i.e., they are guaranteed to agree at some point. However, unlike in Robinson's approach, through the use of "make-progress" rules, we're able to develop a planner which guarantees to eventually find an optimal solution (or report no solution) even when the problem contains actions with zero cost.

Using insights gained from this approach, we then develop a new approach to logic-based planning, *stepless planning*. The idea is to first engineer a planner which produces "partially ordered plans" - actions arranged into a graph of dependencies where stable model semantics ensures that the graph is acyclic; then we show how to express the problem in Σ_2^P of "making progress"; and finally, we show a critical component of the planner, the *suffix layer*, which determines how many occurrences of each of the actions and fluents we will need to produce an optimal plan.

We report experiments on our cost-optimal planners on benchmarks of (Robinson et al. 2010) and compare with Robinson's SAT-based planner. We found that the stepless planner outperformed the other two planners in most domains and the two-threaded planner outperformed the SAT-based planner in most domains.

The paper is organized as follows. Section 2 describes ASP planning translated from SATPlan. Section 3 discusses the problem of no-solution detection and provides solutions. Section 4 extends no-solution detection to optimal planning, with Section 5 adding a *delete-free planner* as a suffix layer to improve the effectiveness. Section 6 gives the two-threaded planner and Section 7 is about the stepless planner. Section 8 reports experiments and Section 9 is on related work and future directions.

We assume that the reader is familiar with STRIPS planning. The ASP encodings in this paper are constructed to run on system CLINGO and generally follow the ASP-Core-2 Standard (Calimeri et al. 2015), except that we adopt two special features provided by CLINGO: (i) we will use ; to separate rule body atoms since the conventional comma sign , is overloaded and has a different meaning in more complex rules, and (ii) the disjunctive head of a rule may be expressed conveniently by a conditional literal.

Due to the page limit, parts of this paper have been moved to the online appendices, including proofs, encodings, and some technical explorations. For more information, the reader may also want to consult the thesis written by the first author of this paper (Spies 2019).

2 Preliminaries: STRIPS Planning in ASP

We adopt a direct translation of 5 rules of SATPlan (Kautz 2004) into ASP and call the resulting planner ASPPlan.

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

```
rule 1.  holds(F,K) :- goal(F); finalStep(K).
rule 2.  happens(A,K-1) : add(A,F),validAct(A,K-1) :- holds(F,K); K > 0.
rule 3.  holds(F,K) :- pre(A,F); happens(A,K); validFluent(F,K).
rule 4.  :- mutexAct(A,B); happens(A,K); happens(B,K).
rule 5.  :- mutex(F,G); holds(F,K); holds(G,K).
```

where $validAct(A, K)$ means that action A can occur at time K and $validFluent(F, K)$ means fluent F can be true at time K.[1] Time-steps used in constructing a plan are also called *layers*.

Rule 1 says that goals hold at the final layer. In rule 2, if a fluent holds at layer K, the disjunction of actions that have that fluent as an effect hold at layer $K - 1$. The next rule says that actions at each layer imply their preconditions. The last two rules are mutex constraints: in rule 4, actions with (directly) conflicting preconditions or effects are mutually exclusive, and in rule 5, the fluents that are inferred to be mutually exclusive are encoded as constraints.

Following SATPlan, we add to our plan "preserving" actions for each fluent. The goal is to simulate the frame axioms by using the existing machinery for having an action add a fluent that gets used some steps later. These preserving actions can be specified as:

```
action(preserve(F)) :- fluent(F).
pre(preserve(F),F) :- fluent(F).
add(preserve(F),F) :- fluent(F).
```

where each fluent F has a corresponding *preserving action* denoted by term $preserve(F)$. Preserving actions can be easily distinguished from regular actions. Now that an action occurs at time K indicates that its add-effect F will hold at time $K + 1$.

Note that the reason why rule 5 of ASPPlan prevents fluents from being deleted before they're used is a bit subtle. In order for a fluent to hold, it must occur in conjunction with a preserving action at each time-step it's held for. A preserving action has that fluent as a precondition and so would be mutex with any action that has it as a delete effect. This means that deleting actions cannot occur as long as that fluent is held (by rule 4).[2]

Like SATPlan, we run this planner by solving at some initial makespan K, where K is the first layer at which `validFluent(F,K)` holds for all `goal(F)`, and if it is UNSAT, we increment $finalStep$ by 1 until we find a plan.

This is a straightforward and unsurprising encoding in every respect, but has a somewhat surprising consequence as compared to SATPlan. Because ASP models are stable, for any fluent F, $holds(F, K)$ can only be true if there exists some action which requires its truth as per rule 3. Similarly for actions as per rule 2. Furthermore, since rule 2 is disjunctive at every step, the set of actions which occurs is a minimal set required to

[1] Blum and Furst (1997) give a handy way to identify for each action and each fluent, what is the first layer at which this action/fluent might occur by building the *planning graph*. Note that validAct/2 and validFluent/2 as well as predicates mutexAct/2 and mutex/2 are all extracted from the planning graph.

[2] As a further note, when PDDL (planning domain definition language) without any extensions is defined, goals can only be positive and actions can only have positive preconditions. There is a `:negative-preconditions` extension to PDDL, but we didn't use it. Any problem which uses :negative-preconditions can be trivially adapted to avoid using it by adding a fluent :not-F for every fluent :F and then adding a corresponding add-effect wherever there's a delete-effect and vice versa.

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

support the fluents at the subsequent step. This conforms exactly to the approach to planning in (Blum and Furst 1997): First build the planning graph, then start from the goal-state planning backwards, at each step selecting a minimal set of actions necessary to add all the preconditions for the current set of actions. That is, in this ASP translation, the neededness-analysis as carried out in (Robinson et al. 2008) is accomplished automatically during grounding or during the search for stable models.

Encoding Reduction: Rule 4 in ASPPlan can blow up in size when grounded because nearly any two actions acting on the same fluent can be considered directly conflicting. For example, imagine a planning problem in which there is a crane which we must use to load boxes onto freighters and there are many boxes and many freighters available but only one crane. Then we will have one such constraint for every two actions of the form, $load(Crate, Freighter)$, for any crate and any freighter. As there is already a quadratic number of actions in the problem description size, the number of mutex constraints over *pairs* of actions is *quartic* in the initial problem description size.

We would like to avoid such an explosion by introducing new predicates to keep the problem size down. We will only consider two actions to be mutex if one deletes the other's precondition. But we will take extra steps to ensure that no add-effect is later used if the same fluent is also deleted at that step. Here is the revised encoding of rule 4.

```
used_preserved(F,K) :- happens(A,K); pre(A,F); not del(A,F).
deleted_unused(F,K) :- happens(A,K); del(A,F); not pre(A,F).
:- {used_preserved(F,K); deleted_unused(F,K);
    happens(A,K) : pre(A,F), del(A,F)} > 1; valid_at(F,K).

deleted(F,K) :- happens(A,K); del(A,F).
:- holds(F,K); deleted(F,K-1).
```

Effectively, we are splitting the ways in which we care that an action A can relate to a fluent F into three different cases: (i) A has F as a precondition, but not a delete-effect; (ii) A has F as a delete-effect, but not a precondition; and (iii) A has F as both a precondition and a delete-effect.

By explicitly creating two new predicates for properties (i) and (ii), we have packed this restriction into one big cardinality constraint. Further, we must account for conflicting effects, so we define one more predicate (`deleted/2`) which encapsulates the union of all actions from properties (ii) and (iii) (those that delete F) and assert that F cannot hold at this step if any of those actions occurred in the previous one.

Note that the formulation in (Blum and Furst 1997) will not allow two actions A and B to ever happen at the same time-step K if they have "conflicting effects" (one adds a fluent F and the other deletes the same fluent). Our encoding allows this, but only in cases where $holds(F, K)$ is false. Except for this technicality, the two are otherwise equivalent. A detailed justification is presented in Appendix B.

3 Planning with No-Solution Detection

From a theoretical standpoint, let us consider why cost-optimal planning is such a difficult problem. When the planner terminates with a plan of cost c, it is additionally asserting

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

"I have proved there does not exist a plan of cost $c-1$". But here we immediately have a problem because all the planners we have written so far in ASP are not actually *planners* in the sense of (Blum and Furst 1997); they cannot identify when a problem has no solution. If there is no solution our ASP planners will simply march on forever searching for one until somebody kills the process. So before we can create a cost-optimal planner, we must first create a (normal) planner which can determine if a problem has no solution.

Planning is PSPACE-complete. What makes planning decidable is, of course, the finite state space. Any plan which goes on too long will eventually visit some state twice, so we only need to search for a plan among those that never revisit the same state. We can, of course, produce a naive upper bound on the number of possible states by taking $n = 2^{|fluents|}$ and then terminate the search after n steps, but let us try to do better. Indeed, a main technical innovation of this work is the development of ASP-based decision procedures for planning, which can potentially prove the unsatisfiability of a planning problem much earlier than when taking the theoretical number of states as upper bound for the planning horizon.

First, instead of requiring goals to hold at final step (rule 1 of Section 2), let us say

```
{holds(F,K)} :- fluent(F); finalStep(K).
```

By using a choice rule here, the planner can now choose any goal it wants and then plan towards that goal. This makes our instance always satisfiable (just produce any valid sequence of actions, then take the end-state and claim that was your goal).

Now here comes the "we must make progress" rule. We'll refer to this as the "layered make-progress" rule (which also includes its strengthening to be discussed in the next section) to distinguish it from the "stepless make-progress rule" for stepless planning.

```
:- not holds(F,K) : not holds(F,J), fluent(F); step(J); step(K); J < K.
```

In English: "For any time-step pair J and K where $J < K$, we cannot allow that every fluent F which does not hold at J also does not hold at K.' That is, any sequence containing time-steps (J, K) $(J < K)$ where the state of K is a subset of that of J would fail this rule.

Thus, the rule guarantees that there exists a makespan n for which our planning instance is UNSAT. This is because we are now enforcing that the state must change at every time-step to take on some value which it did not have in any previous time-step. But if there are only m reachable states in our planning instance, then for all $n > m$ this is clearly impossible.

Hence, we can build a complete planner by running two separate computations in parallel. The first is our usual ASP planner which increases the makespan until it finds an instance for which the solver finds a plan. The second is our "any-goal" planner which increases the makespan until it finds an instance for which the solver returns UNSAT, at which point we record the previous makespan as *maxlength*. Once that is done, if the first instance manages to reach *maxlength* and report UNSAT, we can safely claim to have proved that no plan exists and terminate the solver.

3.1 Stronger Notions of Progress

Unfortunately, there exist problems that contain many independent variables which may be separately manipulated to generate a large easily-traversable state-space. For such

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

problems, our solver above can produce long plans which idly "flip bits" to avoid repeating themselves.

To make this scenario more concrete, imagine that we take any unsolvable planning problem and adjoin to it a binary counter with one hundred two-state switches. In addition to the actions from the original problem, we also have two hundred actions which independently flip each of the switches in the counter (either from 0 to 1 or from 1 to 0). Even though this counter has no impact on the problem itself, it suffices to increase the length of the longest plan by a factor of 2^{100} because for every state in the longest path, we can flip through all possible arrangements of these switches before proceeding to the next state. This easily puts the possibility of solving the problem out of reach whenever there is no solution.

One way to deal with this is to somehow encode into our planner the knowledge that the longest possible time it can take to iterate over the possible states of two independent subproblems is the maximum of the respective longest times rather than the product. One attempt at this is given in Appendix A, where we reduce the length of the longest plan in the above example to 100.

We want to do better though and we can. We are able to formulate a stronger definition of "make progress", which we conjecture perfectly defeats the *independent parts* problem in all its forms. First, a definition.

Definition 3.1
A *partially-ordered plan* is a transitive directed acyclic graph G (equivalently, a partial ordering) of "action occurrences" such that all topological sorts of G are valid sequential plans.

Starting with any sequential plan S, we can generalize it to its canonical partially-ordered plan as follows. If a precedes b in S, then we will say $a \prec b$ for actions a and b (adding an edge from a to b) iff any of the following holds:

1. a adds some fluent which is used as a precondition for b
2. b deletes some fluent which is used as a precondition for a (and $a \neq b$)
3. a adds a fluent which b deletes
4. a deletes some fluent and b adds the same fluent
5. a and b are different instances of the same action[3]
6. There exists an action c such that $a \prec c$ and $c \prec b$

Note that these rules only apply to a-b pairs for which a precedes b in S (otherwise, the statements 3 and 4 above would appear to be contradictory).

If we add a source and sink s and t respectively to any partially-ordered plan such that for all actions $s \prec a \prec t$, we can consider any s-t cut x as a generalized "intermediate state" for this plan. To see this, take any ordering where the s-side actions all precede the t-side actions and look at what fluents hold after we have taken only the s-side actions. Let us call this state x-state.

[3] Actually, this rule is not strictly necessary as any *minimal* plan which satisfies the other five rules *will also* satisfy it. By keeping it, we don't have to worry about discussing *action occurrences* until later when we actually start digging into the stepless planner.

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

Here comes the strongest possible (domain-independent) definition of "make progress" that we can think of. The idea is that for any set of actions identified as being sandwiched in between two cuts, they must make progress by turning some fluent to true.

Definition 3.2
A partially-ordered plan is *strongly minimal* iff, given any two s-t cuts x and y, if there exists any t-side action in x which is an s-side action in y, then there must be some fluent which is true in y-state but not in x-state. We similarly call a sequential plan strongly minimal if its canonical partially-ordered plan is.

An action is said to make progress if no two cuts exist on either side of the action without this property (that some new fluent occurs between them). In a strongly minimal plan, *all* actions make progress.

This beautifully handles the one hundred 3-state switch scenario by forcing us, for each switch w, to consider the generalized intermediate state where all w-flips happen before anything else. Thus if w is flipped to the same state twice we can produce the cuts at each of those states demonstrating that this plan is not minimal. The ASP encoding of this rule is complicated, and so will be relegated to Section 7 when we apply it to stepless planning.[4] This is the "stepless make-progress rule". In our two-threaded planner introduced next, we have chosen to implement the layered make-progress rule.

4 Extending No-Solution Detection to Cost-Optimality Detection

Now, let us add weak constraints for action costs to ASPPlan (called *MinASPPlan*[5]) as well as to the any-goal solver so that a plan with the least action cost can be identified by each solver for a given makespan. We run MinASPPlan and the any-goal solver in parallel.

Once the MinASPPlan solver produces a plan of cost C, we treat C as an upper-bound on the cost of an optimal plan. Then we tell the any-goal solver to only search for non-repeating plans of cost $< C$. When the any-goal solver terminates with UNSAT at some time-step n we claim that all non-repeating plans with cost $< C$ have a makespan $< n$ which means we can stop the MinASPPlan solver after finishing with makespan $n - 1$. Meanwhile, if the any-goal solver gives back a minimum-cost plan for layer n,[6] then that cost *is a lower bound* on the optimal cost of any plan with makespan at least n. Thus, by increasing makespan, MinASPPlan computes successive lower (i.e., non-increasing) upper bounds of optimal cost and the any-goal solver computes successive increasinglower bounds of the optimal cost; when the two meet in the middle, an optimal plan is identified.

Let us summarize what we have so far. We have two threads iteratively solve successively larger instances. We will name them I and II in deference to their similarity to Robinson's (Robinson et al. 2010) Variant-I and Variant-II encodings.

[4] The same technique *can* with some effort be encoded for layered planners as well. Essentially, it requires quite a bit of boilerplate in order to talk about *next* and *previous* occurrences of each fluent and action.
[5] Besides ASPPlan discussed here, it also includes a smart encoding of mutex constraints (cf. Chapter 6 of (Spies 2019)).
[6] Note that make-progress rules are crucial - without them, the minimum cost will always be zero.

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

- I is MinASPPlan; and
- II is the any-goal solver augmented by weak constraints by action costs. It is similar to I except for two major differences: (i) in place of the goal conditions, II is allowed to choose its own goal, and (ii) II is given some notion of progress together with the constraint that it *must make progress* at every time-step.

How we determine when to stop depends on which solver (I or II) lags behind. I's result costs are monotonically nonincreasing while II's result costs are monotonically nondecreasing. If I lags behind, then as soon as I's lowest cost is $\leq$ the II's cost for the layer it is currently trying to solve, we can stop and report the solution at that layer as optimal. If II lags behind, then as soon as its cost for some layer is $\geq$ the best known I-cost so far (at *any* layer), we can stop.

The asymmetry in deciding when to stop happens because of the types of bounds I and II produce. I will never produce a cost $D < C$ if C is the optimal cost, but II will continually increase its lower bound eventually marching straight past C and on to infinity (the point at which it returns no solution). This is why we must take into account the layer at which each lower bound was produced when determining if we are done, but we do not care what layer the upper bound was produced at.

5 Delete-Free Planning

The lower bounds produced by the Variant-II solver can be improved by adding a *suffix layer*, which is a *delete-free planner*. Delete-free planning (DFP) is a special case of planning which happens to be in NP. These are the planning problems without delete-effects. Surprisingly, DFP can be modeled as a graph problem.

Given a directed bipartite graph $G = (X, Y, E)$ with weights on X and a goal set $Y_F \subseteq Y$, find a minimum *acyclic* subgraph $G^* = (X^*, Y^*, E^*)$ such that

1. $Y_F \subseteq Y^*$
2. If $x \in X^*$ and $(y, x) \in E$, then $y \in Y^*$ and $(y, x) \in E^*$
3. For all $y \in Y^*$, E^* contains at least one edge (x, y) (and $x \in X^*$).

Connection to DFP: X is the set of actions, and Y is the set of fluents, the (x, y) edges are add-effects and the (y, x) edges are preconditions. Y_F is the goal set and the initial set has been removed (together with all corresponding preconditions) from the graph. Rule 1 means the goal fluents must be true. Rule 2 means an action implies its preconditions. Rule 3 means every fluent must have a causing action. The graph must be acyclic to ensure the actions can occur in some order. This is possible because there is no incentive to ever take an action or cause a fluent more than once. As soon as any fluent is true, it is permanently true.

We now can encode DFP in ASP as solving the above graph problem *independent of makespan*. Its encoding in ASP can be found in Appendix C.

The key takeaway is that the encoding is an efficient "one-shot" encoding in ASP. Rather than structuring the problem into layers and then iteratively increasing the makespan until a solution is found, we eschew layers entirely and encode the problem as a single ASP instance. This is similar to how (unlike with SAT encodings) Hamiltonian Path can be encoded one-shot in ASP (Spies 2019) without needing numeric fluents

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

or layers or quadratic space when grounded. The problem of ensuring that an acyclic structure exists is solved by relying on stable-model semantics.

This may be considered the most novel contribution of this paper. Besides for delete-free planning (and the suffix layer in the next section), the same trick will also be used later for "Stepless Planning" where we arrange actions and fluents into a graph and rely on stable-model semantics to ensure that the graph is acyclic. Without the ability to do this, other encodings of planning problems are forced to rely on either using numeric fluents (not compatible with SAT-based techniques) or structuring the problem into layers (which multiplies the grounded size of the problem by the number of layers needed).

6 A Two-Threaded Cost-Optimal Planner with Suffix Layer

As with A^*-search, we can generate successively better lower bounds by planning normally from the starting state Q to some intermediate state S chosen by the planner and then finding the minimum-cost solution to the delete relaxation for the planning problem from S to the goal state.[7] This suggests a natural way to modify our Variant-II encoding in Section 4 to find better lower bounds. We append a "suffix layer" at the end, which must generate a plan in the delete relaxation of the problem from the chosen any-goal state to the actual goal state. The costs for any actions taken in the suffix layer must be added to the total cost of our plan. Indeed, in many cases this produces a remarkable lower bound.

We now give a complete description of our two-threaded planner. We have two ASP programs running in parallel. One is the Variant-I standard ASPPlan solver with weak constraints for action costs. The other is the Variant-II solver with a progress rule and appended a suffix layer.

- Both solvers independently run successively on makespan 0, 1, 2 etc. until we kill them.
- Each time the Variant-I solver begins solving a new makespan, we update the current makespan being solved for.
- When the Variant-I solver finds a plan, we record the plan and its cost if this is the lowest-cost plan found so far.
- When the Variant-II solver finds an optimal plan for some makespan using the suffix layer, we record the optimal cost as a lower bound for that makespan (as well as all larger makespans).
- If the Variant-II solver ever finds an optimal plan which *doesn't* use the suffix layer, then that plan is globally optimal. We can return it as a solution and entirely ignore the Variant-I solver (this only happened twice in all of our experiments and seems to be fairly unlikely).
- If the Variant-II solver obtains UNSAT for a layer, we can stop running it and record the cost of that and all future layers as ∞.
- Any time the best-cost plan found so far (by the Variant-I solver) is no greater than the Variant-II lower bound for the currently-solving layer or *any* earlier layer, we can stop both solvers and report that plan as an optimal solution.

[7] More precisely, we encode in ASP the problem of finding the minimum *total* cost across all possible subgoal states of $cost(normalplan)+cost(suffixplan)$ (given that the normal plan respects whichever progress rule we choose to employ)

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

- Any time the Variant-I solver is solving for a makespan whose Variant-II lower bound is ∞, we stop the solver and return the best-cost plan found so far or "no solution" if no plan has been found.

The correctness of our two-threaded planner depends on the correctness of two component solvers. While this is straightforward for Variant-I solver, we have the following claim for Variant-II solver.

Theorem 6.1
Let Q be a planning problem. Assume that we are using the layered 'make-progress' rule.

- (Variant-II Soundness) If Q has an optimal solution P with cost C and makespan n, then for any $k \leq n$, at makespan k the Variant-II solver will find a relaxed plan with cost $\leq C$.
- (Variant-II Completeness) If Q has no solution, then the Variant-II solver will eventually produce an UNSAT instance.

Intuitively, the completeness is due to non-repetition of states enforced by the 'make-progress' rule, and the soundness follows from the fact that if there exists a plan, then it can be "reduced" to a plan which satisfies the 'make progress' rule. Furthermore, it can be truncated at any lower makespan to a partial plan which satisfies the 'make progress' rule. A more detailed argument (and all proofs of the claims of this paper) can be found in Appendix B.

7 Planning without Layers: Stepless Planning

Besides the delete-free planner above (which is only useful for delete-free planning problems), all the planners so far in this paper (and indeed, all SAT/ASP planners that we have encountered) have used layers to order the actions and fluents that occur within a planning problem. But let us consider the notion of *partially ordered plan* from Definition 3.1, where no layers are specified. Any topological sort of this graph corresponds to a valid plan. Perhaps we could avoid layers entirely and embed action-dependencies directly. The idea here is that, just as with delete-free planning, we can create a plan by specifying only which actions and which fluents hold, and we will rely on stable model semantics to ensure that the resulting solution graph is acyclic.

As was mentioned in Section 5 this is the most novel contribution of this paper. We use stable-model semantics, rather than layers, to produce an acyclic plan.

There is a key difference between delete-free planning and full stepless planning though, which accounts for the distinction in computational complexity. In the case of delete-free planning, no fluent holds more than once and no action occurs more than once. In stepless planning, it is possible for an action to occur multiple times. As such, we will have to have separate atoms in our encoding representing each *occurrence* of an action. But prior to solving, we don't know how many occurrences of each action or fluent will be needed.

Here, we will first present a solver that *assumes* it has enough occurrences and then we will come back to the issue of figuring out how many of each are needed in order to produce an optimal plan. The stepless planner is *significantly* more complicated than anything else done in this paper so we put more care into explaining what each line of ASP code does, but we will have to do it in an appendix (Appendix D). Here we provide

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

an outline of the planner. Additionally, since no planner like this has ever been built before, we will take more care to try and bridge the gap between the standard approach to planning and the approach being presented here.

7.1 Stepless Planner Encoding

To avoid an $O\left(|\text{actions}|^2\right)$-size encoding, we don't directly encode dependencies between actions. Instead we use the fluents as intermediate nodes in the solution graph.

An *occurrence* of a fluent F will be encoded as an object in an atom, $fluentOcc(F, M)$, where M is a sequentially-ordered index. $M = 0$ is reserved for the initial fluents. All others start at $M = 1$ (when caused by some action). Similarly, the object $actOcc(A, N)$ indicates an occurrence of action A.[8] In stepless planning, there are no preserving actions since there are no layers to preserve things across, and we don't utilize mutex relationships between objects. Whereas in our previous encodings the causes and destroyers of each fluent were implicit, here we must explicitly give which occurrence of which action AO causes which fluent occurrence FO to hold ($causes(AO, FO)$) and which occurrence of which fluent FO is used as a precondition for which action occurrence AO ($permits(FO, AO)$). Additionally, we need an atom for each deleted fluent occurrence FO which action occurrence AO has as a precondition and deletes ($deletes(AO, FO)$) and also one in the rare case that an action has a fluent as a delete-effect, but not a precondition, for which occurrence of the fluent F the action occurrence AO follows ($follows(AO, FO)$). (Refer to Appendix D under the subtitle *Problem Description.*)

From this we can structure the graph and assert that it is acyclic. For each action occurrence we have an "event"; additionally there is an event for the start and end of each occurrence of each fluent. There is also an event "goal" which corresponds to the goal state being reached.

Events are grouped into *vertices* in our graph each of which contains at most one action occurrence. When an action occurrence *causes* a fluent, the action and the start of that fluent belong to the same vertex. Similarly when it *deletes* a fluent, the action and the end of the fluent belong to the same vertex. To encode this we use the predicate $inVertex/2$ which indicates that its first argument belongs to the vertex named by the second argument. (Refer to Appendix D under the subtitle *Plan Event Graph.*[9])

7.2 Making Stepless Progress with a Suffix Layer

Now we need a way to assert that the action occurrences of a given stepless plan "make progress". With no layers to make assertions about, the only notion of progress we are left with is the definition (Def. 3.2) of a plan which is *strongly minimal*. This definition

[8] We index action and fluent occurrences with numbers N and M and have symmetry-breaking rules ensuring that the occurrences happen in numerical order for a given action or fluent, but it is important to understand that these numbers are *not* layers. There's no global *step* of any kind to which they correspond. A fluent occurrence can be used as a precondition for an appropriate action occurrence regardless of what their indices are or how they relate to each other. The same goes for an action *causing* a fluent. The indices are simply to be able to distinguish between multiple occurrences of the same object; they have no global significance or relation to any other object.

[9] We have removed the goal event from the encoding in the appendix since it is incompatible with the suffix layer. To see the original encoding refer to (Spies 2019).

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

logically takes the form of "there does not exist a pair of (s,t)-cuts such that ...." This means that *given* a particular plan, determining whether it is strongly minimal is likely co-NP-complete (membership is straightforward but the hardness is an open conjecture), and then the problem of *determining the existence of* such a plan for a given collection of atoms and fluents could possibly be Σ_2^P-complete. Luckily, ASP gives us a way to encode problems in Σ_2^P through the use of *disjunctive rules* (Baral 2003). The code for this can be found in Appendix D under the subtitle *Strong Minimality.*

If there aren't enough occurrences of a fluent or action, we can tack on a suffix layer in the same way we did with the stepped-cost-optimal planner. In the code in Appendix D **??** under the subtitle *Suffix Layer*, we replace all uses of *goal* with a *subgoal* which is the entry-point into the suffix layer. The coding is similar to the suffix layer used in the two-threaded planner, but there are a few key differences. First, if the suffix layer is used at all, we use an atom *useSuffix* to indicate that this is true. There is a cost of 1 at level -1 for *useSuffix* so among plans of equal cost, the solver will prefer one which *doesn't* use the suffix to one which does. If an optimal solution doesn't use the suffix, then it must be globally optimal with respect to cost. (Refer to Appendix D under subtitle *Suffix Layer.*)

Finally, we will add rules to enforce the use of action and fluent occurrences from our bag so that the planner resorts to the suffix layer only when it "runs out" of something. With this we know how to expand our bag of occurrences. Each time we get back a plan making use of the suffix layer, look at all the fluents or actions which were saturated by that plan and add another occurrence of each one. (Refer to Appendix D under subtitle *Saturated.*)

This, coupled with our definition of making progress, is what guarantees that it will eventually find a plan or determine that none exists. The suffix layer is only used because the planner ran out of something it needed and needs to request more of that item from the controlling program (in particular, not as a way to save on plan cost).

Appendix E provides a detailed example of running the stepless planner, and Appendix B gives a proof of the following theorem.

Theorem 7.1

- (Stepless Soundness) All plan costs produced by the stepless planner are lower bounds on the cost of the true optimal plan.
- (Stepless Completeness) The stepless planner will eventually find the solution if it exists or produce an UNSAT instance if it doesn't.

8 Experiments

We ran our cost-optimal two-threaded solver and stepless solver on most of the same instances as Robinson (Robinson et al. 2010) and here report results[10].

Experiments were run on a cluster of *c*3.*large* Amazon EC2 instances each with two Intel Xeon 2.8 GHz CPU cores and 3.75 GB of memory. We used GNU Parallel (Tange 2018) to distribute the work of running multiple instances.

[10] excluding *satellites* since our planner doesn't support the : *equality* extension to PDDL and *miconic* since we couldn't find the problem files for it. Robinson was kind enough to send us the instances from his constructed *ftb* domain so we can report performance on that as well

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

Table 1. *Experiments with two-threaded planner and stepless planner*

Problem	C^*	n	n_*	t_π	t_*	$t_{2\text{-}threaded}$	n_s	$t_{stepless}$	l_s	$t_{Robinson}$
block-12	20	20	17	0.5	1203.4	1203.9	-	-	-	?
block-15	16	16	12	0.4	113.4	113.8	7	89.4	33.7	?
block-18	26	26	16	0.9	256.8	257.7	-	-	-	3.2
block-23	30	-	-	-	-	-	-	-	-	29.8
block-25	34	-	-	-	-	-	-	-	-	27.4
depots-2	15	-	12	-	771.5	771.5	2	9.7	4.2	-
depots-13	25	-	-	-	-	-	3	475.9	137.2	-
driverlog-2	19	-	-	-	-	-	20	215.5	44.4	-
driverlog-3	12	7	3	0.1	0.9	1.0	1	0.4	0.4	450.2
driverlog-11	19	-	-	-	-	-	1	13.5	13.5	-
elevators-2	26	3	0	0.4	1.7	2.1	1	2.7	2.7	13.0
freecell-3	18	-	-	-	-	-	2	420.5	344.0	-
ftb-30	1001	25	0	1.8	0.3	2.1	1	5.5	5.5	1.8
ftb-38	601	33	0	2.7	0.2	2.9	1	3.2	3.2	1.5
ftb-39	801	33	0	3.9	0.3	4.2	1	5.6	5.6	2.2
ftb-40	1001	33	0	3.9	0.4	4.3	1	8.2	8.2	?
gripper-1	11	7	4	0.1	0.4	0.5	2	0.4	0.2	14.6
gripper-2	17	11	8	0.6	312.4	313.0	7	23.5	9.7	-
pegsol-9	5	15	11	3.9	35.9	39.8	5	131.5	46.6	386.8
pegsol-16	8	21	17	48.3	1029.0	1509.3	10	910.2	280.8	-
pegsol-18	7	-	-	-	-	-	7	1548.0	537.1	-
rovers-3	11	7	4	0.1	0.2	0.3	1	0.1	0.1	49.4
rovers-4	8	4	0	0.0	0.0	0.0	1	0.1	0.1	?
rovers-6	36	-	-	-	-	-	48	1354.3	391.0	-
rovers-9	31	-	-	-	-	-	53	1040.6	101.4	-
rovers-14	28	-	-	-	-	-	72	900.7	55.9	-
storage-7	14	14	11	0.6	42.9	43.5	10	89.2	42.4	1.1
storage-8	13	-	-	-	-	-	15	799.1	239.5	-
storage-9	11	-	-	-	-	-	9	181.0	46.0	-
storage-13	18	-	-	-	-	-	-	-	-	244.0
TPP-5	19	7	2	0.1	0.2	0.3	2	0.5	0.3	-
TPP-7	34	-	-	-	-	-	13	189.6	32.4	-
transport-1	54	5	0	0.1	0.1	0.2	2	0.5	0.3	0.2
transport-2	131	12	4	74.3	55.1	129.4	2	111.6	106.3	-
transport-11	456	9	3	0.3	1.6	1.9	2	163.4	151.4	-
transport-21	478	7	1	0.2	0.6	0.8	2	5.2	3.5	-
zenotravel-4	8	7	3	0.5	2.8	3.3	3	14.1	6.9	783.4
zenotravel-6	11	7	0	7.2	6.5	13.7	1	2.1	2.1	-
zenotravel-10	22	-	-	-	-	-	1	1387.1	1387.1	-

For comparison, we include Robinson's reported results scaled down by a factor of $\frac{2.6}{2.8}$ to account for the difference in processor speeds.

For each domain, we report the largest instance solved by each of the two-threaded planner, the stepless planner, and Robinson's planner where largest is measured by the amount of time it took that planner to solve the instance. Where it differs, we also report the largest-indexed instance solved by each of the two-thread and stepless planners.

Every plan produced by either planner was validated by the Strathclyde Planning Group plan verifier VAL (Howey et al. 2004).

The column C^* is the optimal cost found for each instance. In all cases the optimal cost for the two-thread planner agrees with the optimal cost reported by Robinson (Robinson et al. 2010) where applicable (Robinson compares his results against a *non*-SAT-based planner and our optimal costs agrees with that as well).

The column n is the lowest makespan at which the problem has a C^* plan (according to our Variant-I solver). Our value for the makespan n agreed with all of Robinson's reported results except for Rovers-3 where we found we only needed a makespan of 7 to

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

produce the optimal plan while Robinson reported a required makespan of 8. We suspect this is because the definition of mutex of (Blum and Furst 1997) is overly restrictive for actions (cf. the footnote on page 3).

n_* is the makespan at which our Variant-II suffix solver proves C^* is optimal. Interestingly, for many of instances this value was 0 which indicates that the optimal plan in the delete-free reduction of the problem has the same cost as the true optimal plan.

t_π is the time required to find the plan (by our Variant-I solver); t_* is the time required to prove optimality (by our Variant-II suffix solver); and $t_{2\text{-}threaded}$ is the sum of these two numbers (can be thought of as "total solve time" although the algorithm necessitates that they run in parallel, so the actual wall-clock time required to run them was the maximum, not the sum, but with two CPU cores rather than one). All reported times are measured in seconds.

n_s is the number of times the stepless solver was run for this instance (each time adding more items to its bag of fluents and actions based on what was saturated in the previous rounds). On the last of these runs it produced an optimal solution which doesn't use the suffix layer and hence is globally cost-optimal. t_s is the total time running the stepless solver across all runs.

One important distinction between the experiments run with the two-threaded solver and those run with the stepless solver is that the two-threaded solver took advantage of "iterative" solving. That is, CLINGO provides an API for interacting with it programatically. Rather than restarting from scratch each time there is a new instance to be solved, we can after observing the solution at makespan k, make some adjustments to the instance so that it now represents the program for makespan $k+1$ and then ask CLINGO to *continue* solving from this point while maintaining any learnt clauses which are still relevant.

This is incredibly powerful and resulted in a major speedup in the two-threaded solver.

For the stepless solver this was not possible since it relies heavily on full-program-spanning loop constraints to give correct results, but CLINGO doesn't support having loop constraints cross multiple iterative stages. Thus, every time new fluents or actions are added to the stepless solver's bag, it starts solving from scratch. In the future we hope CLINGO can support this, but they have no plans to do so at this time.

l_s is the total time required for the last iteration of the stepless solver to run. This one run by itself is sufficient to both find the globally optimal solution *and* prove its optimality. However we know of no more efficient way to find the right bag of actions and fluents in order to guarantee the optimal solution won't use the suffix layer. This number is still interesting in that it provides a lower bound on the time it would take to solve the instance if CLINGO supported loop constraints crossing program section boundaries (so that we could add more occurrences and continue solving rather than having to restart). It gives us some idea of what savings such a modification to CLINGO might provide.

$t_{Robinson}$ is the total time reported by (Robinson et al. 2010) to find the optimal solution scaled by a factor of $\frac{2.6}{2.8}$. A question mark ? in this column indicates the time is unknown since it's not reported in (Robinson et al. 2010). If the solver for which this row is maximal successfully solved the largest instance reported by Robinson in this domain and found this instance to be larger, we fill with a dash mark − rather than ? in this column (our best guess as to whether Robinson's planner solved it). A − in

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

any other column indicates the relevant planner failed to solve the instance in less than 1671.4 seconds (30 minutes scaled down by $\frac{2.6}{2.8}$). In the case of depots-2, the Variant-II suffix solver reached layer 12 before the Variant-I MinASPPlan solver and so it found an optimal no-suffix solution by itself.

All instances were solved with CLINGO version 5.2.3. The controller logic for both the two-threaded solver (handling of incremental solving, coordinating the two solvers, and figuring out when to terminate the search) and the stepless solver (figuring out which occurrences to include and topologically sorting the output) was written in Haskell using the clingo-Haskell bindings written by tsahyt (GitHub alias) to communicate with CLINGO .

We used the default configuration and options for CLINGO except that the stepless planner used the –opt-usc option which finds optimal solutions by expanding an unsatisfiable core (Definition 2 in (Alviano et al. 2015)).

All planners presented here are available on GitHub at `https://github.com/davidspies/aspplan2`. Feel free to contact the repo owner (the first author of this paper) for any help with reproducing these results.

In all domains except for *blocks*, *ftb*, and *storage*, our two-threaded solver outperformed Robinson's SAT-based solver and our stepless solver outperformed both (in terms of number of instances solved). In the case of *storage*, the stepless solver and Robinson's solver each solved an instance which the other failed to solve which seems to point to the possibility that the stepless solver encounters different difficulties from a more traditional approach. One more piece of evidence favoring this conclusion is that the toy example bridge-crossing problem from **??** required a full 30 seconds to solve (whereas the 2-threaded solver solves it in 2 seconds) and in general we found that on small/toy problems the stepless solver's performance is abysmal compared with other approaches we tried but scales better with larger instances.

Prior to running the full suite of experiments, the above observation gave us the mistaken impression that the stepless solver was interesting as a theoretical oddity, but fails to produce decent results in practice, since for every example we ran it on while tuning it, it seemed to run slower than the two-threaded solver. It was a pleasant surprise to discover when officially running the experiments that in fact the inverse was true.

9 Related Work and Final Remarks

Our 2-threaded solver algorithm is inspired by the approach of (Robinson et al. 2010). The better performance of our planner, besides solver technologies, seems partly due to the grounding size and search space pruning under stable model semantics as commented in Section 2, and partly due to less clustered encoding in ASP than in SAT, plus the smart encoding of mutex constraints. But note that their approach works only if we assume the problem is solvable (which he does) and all actions have positive (nonzero) costs (which he also does).

Eiter et al. (2003) propose an approach to finding globally cost-optimal plans in ASP with action costs, but confine their discussion to the planning problems which look for *polynomial length plans*. They make the assumption that the planning domain has some polynomial upper bound on plan lengths which is known in advance. In contrast, we do not make such an assumption. It is interesting to note that through some key technical

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

innovations, we are able to show that the current ASP techniques are capable of encoding cost-optimal planners without this assumption.

Partially-ordered plans have been explored elsewhere. One is the CPT planner (Vidal and Geffner 2006) for optimal temporal planning using Constraint Programming (CP), where actions have *durations* and makespan refers to total duration which corresponds to the cost of a plan in our setting. Optimality here means minimal duration. CPT consists of pre-processing that induces lower bounds to be used in starting makespan and in formulation of constraints, a branching scheme, and a CP-based branch and bound search. The branching scheme is specifically designed for temporal planning aiming for increased reasoning efficiency. The current makespan B increases by 1 if no plan is found. Thus, the first plan found is guaranteed to be duration minimal. CPT dynamically generates *action tokens* from *action types* achieving the similar goal of the suffix layer of our stepless planner. Thus, a main difference between CPT and our stepless planner is no-solution detection in our case and the lack of it in CPT, i.e., the current CPT does not terminate on its own for UNSAT instances.

Delete-free planning has been investigated as a stand-alone topic, including a CP solution (Barták et al. 2012). To the best of our knowledge, our modeling of delete-free planning as a graph problem is original, and it leads to a five-line ASP program which does everything (cf. Appendix C).

The *Madagascar Planner* is a family of efficient implementations of the SAT based techniques for planning. The main idea is, instead of using the standard decision heuristics such as VSIDS, planning-specific variable selection heuristics are applied (Rintanen 2012). One would expect that the same idea can work for ASP-based planning, and in this case, our 2-threaded cost-optimal planner can benefit from it directly.

Cost-optimal planners can also be built on the SAS^+ platform. An SAS^+ planner based on greedy selection of a subset of heuristics for guiding A^* search (Lelis et al. 2016) has made to the top tier in IPC-2018. SAS^+ planning can be encoded in SAT and ASP as well, but the most critical component, the selection algorithm, needs to be implemented by an external program.

Though the goal of this paper is limited to ASP-based cost-optimal planning, there is always a question of whether such a planner is competitive in efficiency (in terms of solving time) with state-of-the-art planners, e.g., the top planners from IPC. Further investigation and experimentation are needed to address such questions. A major advantage of ASP-based planners is the succinctness and elegance of the encoding. An expressive KR language like ASP provides some unique advantages, e.g., determining the existence of a plan that satisfies the stepless make-progress rule is likely Σ_2^P-hard, which can be a challenge for other KR languages.

Stepless planning is a brand new approach to logic-based planning and brings with it a lot of unknowns and potentials for future directions. One issue is that the lack of any notion of simultaneity makes certain standard optimizations difficult, such as incorporating mutex constraints and supporting *conditional-effects* (an extension to STRIPS planning). The latter extension has been realized in SAT-based planning (Rintanen 2011), but incorporating it to stepless planning appears to be non-trivial. Our stepless planner is a nontrivial application that requires supportedness cycles to extend across different program sections, it would be nice if CLINGO supported iterative solving with this.

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

More recently, no-solution detection for planners has become an interesting topic, along with the competition called Unsolvability IPC, which aims to test classical automated planners to detect when a planning task has no solution. Our no-solution techniques presented in this paper may be relevant. This is one interesting future direction.

Finally, *property directed reachability* (PDR), a promising method for deciding reachability in symbolically represented transition systems, which was originally conceived as a model checking algorithm for hardware circuits, has recently been related to planning (Suda 2014). The relationship with our stepless planner deserves a further study; in particular, an interesting question is whether and how PDR-based planners can be strengthened to become cost-optimal.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068419000395.

References

Alviano, M., Dodaro, C., Marques-Silva, J., and Ricca, F. 2015. Optimum stable model search: algorithms and implementation. *Journal of Logic and Computation*.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, New York, NY.

Barták, R., Dvorak, F., Gemrot, J., Brom, C., and Toropila, D. 2012. When planning should be easy: On solving cumulative planning problems. In *Proc. 25th International Florida Artificial Intelligence Conference*, Florida, USA.

Blum, A. L. and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence 90,* 1, 281–300.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., and Schaub, T. 2015. ASP-Core-2 input language format. https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf. ASP Standardization Working Group.

Chen, Y., Lv, Q., and Huang, R. 2008. Plan-A: A cost-optimal planner based on SAT-constrained optimization. *Proc. 6th International Planning Competition (IPC-08)*.

Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J., and Schaub, T. 2019. plasp 3: Towards effective ASP planning. *Theory and Practice of Logic Programming 19,* 3, 477–504.

Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. 2003. Answer set planning under action costs. *Journal of Artificial Intelligence Research 19*, 25–71.

Fikes, R. and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2,* 3/4, 189–208.

Gebser, M., Kaminski, R., Knecht, M., and Schaub, T. 2011. plasp: A prototype for PDDL-based planning in ASP. In *Proc. LPNMR-11*, pp. 358–363. Vancouver, Canada.

Howey, R., Long, D., and Fox, M. 2004. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proc. 16th IEEE International Conference on Tools with Artificial Intelligence*, Boca Raton, Florida, USA, pp. 294–301.

Kautz, H. 2004. Satplan04: Planning as satisfiability. *Working Notes on the Fourth International Planning Competition (IPC-04)*, 44–45.

Lelis, L. H. S., Franco, S., Abisrror, M., Barley, M., Zilles, S., and Holte, R. C. 2016. Heuristic subset selection in classical planning. In *Proc. IJCAI-16*, New York, USA, pp. 3185–3191.

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138,* 1-2, 39–54.

Maratea, M. 2012. Planning as satisfiability with IPC simple preferences and action costs. *AI Communications 25,* 4, 343–360.

Rintanen, J. 2011. Heuristics for planning with SAT and expressive action definitions. In *Proc. 21st International Conference on Automated Planning and Scheduling*, Freiburg, Germany.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *Artificial Intelligence 193*, 45–86.

Robinson, N., Gretton, C., Pham, D. N., and Sattar, A. 2008. A compact and efficient SAT encoding for planning. In *Proc. 18th International Conference on Automated Planning and Scheduling*, Sydney, Australia, pp. 296–303.

Robinson, N., Gretton, C., Pham, D.-N., and Sattar, A. 2010. Cost-optimal planning using weighted MaxSAT. In *Proc. the ICAPS'10 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, Toronto, Canada.

Spies, D. 2019. Domain-independent cost-optimal planning in ASP. MSc. Thesis, University of Alberta, Edmonton, Canada.

Suda, M. 2014. Property directed reachability for automated planning. *Journal of Artificial Intelligence Research 50*, 265–319.

Tange, O. 2018. *GNU Parallel 2018*. Ole Tange.

Vidal, V. and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence 170,* 3, 298–335.

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press

https://doi.org/10.1017/S1471068419000395 Published online by Cambridge University Press