Eliminating All Redundant Actions from Plans Using SAT and MaxSAT

Tomáš Balyo
Department of Theoretical Computer Science
and Mathematical Logic,
Faculty of Mathematics and Physics
Charles University in Prague
biotomas@gmail.com

Lukáš Chrpa
PARK Research Group
School of Computing and Engineering
University of Huddersfield
l.chrpa@hud.ac.uk

Abstract

Satisfiability (SAT) techniques are often successfully used for solving planning problems. In this paper we show, that SAT and maximum satisfiability (MaxSAT) can be also used for post-processing optimization of plans. We will restrict ourselves to improving plans by removing redundant actions from them which is a special case of plans optimization. There exist polynomial algorithms for removing redundant actions, but none of them can remove all such actions since guaranteeing that a plan does not contain redundant actions is NP-complete. We introduce two new algorithms, based on SAT and MaxSAT, which remove all redundant actions. The MaxSAT based algorithm additionally guarantees to remove a maximum set of redundant actions. We test the described algorithms on plans obtained by state-of-the-art planners on IPC 2011 benchmarks. The proposed algorithms are very fast for these plans despite the complexity results.

Introduction

Automated Planning is an important research area for its good application potential (Ghallab, Nau, and Traverso 2004). With intelligent systems becoming ubiquitous there is a need for planning systems to operate in almost real-time. Sometimes it is necessary to provide a solution in a very little time to avoid imminent danger (e.g damaging a robot) and prevent significant financial losses. Satisficing planning engines such as FF (Hoffmann and Nebel 2001), Fast Downward (Helmert 2006) or LPG (Gerevini, Saetti, and Serina 2003) are often able to solve a given problem quickly, however, quality of solutions might be low. Optimal planning engines, which guarantee the best quality solutions, often struggle even on simple problems. Therefore, a reasonable way how to improve the quality of the solutions produced by satisficing planning engines is to use post-planning optimization techniques.

In this paper we restrict ourselves to optimizing plans by only removing redundant actions from them. Guaranteeing that a plan does not contain redundant actions is NP-complete (Fink and Yang 1992). There are polynomial algorithms, which remove most of the redundant actions, but none of them removes all such actions. We propose two new algorithms which are guaranteed to remove them all. One uses satisfiability (SAT) solving, the other one relies on maximum satisfiability (MaxSAT) solving. We compare our algorithms with a heuristic algorithm on plans obtained by state-of-the-art planners on IPC 2011 benchmarks.

Related Work

Various techniques have been proposed for post-planning plan optimization. Westerberg and Levine (2001) proposed a technique based on Genetic Programming, however, it is not clear whether it is required to hand code optimization policies for each domain as well as how much runtime is needed for such a technique. Planning Neighborhood Graph Search (Nakhost and Müller 2010) is a technique which expands a limited number of nodes around each state along the plan and then by applying Dijsktra‘s algorithm finds a better quality (shorter) plan. This technique is anytime since we can iteratively increase the limit for expanded nodes in order to find plans of better quality. AIRS (Estrem and Krebsbach 2012) improves quality of plans by identifying suboptimal subsequences of actions according to heuristic estimation (a distance between given pairs of states). If the heuristic indicates that states might be closer than they are, then a more expensive (optimal) planning technique is used to find a better sequence of actions connecting the given states. A similar approach exists for optimizing parallel plans (Balyo, Barták, and Surynek 2012). A recent technique (Siddiqui and Haslum 2013) uses plan deordering into ‘blocks’ of partially ordered subplans which are then optimized. This approach is efficient since it is able to optimize subplans where actions might be placed far from each other in a totaly ordered plan.

Determining and removing redundant actions from plans is a specific sub-category of post-planning plan optimization. An influential work (Fink and Yang 1992) defines four categories of redundant actions and provides complexity results for each of the categories. One of the categories refers to Greedily justified actions. A greedily justified action in the plan is, informally said, such an action which if it and actions dependent on it are removed from the plan, the plan becomes invalid. Greedy justification is used in the Action Elimination (AE) algorithm (Nakhost and Müller 2010) which is discussed in detail later in the text. Another of the categories refers to Perfectly Justified plans, plans in which no redundant actions can be found. Minimal reduction of plans (Nakhost and Müller 2010) is a special case of Perfectly Justified plans having minimal cost of the plan. Both Perfect Justification and Minimal reduction are NP-complete. Determining redundant pairs of inverse actions (inverse actions are those that revert each other's effects), which aims to eliminate the most common type of redundant actions in plans, has been also recently studied (Chrpa, McCluskey, and Osborne 2012a; 2012b).

Preliminaries

In this section we give the basic definitions and properties used in the rest of the paper.

Satisfiability

A *Boolean variable* is a variable with two possible values *True* and *False*. A *literal* of a Boolean variable x is either x or $\neg x$ (*positive* or *negative literal*). A *clause* is a disjunction (OR) of literals. A clause with only one literal is called a *unit clause* and with two literals a *binary clause*. An implication of the form $x \Rightarrow (y_1 \vee \cdots \vee y_k)$ is equivalent to the clause $(\neg x \vee y_1 \vee \cdots \vee y_k)$. A *conjunctive normal form (*CNF*) formula* is a conjunction (AND) of clauses. A truth assignment ϕ of a formula F assigns a truth value to its variables. The assignment ϕ satisfies a positive (negative) literal if it assigns the value True (False) to its variable and ϕ satisfies a clause if it satisfies any of its literals. Finally, ϕ satisfies a CNF formula if it satisfies all of its clauses. A formula F is said to be satisfiable if there is a truth assignment ϕ that satisfies F. Such an assignment is called a *satisfying assignment*. The satisfiability problem (SAT) is to find a satisfying assignment of a given CNF formula or determine that it is unsatisfiable.

Partial Maximum Satisfiability

A *partial maximum satisfiability (PMaxSAT) formula* is a CNF formula consisting of two kinds of clauses called *hard* and *soft* clauses. A PMaxSAT formula is satisfied under a truth assignment ϕ if it satisfies all of its hard clauses.

The *partial maximum satisfiability problem* (PMaxSAT) is to find a satisfying assignment ϕ for a given PMaxSAT formula such that ϕ satisfies as many soft clauses as possible.

Planning

In this section we give the formal definitions related to planning. We will use the multivalued SAS+ formalism (Bäckström and Nebel 1995) instead of the classical STRIPS formalism (Fikes and Nilsson 1971) based on propositional logic.

A planning task Π in the SAS+ formalism is defined as a tuple $\Pi = \{X, O, s_I, s_G\}$ where

- $X = \{x_1, \ldots, x_n\}$ is a set of multivalued variables with finite domains $\mathrm{dom}(x_i)$.
- O is a set of actions (or operators). Each action $a \in O$ is a tuple $(\mathrm{pre}(a), \mathrm{eff}(a))$ where $\mathrm{pre}(a)$ is the set of preconditions of a and $\mathrm{eff}(a)$ is the set of effects of a. Both preconditions and effects are of the form $x_i = v$ where $v \in \mathrm{dom}(x_i)$.
- A state is a set of assignments to the state variables. Each state variable has exactly one value assigned from its respective domain. We denote by S the set of all states. $s_I \in S$ is the initial state. s_G is a partial assignment of the state variables (not all variables have assigned values) and a state $s \in S$ is a goal state if $s_G \subseteq s$.

An action a is *applicable* in the given state s if $\mathrm{pre}(a) \subseteq s$. By $s' = \mathrm{apply}(a, s)$ we denote the state after executing the action a in the state s, where a is applicable in s. All the assignments in s' are the same as in s except for the assignments in $\mathrm{eff}(a)$ which replace the corresponding (same variable) assignments in s.

A *(sequential) plan* P of *length* k for a given planning task Π is a sequence of actions $P = \{a_1, \ldots, a_k\}$ such that $s_G \subseteq \mathrm{apply}(a_k, \mathrm{apply}(a_{k-1} \ldots \mathrm{apply}(a_2, \mathrm{apply}(a_1, s_I)) \ldots))$. We will denote by $|P|$ the length of the plan P.

Redundant Plans

A plan P for a planning task Π is called *redundant* if there is a subsequence P' of P ($|P'| < |P|$), such that P' is a valid plan for Π. The actions in P that are not present in P' are called *redundant actions*. A plan which is not redundant is called a *perfectly justified plan*.

A plan P for a planning task Π is called an *optimal plan* if there is no other plan P' for Π such that $|P'| < |P|$. Note, that a perfectly justified plan is not necessarily an optimal plan. On the other hand, an optimal plan is always perfectly justified.

Determining whether a plan is perfectly justified is NP-complete (Fink and Yang 1992). Nevertheless, there are several heuristic approaches, which can identify most of the redundant actions in plans in polynomial time. One of the most efficient of these approaches was introduced in (Fink and Yang 1992) under the name Linear Greedy Justification. It was reinvented in (Nakhost and Müller 2010) and called Action Elimination. In this paper we use the latter name.

Action Elimination (see Figure 1) tests for each action if it is greedily justified. An action is greedily justified if removing it and all the following actions that depend on it makes the plan invalid. One such test runs in $O(np)$ time, where $n = |P|$ and p is the maximum number of preconditions and effects any action has. Every action in the plan is tested, therefore Action Elimination runs in $O(n^2p)$ time.

There are plans, where Action Elimination cannot eliminate all redundant actions (Nakhost and Müller 2010). An interesting question is how often this occurs for the planning domains used in the planning competitions (Coles et al. 2012). To find out, first we need to design an algorithm that always eliminates all redundant actions, i.e., find perfectly justified plans. As mentioned earlier, this problem is NP-complete and therefore we find it reasonable to solve it using a SAT reduction approach. In the next section we will introduce a translation of this problem into SAT.

Satisfiability Encoding of Plan Redundancy

This section is devoted to introducing an algorithm, which given a planning task Π and a valid plan P for Π, outputs a

```
ActionElimination ($\Pi$, $P$)
s := s_I
i := 1
repeat
   mark($a_i$)
   $s' := s$
   for $j := i + 1$ to $|P|$ do
      if applicable($a_j$, $s'$) then
         $s'$ := apply($a_j$, $s'$)
      else
         mark($a_j$)
   if goalSatisfied($s'$) then
      $P$ := removeMarked($P$)
   else
      unmarkAllActions()
      $s$ := apply($a_i$, $s$)
   $i := i + 1$
until $i > |P|$
return $P$
```

Figure 1: Pseudo-code of the Action Elimination algorithm as presented in (Nakhost and Müller 2010).

CNF formula $F_{\Pi,P}$, such that $F_{\Pi,P}$ is satisfiable if and only if P is a redundant plan for Π.

We provide several definitions which are required to understand the concept of our approach. An action a is called a *supporting action* for a condition c if $c \in \text{eff}(a)$. An action a is an *opposing action* of a condition $c := x_i = v$ if $x_i = v' \in \text{eff}(a)$ where $v \neq v'$. The *rank* of an action a in a plan P is its order in the sequence P. We will denote by $Opps(c, i, j)$ the set of ranks of opposing actions of the condition c which have their rank between i and j ($i \leq j$). Similarly, by $Supps(c, i)$ we will mean the set of ranks of supporting actions of the condition c which have ranks smaller than i.

In our encoding we will have two kinds of variables. First, we will have one variable for each action in the plan P, which will represent whether the action is required for the plan. We will say that $a_i = True$ if the i-th action of P (the action with the rank i) is required. The second kind of variables will be option variables, their purpose and meaning is described below.

The main idea of the translation is to encode the fact, that if a certain condition c_i is required to be true at some time i in the plan, then one of the following must hold:

- The condition c_i is true since the initial state and there is no opposing action of c_i with a rank smaller than i.
- There is a supporting action a_j of c_i with the rank j and there is no opposing action of c_i with the rank between j and i.

These two kinds of properties represent the options for satisfying c_i. There is at most one option of the first kind and at most $|P|$ of the second kind. For each one of them we will use a new option variable $y_{c,i,k}$, which will be true if the condition c at time i is satisfied using the k-th option.

Now we demonstrate how to encode the fact, that we require condition c to hold at time i. If c is in the initial state, then the first option will be expressed using the following conjunction of clauses.

$$F_{c,i,0} = \bigwedge_{j \in Opps(c,0,i)} (\neg y_{c,i,0} \vee \neg a_j)$$

These clauses are equivalent to the implications below. The implications represent that if the given option is true, then none of the opposing actions can be true.

$$(y_{c,i,0} \Rightarrow \neg a_j); \forall j \in Opps(c, 0, i)$$

For each supporting action a_j ($j \in Supps(c, i)$) with rank j we will introduce an option variable $y_{c,i,j}$ and add the following subformula.

$$F_{c,i,j} = (\neg y_{c,i,j} \vee a_j) \bigwedge_{k \in Opps(c,j,i)} (\neg y_{c,i,j} \vee \neg a_k)$$

These clauses are equivalent to the implications that if the given option is true, then the given supporting action is true and all the opposing actions located between them are false. Finally, for the condition c to hold at time i we need to add the following clause, which enforces at least one option variable to be true.

$$F_{c,i} = (y_{c,i,0} \bigvee_{j \in Supps(c,i)} y_{c,i,j})$$

Using the encoding of the condition requirement it is now easy to encode the dependencies of the actions from the input plan and the goal conditions of the problem. For an action a_i with the rank i we will require that if this action variable is true, then all of its preconditions must be true at time i. For an action a_i the following clauses will enforce, that if the action variable is true, then all the preconditions must hold.

$$F_{a_i} = \bigwedge_{c \in \text{pre}(a_i)} \left((\neg a_i \vee F_{c,i}) \wedge F_{c,i,0} \bigwedge_{j \in Supps(c,i)} F_{c,i,j} \right)$$

We will need to add these clauses for each action in the plan. Let us call these clauses F_A.

$$F_A = \bigwedge_{a_i \in P} F_{a_i}$$

For the goal we will just require all the goal conditions to be true in the end of the plan. Let $n = |P|$, then the goal conditions are encoded using the following clauses.

$$F_G = \bigwedge_{c \in s_G} \left(F_{c,n} \wedge F_{c,n,0} \bigwedge_{j \in Supps(c,n)} F_{c,n,j} \right)$$

The last clause we need to add is related to the redundancy property of the plan. The following clause is satisfied if at least one of the actions in the plan is omitted.

$$F_R = \left(\bigvee_{a_i \in P} \neg a_i \right)$$

Finally, the whole formula $F_{\Pi,P}$ consists of the redundancy clause, the goal clauses, and the action dependency clauses for each action in P.

$$F_{\Pi,P} = F_R \wedge F_G \wedge F_A$$

If the formula is satisfiable, we also want to use its satisfying assignment to construct a new reduced plan. A plan obtained using a truth assignment ϕ will be denoted as P_ϕ. We define P_ϕ to be a subsequence of P such that the i-th action of P is present in P_ϕ if and only if $\phi(a_i) = True$.

Lemma 1. *An assignment ϕ satisfies $F_G \wedge F_A$ if and only if P_ϕ is a valid plan for Π.*

Proof. (sketch) A plan is valid if all the actions in it are applicable when they should be applied and the goal conditions are satisfied in the end. We constructed the clauses of F_G to enforce that at least one option of satisfying each condition will be true. The selected option will then force the required action and none of its opposing actions to be in the plan. Using the same principles, the clauses in F_A guarantee that if an action is present in the plan, then all its preconditions will hold when the action is applied. □

Proposition 1. *The formula $F_{\Pi,P}$ is satisfiable if and only if P is a redundant plan for Π.*

Proof. The clause F_R is satisfied by an assignment ϕ if and only if at least one a_i is false, i.e., not present in P_ϕ which implies $|P_\phi| < |P|$. Using the previous lemma, we can conclude, that the entire formula $F_{\Pi,P} = F_R \wedge F_G \wedge F_A$ is satisfied of and only if there is a valid plan, which can be obtained from P by omitting at least one of its actions. □

Let us conclude this section by computing the following upper bound on the size of the formula $F_{\Pi,P}$.

Proposition 2. *Let p be the maximum number of preconditions of any action in P ,g the number of goal conditions of Π, and $n = |P|$. Then the formula $F_{\Pi,P}$ has at most $n^2p + ng + n$ variables and $n^3p + n^2g + np + g + 1$ clauses, from which $n^3p + n^2g$ are binary clauses.*

Proof. The are n action variables. For each required condition we have at most n option variables, since there are at most n supporting actions for any condition in the plan. We will require at most $(g + np)$ conditions for the g goal conditions and the n actions with at most p preconditions each. Therefore the total number of option variables is $n(np + g)$.

For the encoding of each condition at any time we use at most n options. Each of these options are encoded using n binary clauses (the are at most n opposing actions for any condition). Additionally we have one long clause saying that at least one of the options must be true. We have np required conditions because of the actions and g for the goal conditions. Therefore in total we have at most $(np + g)n^2$ binary clauses and $(np + g)$ longer clauses related to conditions. There is one additional long clause – the redundancy clause. □

Making Plans Perfectly Justified

In this section we describe how to use the encoding described in the previous section to convert any given plan into a perfectly justified plan.

The idea is very similar to the standard planning as SAT approach (Kautz and Selman 1992), where we repeatedly construct formulas and call a SAT solver until we find a plan. In this case we start with a plan, and keep improving it by SAT calls until it is perfectly justified.

```
RedundancyElimination ($\Pi$, $P$)
    $F_{\Pi,P}$ := encodeRedundancy($\Pi$, $P$)
    while isSatisfiable($F_{\Pi,P}$) do
        $\phi$ := getSatAssignment($F_{\Pi,P}$)
        $P := P_\phi$
        $F_{\Pi,P}$ := encodeRedundancy($\Pi$, $P$)
    return $P$
```

Figure 2: Pseudo-code of the SAT based redundancy elimination algorithm. It returns a perfectly justified plan.

The algorithm's pseudo-code is presented in Figure 2. It uses a SAT solver to determine whether a plan is perfectly justified or it can be improved. It can be improved if the formula $F_{\Pi,P}$ is satisfiable. In this case a new plan is constructed using the satisfying assignment. The while loop of the algorithm runs at most $|P|$ times, since every time at least one action is removed from P (in practice several actions are removed in each step).

The algorithm can be implemented in a more efficient manner if we have access to an incremental SAT solver. We need the simplest kind of incrementality – adding clauses.

```
IncrementalRedundancyElimination ($\Pi$, $P$)
    solver = new SatSolver
    solver.addClauses(encodeRedundancy($\Pi$, $P$))
    while solver.isSatisfiable() do
        $\phi$ := solver.getSatAssignment()
        $C := \bigvee\{\neg a_i | a_i \in P_\phi\}$
        solver.addClause($C$)
        foreach $a_i \in P$ do if $\phi(a_i) = False$ then
            solver.addClause($\{\neg a_i\}$)
        $P := P_\phi$
    return $P$
```

Figure 3: Pseudo-code of the incremental SAT based redundancy elimination algorithm.

The incremental algorithm is presented in Figure 3. It adds a new clause C in each iteration of the while loop. This clause is a redundancy clause for the actions remaining in the current plan. It will enforce, that the next satisfying assignment will remove at least one further action. The redundancy clauses added in the previous iterations could be removed, but it is not necessary. The algorithm also adds unit clauses to enforce that the already eliminated actions cannot be reintroduced.

The algorithms presented in this section are guaranteed to produce plans that are perfectly justified, i.e., it is not possible to remove any further actions from them. Nevertheless, it might be the case, that if we had removed a different set of redundant actions from the initial plan, we could have arrived at a shorter perfectly justified plan. In other words, the elimination of redundancy is not confluent. The following example demonstrates this fact.

Example 1. *Let us have a simple path planning scenario on a graph with n vertices $v_1, \ldots, v_n$ and edges (v_i, v_{i+1}) for each $i < n$ and (v_n, v_1) to close the circle. We have one agent traveling on the graph from v_1 to v_n. We have two move actions for each edge (for both directions), in total $2n$ move actions. The optimal plan for the agent is a one action plan $\{move(v_1, v_n)\}$.*

Let us assume that we are given the following plan for redundancy elimination: $\{move(v_1, v_n)$, $move(v_n, v_1)$, $move(v_1, v_2)$, $move(v_2, v_3)$, ..., $move(v_{n-1}, v_n)\}$.

The plan can be made perfectly justified by either removing all but the first action (and obtaining the optimal plan) or by removing the first two actions (ending up a with a plan of n actions). Action elimination would remove the first two actions, for the SAT algorithm we cannot tell which actions would be removed, it depends on the satisfying assignment the SAT solver returns.

The example shows us, that it matters very much in what order we remove the actions and achieving perfect justification does not necessarily mean we did a good job. What we actually want is to remove as many actions as possible. How to do this efficiently is described in the next section.

Maximum Redundancy Elimination

In the section we describe how to do the best possible redundancy elimination for a plan. The problem of *maximum redundancy elimination (MRE)* is to find a subsequence R of redundant actions in a plan P, such that there is no other subsequence R' of redundant actions which is longer than R. A similar notion (minimal reduction) was defined for plans with actions costs (Nakhost and Müller 2010).

The plan resulting from MRE is always perfectly justified, on the other hand a plan might be perfectly justified and at the same time much longer than a plan obtained by MRE (see Example 1).

The solution we propose for MRE is also based on our redundancy encoding, but instead of a SAT solver we will use a partial maximum satisfiability (PMaxSAT) solver. We will construct a PMaxSAT formula, which is very similar to the formula used for redundancy elimination.

A PMaxSAT formula consists of hard and soft clauses. The hard clauses will be the clauses we used for redundancy elimination without the redundancy clause F_R.

$$H_{\Pi,P} = F_G \wedge F_A$$

The soft clauses will be unit clauses containing the negations of the action variables.

$$S_{\Pi,P} = \bigwedge_{a_i \in P} (\neg a_i)$$

The PMaxSAT solver will find an assignment ϕ that satisfies all the hard clauses (which enforces the validity of the plan P_ϕ due to Lemma 1) and satisfies as many soft clauses as possible (which removes as many actions as possible).

```
    MaximumRedundancyEliminaion (Π, P)
MR1     F := encodeMaximumRedundancy(Π, P)
MR2     φ := partialMaxSatSolver(F)
MR3     return P_φ
```

Figure 4: Pseudo-code of the maximum redundancy elimination algorithm.

The algorithm (Figure 4) is now very simple and straightforward. We just construct the formula and use a PMaxSAT solver to obtain an optimal satisfying assignment. Using this assignment we construct an improved plan the same way as we did in the SAT based redundancy elimination algorithm.

Experimental Evaluation

In this section we present the results of our experimental study regarding elimination of redundant actions from plans. We implemented the Action Elimination algorithm as well as the SAT and MaxSAT based algorithms and used plans obtained by several planners for the problems of the International Planning Competition (Coles et al. 2012).

Experimental Settings

Since, our tools take input in the SAS+ format, we used Helmert's translation tool, which is a part of the Fast Downward planning system (Helmert 2006), to translate the IPC benchmark problems that are provided in PDDL.

To obtain the initial plans, we used the following state-of-the-art planners: FastDownward (Helmert 2006), Metric FF (Hoffmann 2003), and Madagascar (Rintanen 2013). Each of these planners was configured to find plans as fast as possible and ignore plan quality.

We tested four redundancy elimination methods:

- *Action Elimination (AE)* is our own Java implementation of the Action Elimination algorithm as displayed in Figure 1.
- *Action Elimination + SAT (AE+S)* is an algorithm that first runs Action Elimination on the initial plan and incremental SAT reduction (see Figure 3) on the result. We used the incremental Java SAT solver Sat4j (Berre and Parrain 2010).
- *SAT Reduction (SAT)* is using the incremental SAT reduction directly without using Action Elimination for preprocessing (same as AE+S without AE).
- *Maximum Elimination (MAX)* is a Partial MaxSAT reduction based algorithm displayed in Figure 4. We implemented the translation in Java and used the QMaxSAT (Koshimura et al. 2012) state-of-the-art MaxSAT solver written in C++ to solve the instances.

For each of these methods we measured the total runtime and the total number of removed redundant actions for each domain and planner.

Table 1: Experimental results on the plans for the IPC 2011 domains found by the planners Fast Downward, Metric FF, and Madagascar. The planners were run with a time limit of 10 minutes. The column "#Plans" contains the number of plans found and "Length" represents the sum of their lengths. By Δ_{ALG} and T_{ALG} we mean the total number of removed redundant actions and the time in seconds it took for all plans for a given algorithm ALG. The algorithms are Action Elimination (AE), Action Elimination followed by SAT reduction (AE+S), SAT reduction on the original plan (SAT), and maximum elimination using a MaxSat solver (MAX).

	Domain	#Plans	Length	Δ_{AE}	T_{AE}	Δ_{AE+S}	T_{AE+S}	Δ_{SAT}	T_{SAT}	Δ_{MAX}	T_{MAX}
Metric FF	elevators	20	4273	79	0,81	79	2,31	79	3,14	79	0,17
	floortile	2	81	10	0,02	10	0,08	10	0,10	10	0,00
	nomystery	5	107	0	0,01	0	0,16	0	0,17	0	0,00
	parking	18	1546	124	0,18	124	1,13	124	1,60	124	0,03
	pegsol	20	637	0	0,10	0	1,10	0	1,16	0	0,02
	scanalyzer	18	571	30	0,06	30	0,78	30	0,88	30	0,01
	sokoban	13	2504	6	0,39	6	2,34	6	2,52	6	0,36
	tidybot	17	1136	144	0,17	144	0,92	144	1,66	144	0,04
	transport	6	1329	164	0,34	164	0,82	164	2,15	165	0,25
	visitall	3	1137	166	0,14	166	0,47	166	0,97	172	0,08
	woodworking	19	1471	22	0,37	22	1,14	22	1,28	22	0,02
Fast Downward	barman	20	3749	528	0,52	582	3,44	596	7,18	629	0,44
	elevators	20	4625	94	0,84	94	2,41	94	3,45	94	0,19
	floortile	5	234	22	0,06	22	0,20	22	0,27	22	0,00
	nomystery	13	451	0	0,05	0	0,47	0	0,48	0	0,00
	parking	20	1494	4	0,17	4	1,21	4	1,26	4	0,03
	pegsol	20	644	0	0,11	0	1,11	0	1,18	0	0,02
	scanalyzer	20	823	26	0,10	26	1,16	26	1,33	26	0,03
	sokoban	17	5094	244	0,62	458	5,25	458	8,39	460	1,84
	tidybot	16	1046	64	0,14	64	0,91	64	1,28	64	0,03
	transport	17	4059	289	0,65	289	1,64	289	2,93	290	0,20
	visitall	20	28776	122	3,66	122	9,47	122	12,89	122	7,77
	woodworking	20	1605	27	0,41	27	1,16	27	1,33	30	0,03
Madagascar	barman	8	1785	303	0,25	303	1,59	303	3,53	318	0,30
	elevators	20	11122	2848	1,46	3017	4,13	3021	17,62	3138	2,03
	floortile	20	1722	30	0,39	30	1,05	30	1,32	30	0,03
	nomystery	15	480	0	0,06	0	0,51	0	0,53	0	0,01
	parking	18	1663	152	0,20	152	1,17	152	1,78	152	0,03
	pegsol	19	603	0	0,09	0	1,06	0	1,10	0	0,01
	scanalyzer	18	1417	232	0,24	232	0,88	232	1,61	236	0,05
	sokoban	1	121	22	0,02	22	0,13	22	0,29	22	0,01
	tidybot	16	1224	348	0,16	348	0,84	348	2,13	350	0,08
	transport	4	1446	508	0,20	539	0,40	532	1,65	553	0,16
	woodworking	20	1325	0	0,31	0	1,11	0	1,21	0	0,01

All the experiments were run on a computer with Intel Core i7 960 CPU @ 3.20 GHz processor and 24 GB of memory. The planners had a time limit of 10 minutes to find the initial plans. The benchmark problems are taken from the satisficing track of IPC 2011 (Coles et al. 2012).

Experimental Results

The results of our experiments are displayed in Table 1. We can immediately notice that the runtime of all of our methods is very low. None of the methods takes more than one second on average for any of the plans. Note, that the runtime of the MAX method is often the smallest contrary to the fact, that it is the only one which guarantees eliminating the maximum number of redundant actions (AE+S and SAT only guarantee perfect justification).

Looking at the number of removed actions in Table 1 we can make several interesting observations. For example, in the nomystery and pegsol domains no redundant actions were found in plans obtained by any planner and also Madagascar's plans for the woodworking domain were always perfectly justified. In the most cases the AE algorithm provides perfectly justified plans (this is when the values of Δ_{AE} and Δ_{AE+S} are equal). The SAT method performs better than AE+S on barman for Fast Downward and elevators for Madagascar, but removes less actions on transport for Madagascar. Although both methods reach perfect justification, the results are different since removing redundant actions is not confluent (see example 1). As expected, the MAX method removes the highest (or equal) number of actions in each case. It is strictly dominant for 11 planner domain combinations. Considering the good runtime performance of this method we can conclude, that MAX is the best way of eliminating redundant actions.

Conclusions

In this paper, we have introduced a SAT encoding for the problem of detecting redundant actions in plans and used it to build two algorithms for plan optimization. One is based on SAT solving and the other on partial MaxSAT solving. Contrary to existing algorithms, both of our algorithms guarantee, that they output a plan with no redundant actions. Additionally, the MaxSAT based algorithm always eliminates a maximum set of redundant actions. According to our experiments done on IPC benchmark problems with plans obtained by state-of-the-art planners, our newly proposed algorithms perform very well in practice.

Acknowledgments The research is supported by the Czech Science Foundation under the contract P103/10/1287 and by the Grant Agency of Charles University under contract no. 600112. This research was also supported by the SVV project number 260 104.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for sas+ planning. *Computational Intelligence* 11:625–656.

Balyo, T.; Barták, R.; and Surynek, P. 2012. Shortening plans by local re-planning. In *Proceedings of ICTAI*, 1022–1028.

Berre, D. L., and Parrain, A. 2010. The sat4j library, release 2.2. *JSAT* 7(2-3):59–64.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012a. Determining redundant actions in sequential plans. In *Proceedings of ICTAI*, 484–491.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012b. Optimizing plans through analysis of action dependencies and independencies. In *Proceedings of ICAPS*, 338–342.

Coles, A. J.; Coles, A.; Olaya, A. G.; Celorrio, S. J.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A survey of the seventh international planning competition. *AI Magazine* 33(1).

Estrem, S. J., and Krebsbach, K. D. 2012. Airs: Anytime iterative refinement of a solution. In *Proceedings of FLAIRS*, 26–31.

Fikes, R., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.* 2(3/4):189–208.

Fink, E., and Yang, Q. 1992. Formalizing plan justifications. In *In Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence*, 9–14.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research (JAIR)* 20:239 – 290.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal Artificial Intelligence Research (JAIR)* 20:291–341.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of ECAI*, 359–363.

Koshimura, M.; Zhang, T.; Fujita, H.; and Hasegawa, R. 2012. Qmaxsat: A partial max-sat solver. *JSAT* 8(1/2):95–100.

Nakhost, H., and Müller, M. 2010. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In *Proceedings of ICAPS*, 121–128.

Rintanen, J. 2013. Planning as satisfiability: state of the art. http://users.cecs.anu.edu.au/ jussi/satplan.html.

Siddiqui, F. H., and Haslum, P. 2013. Plan quality optimisation via block decomposition. In *Proceedings of IJCAI*.

Westerberg, C. H., and Levine, J. 2001. Optimising plans using genetic programming. In *Proceedings of ECP*, 423–428.