On Different Strategies for Eliminating Redundant Actions from Plans

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

Lukáš Chrpa and **Asma Kilani**
PARK Research Group
School of Computing and Engineering
University of Huddersfield
{l.chrpa,u0950056}@hud.ac.uk

Abstract

Satisficing planning engines are often able to generate plans in a reasonable time, however, plans are often far from optimal. Such plans often contain a high number of redundant actions, that are actions, which can be removed without affecting the validity of the plans. Existing approaches for determining and eliminating redundant actions work in polynomial time, however, do not guarantee eliminating the 'best' set of redundant actions, since such a problem is NP-complete. We introduce an approach which encodes the problem of determining the 'best' set of redundant actions (i.e. having the maximum total-cost) as a weighted MaxSAT problem. Moreover, we adapt the existing polynomial technique which greedily tries to eliminate an action and its dependants from the plan in order to eliminate more expensive redundant actions. The proposed approaches are empirically compared to existing approaches on plans generated by state-of-the-art planning engines on standard planning benchmarks.

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 only by removing redundant actions from them which may serve as a pre-processing for other more complex plan optimization techniques. Even 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 of them. One of these algorithms, Action Elimination (Nakhost and Müller 2010), iteratively tries to remove an action and its dependants from the plan. We adapt the Action Elimination algorithm in order to take into account action cost and propose two new algorithms which guarantee to remove the 'best' set of redundant actions. One uses partial maximum satisfiability (PMaxSAT) solving, the other one relies on weighted partial maximum satisfiability (WPMaxSAT) solving. We compare our new algorithms with existing (polynomial) algorithms for removing redundant actions. The comparison is done on plans obtained by state-of-the-art planners on IPC-2011[1] benchmarks.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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 totally ordered plan.

[1] http://www.plg.inf.uc3m.es/ipc2011-deterministic

Determining and removing redundant actions from plans is a specific sub-category of post-planning plan optimization. Removing redundant actions from plans is a complementary technique to those mentioned above, since it can be used to 'pre-optimize' plans before applying a more complex optimization technique. 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, then 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 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.

Maximum Satisfiability

The *Maximum Satisfiability (MaxSAT)* problem is the problem of finding a truth assignment of a given CNF formula, such that the maximum number of clauses are satisfied. A *Weighted CNF (WCNF)* formula is a CNF formula where each clause has a non-negative integer weight assigned to it. The *Weighted MaxSAT (WMaxSAT)* problem is to find a truth assignment that maximizes the sum of the weights of satisfied clauses. MaxSAT is a special case of WMaxSAT where each clause has the same weight.

A *partial maximum satisfiability (PMaxSAT) formula* is a CNF formula consisting of two kinds of clauses called *hard* and *soft* clauses. The *partial maximum satisfiability* (PMaxSAT) problem is to find a truth assignment ϕ for a given PMaxSAT formula such that ϕ satisfies all the hard clauses and as many soft clauses as possible.

Similarly to MaxSAT, PMaxSAT also has a weighted version called *Weighted Partial MaxSAT* (WPMaxSAT). In WPMaxSAT the soft clauses have weights and the task is to find a truth assignments that satisfies all the hard clauses and maximizes the sum of the weights of satisfied soft clauses.

Planning

In this subsection 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 $\Pi = \langle X, O, s_I, s_G \rangle$ 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 $v \in \mathrm{dom}(x_i)$. Actions can have a non-negative integer cost assigned to them. We will denote by $C(a)$ the cost of an action a.
- 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. By $s_I \in S$ we denote the initial state. The goal conditions are expressed as a partial assignment s_G 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 $s_G \subseteq \mathrm{apply}(a_k, \mathrm{apply}(a_{k-1} \ldots \mathrm{apply}(a_2, \mathrm{apply}(a_1, s_I)) \ldots))$. By $P[i]$ we will mean the i-th action of a plan P. We will denote by $|P|$ the length of the plan P. The cost of a plan P is denoted by $C(P)$ and it is defined as the sum of the costs of the actions inside P, i.e., $C(P) := \sum\{C(P[i]);\ i \in 1 \ldots |P|\}$. A plan P for a planning task Π is called an *optimal plan* (*cost optimal plan*) if there is no other plan P' for Π such that $|P'| < |P|$ ($C(P') < C(P)$).

Redundant Actions in Plans

Intuitively, redundant actions in a plan are those, which if removed then the rest of the plan is still valid. Subsequences of original plans from which redundant actions were removed are called plan reductions (Nakhost and Müller 2010).

Definition 1. *Let P be a plan for a planning task Π. Let P' be a (possibly improper) subsequence of P. We say that P' is a* plan reduction *of P denoted as $P' \preceq P$ if and only if P' is also a plan for Π. If $P' = P$, then we say that P' is an* empty plan reduction *of P.*

Remark 1. *We can easily derive that $\preceq$ is a partial-order relation, since $\preceq$ is reflexive, transitive and antisymmetric.*

Definition 2. *We say that a plan P for a planning task Π is* redundant *if and only if there exists P' such that $P' \preceq P$ and $|P'| < |P|$ (P' is a non-empty plan reduction of P).*

In literature, a plan which is not redundant is called a *perfectly justified plan*. Note that deciding whether a plan is perfectly justified is NP-complete (Fink and Yang 1992). From this we can derive that deciding whether there exists a non-empty reduction of a given plan is NP-complete as well. Clearly, an optimal plan is not redundant (it is perfectly justified). However, as the following example shows we might obtain several different plan reductions of an original plan. Even if these reductions are perfectly justified plans, their quality might be (very) different.

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: $[move(v_1, v_n)$, $move(v_n, v_1)$, $move(v_1, v_2)$, $move(v_2, v_3)$, $\ldots$, $move(v_{n-1}, v_n)]$.

The plan can be made non-redundant 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).

Achieving maximum possible plan optimization just by removing redundant actions means that we have to identify the 'best' set of them. In other words, we have to find a minimal reduction of the original plan, which is also NP-complete (Nakhost and Müller 2010).

Definition 3. *Let P be a plan for a planning task Π. We say that P' is a* minimal plan reduction *of P if and only if $P' \preceq P$, P' is not a redundant plan and there is no P'' such that $P'' \preceq P$ and $C(P'') < C(P')$.*

Given the example, it matters which redundant actions and in what order are removed. Polynomial methods (discussed in the Related Work Section) such as Action Elimination (Nakhost and Müller 2010) are identifying and removing sets of redundant actions successively. Even if we get a non-redundant plan, which is not guaranteed by such methods, it might not necessarily be a minimal plan reduction. In other words, we can 'get stuck' in local minima.

Intuitively, compatible plan reductions are those that it is possible to successively eliminate the corresponding sets of redundant actions. The formal definition follows.

Definition 4. *Let P be a plan for a planning task Π. Let P^1 and P^2 be plan reductions of P (i.e. $P^1 \preceq P$ and $P^2 \preceq P$). We say that P^1 and P^2 are* compatible *if and only if there exists P^k such that $P^k \preceq P^1$ and $P^k \preceq P^2$. Otherwise, we say that P^1 and P^2 are* incompatible.

Informally speaking, having all plan reductions of a given plan compatible means that eliminating any set of redundant actions from the plan will not hinder the possibility of finding a minimal plan reduction afterwards. It is formally proved in the following lemma.

Lemma 1. *Let P be a plan for a planning task Π. Let $R = \{P' \mid P' \preceq P\}$ be a set of all plan reductions of P. If $\forall P^1, P^2 \in R$ it is the case that P^1 and P^2 are compatible, then for every $P' \in R$ there exists $P'' \in R$ such that $P'' \preceq P'$ and P'' is a minimal reduction of P (and P').*

Proof. Without loss of generality, we assume that P'' is a minimal plan reduction of P and P' is a plan reduction of P (not necessarily minimal). According to Definition 4 we can see that for P' and P'' there exists P^k such that $P^k \preceq P'$ and $P^k \preceq P''$. Since P'' is a minimal reduction of P and, clearly, P'' is not a redundant plan, we can derive that $P^k = P''$ and thus $P'' \preceq P'$. □

Deciding that all plan reductions are compatible is clearly intractable, since deciding whether a plan is redundant (it has a plan reduction) is intractable as well (as mentioned before). Despite this we believe that in some cases we might be able to use polynomial methods without losing the possibility to reach minimal plan reductions.

Greedy Action Elimination

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 by Fink and Yang (1992) under the name Linear Greedy Justification. It was reinvented by Nakhost and Müller (2010) and called Action Elimination. In this paper we use the latter name and extend the algorithm to take into account the action costs. Initially, we describe the original Action Elimination algorithm.

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 $O(n^2 p)$ time.

The Action Elimination algorithm ignores the cost of the actions and eliminates a set of redundant actions as soon as it discovers them. In this paper, we modify the Action Elimination algorithm to be less 'impatient'. Before removing any set of redundant actions, we will initially identify all the sets

```
ActionElimination (Π, P)
AE01  s := s_I
AE02  i := 1
AE03  repeat
AE04    mark(P[i])
AE05    s' := s
AE06    for j := i + 1 to |P| do
AE07      if applicable(P[j], s') then
AE08        s' := apply(P[j], s')
AE09      else
AE10        mark(P[j])
AE11    if goalSatisfied(Π, s') then
AE12      P := removeMarked(P)
AE13    else
AE14      unmarkAllActions()
AE15      s := apply(P[i], s)
AE16    i := i + 1
AE17  until i > |P|
AE18  return P
```

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

of redundant actions and then remove the one with the highest sum of costs of the actions in it. We will iterate this process until no more sets of redundant actions are found. We will call this new algorithm *Greedy Action Elimination*.

Greedy Action Elimination relies on two functions: evaluateRemove and remove (see Figure 2). The function evaluateRemove tests if the k-th action and the following actions that depend on it can be removed. It returns -1 if those actions cannot be removed, otherwise it returns the sum of their costs. The remove function returns a plan with the k-th action and all following actions that depend on it removed from a given plan. The Greedy Action Elimination algorithm (see Figure 3) calls evaluateRemove for each position in the plan and records the most costly set of redundant actions. The most costly set is removed and the search for sets of redundant actions is repeated until no such set is detected.

The worst case time complexity of Greedy Action Elimination is $O(n^3p)$, where $n = |P|$ and p is the maximum number of preconditions or effects any action in P has. This is due to the fact, that the main repeat loop runs at most n times (each time at least one action is eliminated) and the for loop calls n times evaluateRemove and once remove. Both these functions run in $O(np)$ time, therefore the total runtime of the algorithm is $O(n(n^2p + np)) = O(n^3p)$.

There are plans, where Action Elimination cannot eliminate all redundant actions (Nakhost and Müller 2010). This also holds for the Greedy Action Elimination. An interesting question is how often this occurs for the planning domains used in the planning competitions (Coles et al. 2012) and how much do the reduced plans differ from minimal plan reductions. To find that out, we first need to design algorithms that guarantee achieving minimal length reduction (eliminate the maximum number of redundant actions regardless

```
evaluateRemove (Π, P, k)
E01  s := s_I
E02  for i := 1 to k − 1 do
E03    s := apply(P[i],s)
E04  cost := C(P[k])
E05  for i := k + 1 to |P| do
E06    if applicable(P[i], s) then
E07      s := apply(P[i], s)
E08    else
E09      cost := cost + C(P[i])
E10  if goalSatisfied(Π, s) then
E11    return cost
E12  else
E13    return −1

remove (P, k)
R01  s := s_I
R02  P' := [ ]    // empty plan
R03  for i := 1 to k − 1 do
R04    s := apply(P[i],s)
R05    P' := append(P',P[i])
R06  for i := k + 1 to |P| do
R07    if applicable(P[i], s) then
R08      s := apply(P[i], s)
R09      P' := append(P', P[i])
R10  return P'
```

Figure 2: Pseudo-code of the evaluateRemove and remove functions used in the Greedy Action Elimination algorithm (see Figure 3).

their cost) and minimal plan reduction. As mentioned earlier, these problems are NP-complete and therefore we find it reasonable to solve it using a MaxSAT reduction approach. In the next section we will introduce an encoding of the problem of redundant actions into propositional logic.

Propositional Encoding of Plan Redundancy

This section is devoted to introducing an algorithm, which given a planning task Π and a plan P for Π, outputs a CNF formula $F_{\Pi,P}$, such that each satisfying assignment of $F_{\Pi,P}$ represents a plan reduction P' of P, i.e.,$P' \preceq P$.

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 (index) 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

```
    greedyActionElimination (Π, P)
G01   repeat
G02     bestCost := 0
G03     bestIndex := 0
G04     for i := 1 to |P| do
G05       cost := evaluateRemove(Π, P, i)
G06       if cost ≥ bestCost then
G07         bestCost := cost
G08         bestIndex := i
G09     if bestIndex ≠ 0 then
G10       P := remove(P, bestIndex)
G11   until bestIndex = 0
G12   return P
```

Figure 3: Pseudo-code of the Greedy Action Elimination algorithm, an action cost-aware version of the Action Elimination algorithm. It greedily removes the most costly sets of redundant actions.

plan. We will say that $a_i = True$ if the i-th action of P (the action with the rank i, i.e, $P[i]$) 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 $P[j]$ of c_i with the rank $j < i$ 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.

The main idea is similar to the ideas used in the 'relaxed' encodings for finding plans via SAT (Balyo 2013; Wehrle and Rintanen 2007; Rintanen, Heljanko, and Niemelä 2006).

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 $P[j]$ ($j \in Supps(c, i)$) with rank $j < i$ 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 $P[i]$ with the rank i we will require that if its action variable a_i is true, then all of its preconditions must be true at time i. For an action $P[i]$ the following clauses will enforce, that if the action variable a_i is true, then all the preconditions must hold.

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

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

$$F_A = \bigwedge_{i \in 1 \ldots |P|} F_{a_i}$$

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 whole formula $F_{\Pi,P}$ is the conjunction of the goal clauses, and the action dependency clauses for each action in P.

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

From a satisfying assignment of this formula we can produce a 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, i.e., $P[i]$ is present in P_ϕ if and only if $\phi(a_i) = True$.

Proposition 1. *An assignment ϕ satisfies $F_{\Pi,P}$ if and only if P_ϕ is a 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 needs to be applied. □

The following observations follow directly from the Proposition. The formula $F_{\Pi,P}$ is always satisfiable for any planning task Π and its plan P. One satisfying assignment

ϕ has all variables a_i set to the value $True$. In this case, the plan P_ϕ is identical to the input plan P. If P is already a perfectly justified plan, then there is no other satisfying assignment of $F_{\Pi,P}$ since all the actions in P are necessary to solve the planning task.

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 $n^3p + n^2g + np + g$ clauses, from which $n^3p + n^2g$ are binary clauses.*

Proof. There 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 is 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. □

Minimal Length Reduction

In this section we describe how to do the best possible redundancy elimination for a plan if we do not consider action costs. This process is called Minimal Length Reduction (MLR) and its goal is to remove a maximum number of actions from a plan. It is a special case of minimal reduction, where all actions have unit cost.

The plan resulting from MLR 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 MLR (see Example 1).

The solution we propose for MLR is based on our redundancy encoding and using a partial maximum satisfiability (PMaxSAT) solver.

Recall, that a PMaxSAT formula consists of hard and soft clauses. The hard clauses will be the clauses we introduced in the previous section.

$$H_{\Pi,P} = F_{\Pi,P}$$

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 Proposition 1) and satisfies as many soft clauses as possible (which removes as many actions as possible).

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 ϕ. Using this assignment we construct an improved plan P_ϕ as defined in the previous section.

```
    MinimalLengthReduction (Π,P)
MLR1  F := encodeMinimalLengthReduction(Π,P)
MLR2  φ := partialMaxSatSolver(F)
MLR3  return P_φ
```

Figure 4: Pseudo-code of the minimal length reduction algorithm.

```
    MinimalReduction (Π, P)
MR1  F := encodeMinimalReduction(Π, P)
MR2  φ := weightedPartialMaxSatSolver(F)
MR3  P' := MinimalLengthReduction (Π, P_φ)
MR4  return P'
```

Figure 5: Pseudo-code of the minimal reduction algorithm.

Minimal Reduction

The problem of minimal (plan) reduction can be solved in a very similar way to MLR. The difference is that we need to construct a Weighted Partial MaxSAT (WPMaxSAT) formula and use a WPMaxSAT solver.

Our WPMaxSAT formula is very similar to the PMaxSAT formula from the previous section. The hard clauses are again equal to $F_{\Pi,P}$ and the soft clauses are unit clauses containing the negations of the action variables. Each of these unit clauses has an associated weight, which is the cost of the corresponding action. The WPMaxSAT solver will find a truth assignment ϕ that maximizes the weight of satisfied soft clauses which is equivalent to removing actions with a maximal total cost. The validity of the plan P_ϕ obtained from the optimal assignment ϕ is guaranteed thanks to Proposition 1 and the fact that all the hard clauses must be satisfied under ϕ.

The plan P_ϕ obtained from the WPMaxSAT solution ϕ may contain redundant actions of zero cost. To obtain a perfectly justified plan we will run MLR on P_ϕ. The algorithm is displayed in Figure 5.

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 (AE) algorithm as well as its greedy variant and the PMaxSAT and WPMaxSat based algorithms – minimal length reduction (MLR) and minimal reduction (MR). We used plans obtained by three state-of-the-art satisficing planners for the problems of the International Planning Competition (Coles et al. 2012) and compared the algorithms with each other and with a plan optimization tool which focuses on redundant inverse actions elimination (IAE) (Chrpa, McCluskey, and Osborne 2012a).

Experimental Settings

Since, our tools take input in the SAS+ format, we used Helmert's translation tool, which is a part of the Fast Down-

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 'Found Plan' contains the number and the total cost of the found plans. The following columns contain the total cost of the eliminated actions (Δ) and the total time in seconds (T[s]) required for the optimization process for the five evaluated algorithms.

Domain		Found Plan		IAE		AE		Greedy AE		MLR		MR	
		Nr.	Cost	Δ	T[s]	Δ	T[s]	Δ	T[s]	Δ	T[s]	Δ	T[s]
Metric FF	elevators	20	25618	2842	1,31	2842	0,87	2842	0,91	2842	0,17	2842	1,77
	floortile	2	195	29	0,00	30	0,01	30	0,03	30	0,00	30	0,00
	nomystery	5	107	0	0,14	0	0,01	0	0,01	0	0,00	0	0,00
	parking	18	1546	118	0,16	124	0,10	124	0,30	124	0,03	124	0,27
	pegsol	20	300	0	0,00	0	0,07	0	0,06	0	0,02	0	0,28
	scanalyzer	18	1137	0	0,00	62	0,04	62	0,06	62	0,01	62	0,17
	sokoban	13	608	0	0,71	2	0,40	2	0,33	2	0,36	2	9,12
	transport	6	29674	2650	0,32	3013	0,27	3035	0,52	3035	0,25	3035	3,34
Fast Downward	barman	20	7763	436	0,98	753	0,51	780	1,08	926	0,43	926	10,85
	elevators	20	28127	1068	1,51	1218	0,79	1218	1,20	1218	0,19	1218	1,99
	floortile	5	572	66	0,00	66	0,04	66	0,08	66	0,00	66	0,01
	nomystery	13	451	0	4,25	0	0,04	0	0,04	0	0,01	0	0,04
	parking	20	1494	4	0,06	4	0,09	4	0,10	4	0,04	4	0,21
	pegsol	20	307	0	0,00	0	0,06	0	0,06	0	0,02	0	0,30
	scanalyzer	20	1785	0	0,01	78	0,06	78	0,08	78	0,04	78	0,49
	sokoban	17	1239	0	6,48	58	0,53	58	0,75	102	1,92	102	250,27
	transport	17	74960	4194	1,11	5259	0,56	5260	1,02	5260	0,19	5260	1,92
Madagascar	barman	8	3360	296	0,97	591	0,25	598	0,52	606	0,28	606	6,33
	elevators	20	117641	7014	6,77	24096	1,21	24728	10,44	28865	1,90	28933	37,34
	floortile	20	4438	96	0,09	96	0,31	96	0,37	96	0,04	96	0,24
	nomystery	15	480	0	2,63	0	0,04	0	0,04	0	0,01	0	0,02
	parking	18	1663	152	0,17	152	0,12	152	0,40	152	0,04	152	0,36
	pegsol	19	280	0	0,00	0	0,05	0	0,06	0	0,01	0	0,26
	scanalyzer	18	1875	0	0,05	232	0,19	236	0,47	236	0,04	236	0,31
	sokoban	1	33	0	0,01	0	0,02	0	0,04	0	0,01	0	0,19
	transport	4	20496	4222	0,23	6928	0,20	7507	0,56	7736	0,16	7736	9,56

ward 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 five redundancy elimination methods:

- *Inverse action elimination (IAE)* is implemented in C++ and incorporates techniques described in (Chrpa, McCluskey, and Osborne 2012a)
- *Action Elimination (AE)* is our own Java implementation of the Action Elimination algorithm as displayed in Figure 1.
- *Greedy Action Elimination (Greedy AE)* is our Java implementation of the Greedy Action Elimination algorithm as displayed in Figure 3.
- *Minimal Length Reduction (MLR)* 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. We selected QMaxSAT due to its availability and very good results in the 2013 MaxSAT competition.
- *Minimal Reduction (MR)* is a Weighted Partial MaxSAT reduction based algorithm displayed in Figure 5. The translation is implemented in Java and we used the Toysat (Sakai 2014) Weighted MaxSAT solver written in Haskell

to solve the instances. Although Toysat did not place very well in the 2013 MaxSAT competition, it significantly outperformed all the other available solvers on our formulas.

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

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 the methods is usually very low. Most of the optimizations run under a second and none of the methods takes more than two seconds on average for any of the plans except for the Fast Downward plans for the sokoban domain.

Looking at the total cost 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.

The IAE method does not often eliminate as many redundant actions as the other methods. Clearly, this is because of the IAE method is specific, i.e., only pairs or nested pairs of inverse actions are considered. Surprisingly, the runtime is high despite the complexity results (Chrpa, McCluskey, and Osborne 2012a). This can be partially explained by the fact, that IAE takes input in the PDDL format, which is much more complex to process than the SAS+ format.

The Action Elimination (AE) algorithm, although it ignores the action costs, performs rather well. Except for eight planner/domain combinations it achieves minimal reduction, i.e., the best possible result. Furthermore, AE has consistently very low runtime.

The Greedy Action Elimination algorithm improves upon AE in seven cases and achieves minimal reduction in all but five planner/domain pairs. The runtime of the greedy variant is not significantly increased in most of the cases except for the Madagascar plans for the elevators domain. Overall, we can say, that using Greedy AE instead of just plain AE pays off in most of the cases.

The Minimal Length Reduction (MLR) algorithm is guaranteed to remove the maximum number of redundant actions (not considering their cost) and this is also enough to achieve minimal reduction in each case except for the Madagascar plans for the elevators domain. The runtime of this method is very good, considering it is guaranteed to find an optimal solution for an NP-hard problem.

The last and the strongest of the evaluated algorithms, Minimal Reduction (MR), is guaranteed to achieve minimal reduction. Nevertheless, its runtime is still very reasonable for each planner/domain pair. Even the 250 seconds required to optimize the 17 sokoban plans from Fast Downward is negligible compared to the time planners need to solve this difficult domain.

Discussion

Clearly, the WPMaxSAT based method is guaranteed to provide minimal reduction of plans and therefore cannot be outperformed (in terms of quality) by the other methods. Despite NP-completeness of this problem, runtimes are usually very low and in many cases even lower than the other polynomial methods we compared with. On the other hand, when the problem becomes harder the runtimes can significantly increase (e.g in the Sokoban domain). We have observed that the problem of determining redundant actions (including minimal reduction) is in most of the cases very easy. Therefore, runtime often depends more on the efficiency of implementation of particular methods rather than the worst case complexity bounds.

Our results also show that in the most cases using the polynomial method (AE or Greedy AE) provides minimal reduction, so the WMaxSAT method usually does not lead to strictly better results. Guaranteeing in which cases (Greedy) AE provides minimal reduction is an interesting open question. To answer this question we have to identify cases in which we can decide in polynomial time that the optimized plan is not redundant as well as that all reductions of the original plan are compatible (see Lemma 1).

Conclusions

In this paper we adapted the Action Elimination algorithm to work with action costs and experimentally showed that it often improves plans better that the original Action Elimination algorithm. We have also 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 partial MaxSAT solving and the other on weighted partial MaxSAT solving. Contrary to existing algorithms, both of our algorithms guarantee, that they output a plan with no redundant actions. Additionally, the partial MaxSAT based algorithm always eliminates a maximum number of redundant actions and the weighted partial MaxSAT based algorithm removes a set of actions with the highest total cost.

According to our experiments we have done on the IPC benchmarks with plans obtained by state-of-the-art planners, our newly proposed algorithms perform very well in general. However, in a few cases the weighted MaxSAT algorithm is slow. This supports the complexity results for the problem of finding minimal reduction. Interestingly, (Greedy) AE has found minimal reductions in many cases. Therefore, it is an interesting question in which cases we can guarantee finding minimal reductions by the (greedy) AE.

In future, apart from addressing that question, we plan to incorporate the IAE algorithm into the (greedy) AE one. For determining redundancy of pairs of inverse actions we need to investigate only actions placed in between (Chrpa, McCluskey, and Osborne 2012b), so we believe that we can improve the efficiency of the (greedy) AE algorithm.

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.

Balyo, T. 2013. Relaxing the relaxed exist-step parallel planning semantics. In *ICTAI*, 865–871. IEEE.

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 *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.

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.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.* 170(12-13):1031–1080.

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

Sakai, M. 2014. Toysolver home page. https://github.com/msakai/toysolver.

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

Wehrle, M., and Rintanen, J. 2007. Planning as satisfiability with relaxed exist-step plans. In Orgun, M. A., and Thornton, J., eds., *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, 244–253. Springer.

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