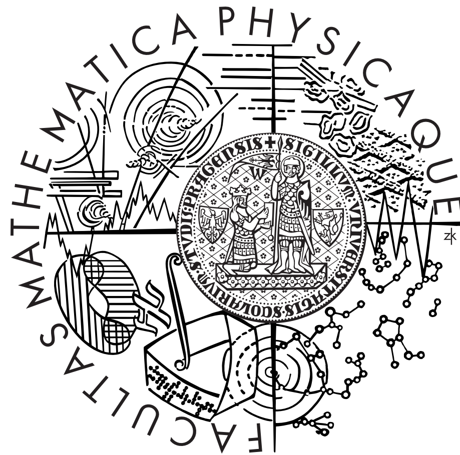
Charles University in Prague

Faculty of Mathematics and Physics

DOCTORAL THESIS



Tomáš Balyo

Modelling and Solving Problems Using SAT Techniques

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the doctoral thesis:	Roman Barták
Study programme:	Computer Science
Specialization:	Theoretical Computer Science

Prague 2014

I wish to thank my supervisor prof. RNDr. Roman Barták, Ph.D. He has generously given his time, talents and advice to assist me in my research and the production of this thesis.

I would also like to thank all my research collaborators and anonymous reviewers, who helped me to publish my results and provided useful feedback.

I thank the the Charles University, the Faculty of Mathematics and Physics and the Grant Agency of the Charles University for their financial support.

Last but not least, I thank my loving family, who supported my studies.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, 09.06.2014 signature of the author

Název práce: Modelling and Solving Problems Using SAT Techniques

Autor: Tomáš Balyo

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí disertační práce: prof. RNDr. Roman Barták, Ph.D., KTIML, MFF UK

Abstrakt: Řešení problémů plánování prostřednictvím překladů do splnitelnosti (SAT) je jedním z nejúspěšnějších přístupů k automatickému plánování. V této práci popíšeme několik způsobů jak přeložit problém plánování reprezentovaný v SAS+ formalismu do SAT. Přezkoumáme a přizpůsobíme stávající kódování a také zavedeme nové vlastní způsoby kódování. Porovnáme jednotlivá kódování pomocí výpočtu horních odhadů na velikosti formulí, které produkují, a pomocí spuštění rozsáhlých experimentů na referenčních problémech z Mezinárodní plánovací soutěže 2011. V experimentální části také porovnáme své kódování s nejmodernejšími kódováními z plánovače Madagascar. Experimenty ukazují, že naše techniky dokažou překonat tato kódování. V předložené práci také řešíme speciální případ optimalizace plánů – odstranění redundantních akcí. Odstranění všech redundantních akcí je NP-úplný problém. Prostudujeme existující polynomialní heuristické přístupy a navrhneme vlastní heuristický přístup, který dokaže eliminovat vyšší počet a dražší redundantní akce než stávající techniky. Také navrhneme způsob kódování problému redundance plánů do SAT, který nám za použití MaxSAT řešičů umožní optimálně vyřešit problém eliminace redundantních akcí. Naše experimenty provedené s plány od nejmodernejších satisficing plánovačů pro referenční problémy prokázaly, že všechny námi navrhované techniky fungují v praxi velmi dobře.

Klíčová slova: Splnitelnost, Plánování, Modelování

Title: Modelling and Solving Problems Using SAT Techniques

Author: Tomáš Balyo

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., KTIML, MFF UK

Abstract: Solving planning problems via translation to satisfiability (SAT) is one of the most successful approaches to automated planning. In this thesis we describe several ways of encoding a planning problem represented in the SAS+ formalism into SAT. We review and adapt existing encoding schemes as well as introduce new original encodings. We compare the encodings by calculating upper bounds on the size of the formulas they produce as well as by running extensive experiments on benchmark problems from the 2011 International Planning Competition (IPC). In the experimental section we also compare our encodings with the state-of-the-art encodings of the planner Madagascar. The experiments show, that our techniques can outperform these state-of-the-art encodings. In the presented thesis we also deal with a special case of post-planning optimization – elimination of redundant actions. The elimination of all redundant actions is NP-complete. We review the existing polynomial heuristic approaches and propose our own heuristic approach which can eliminate a higher number and more costly redundant actions than the existing techniques. We also propose a SAT encoding for the problem of plan redundancy which together with MaxSAT solvers allows us to solve the problem of action elimination optimally. Experiments done with plans from state-of-the-art satisficing planners for IPC benchmark problems demonstrate, that all the proposed techniques work well in practice.

Keywords: Satisfiability, Planning, Modeling

Contents

1 Introduction **3**
- 1.1 Contributions . . . 4
- 1.2 Overview by Chapters . . . 4

2 Preliminaries **6**
- 2.1 Satisfiability (SAT) . . . 6
- 2.2 Maximum Satisfiability (MaxSAT) . . . 7
- 2.3 Planning . . . 8
  - 2.3.1 Redundant Actions . . . 10
  - 2.3.2 Parallel Plans . . . 12

3 Finding Plans using SAT **15**
- 3.1 The Direct Encoding . . . 15
- 3.2 The SASE Encoding . . . 19
- 3.3 The Reinforced Encoding . . . 23
- 3.4 The Exist Step Encoding . . . 26
  - 3.4.1 Variables and Plan Extraction . . . 26
  - 3.4.2 Action Ranking . . . 26
  - 3.4.3 Basic Clauses . . . 28
  - 3.4.4 Action Interference Clauses . . . 30
  - 3.4.5 Correctness . . . 32
- 3.5 Selective Encoding . . . 33
- 3.6 Properties of the Encoded Formulas . . . 34
  - 3.6.1 Planning Task Parameters . . . 34
  - 3.6.2 Number of Variables . . . 35
  - 3.6.3 Number of Clauses . . . 35
- 3.7 Experiments . . . 38
  - 3.7.1 Experimental Setting . . . 39
  - 3.7.2 Ranking Comparison . . . 39
  - 3.7.3 Performance Results . . . 40
  - 3.7.4 Properties of the Formulas . . . 43
  - 3.7.5 Discussion . . . 45

4 Improving Plans **48**
- 4.1 Related Work . . . 48
- 4.2 Greedy Action Elimination . . . 50
- 4.3 Propositional Encoding of Plan Redundancy . . . 52
  - 4.3.1 Correctness and Size . . . 54
- 4.4 Making Plans Perfectly Justified . . . 56
- 4.5 Minimal Length Reduction and Minimal Reduction . . . 58
- 4.6 Experiments . . . 59
  - 4.6.1 Experimental Setting . . . 59

4.6.2 Number of Removed Actions 60
4.6.3 Cost of Removed Actions 63
4.6.4 Runtime 64
4.6.5 Discussion 65

Conclusion **67**
Conclusion 67

1. Introduction

One of the most studied problems in computer science, both theoretical and applied, is the Boolean satisfiability problem (SAT). SAT solvers have seen a lot of progress in the last two decades which allowed SAT solving to become a core component in many different applications. One of the first and most successful applications of SAT was automated planning [28].

Planning [22] is the problem of finding a sequence of actions – a plan, that transforms the world from some initial state to a goal state. In this thesis we will consider only the simplest (most limited) definition of planning – often referred to as *classical* or *STRIPS* planning. The constraints of classical planning are the following.

- The world is fully-observable, deterministic and static (only the agent, that we make the plan for can change the world).
- The number of the possible states of the world as well as the number of possible actions is finite, though possibly very large.
- We will assume, that the actions are instantaneous (take a constant time) and therefore we only need to deal with their sequencing.

Other kinds of planning such as *temporal* and *probabilistic* planning [22], which remove these limitations, are also studied. Their advantage is that they model the real world more faithfully and thus are more applicable. On the other hand, solving these kinds of problems can be much harder and there are not many efficient planners capable of solving them.

Actions have preconditions, which specify in which states of the world they can be applied as well as effects, which dictate how the world will be changed after the action is executed. The actions may have a cost assigned to them. A cost of an action is a non-negative integer, and a cost of plan is the sum of the costs of the actions in it. The general goal is to find plans with low costs. The task of finding plans with the lowest cost (*optimal* plans) is called *optimal planning*. The time required to find optimal plans is usually very high and we are often satisfied with suboptimal plans that are 'good enough'. The task of finding 'good enough' plans is referred to as *satisficing* planning.

The complexity difference between optimal and satisficing planning depends on the planning problem instance. There are instances where a plan can be found in polynomial time while finding an optimal plan is NP-hard [11]. However, in many cases (and in general) already finding a plan of arbitrary quality (but with polynomial length) is NP-complete [12]. Determining whether there is a solution (a plan) for a given planning task is in general PSPACE-complete [12].

Despite the complexity results many 'real world' planning tasks can be successfully solved by modern state-of-the-art planning systems. Satisficing planners such as FF [25], Fast Downward [23] or LPG [21] are very efficient on a wide range of problems. They are based on heuristic guided search of the state space of a

given planning problem. Another approach, formerly very successfully used for optimal planning [27] and currently also for satisficing planning [33], is translating the planning task into a series of propositional satisfiability (SAT) formulas and then using a SAT solver.

The method was first introduced by Kautz and Selman [27] and is still very popular and competitive. This is partly due to the power of SAT solvers, which are getting more efficient year by year. Since then many new improvements have been made to the method, such as new compact and efficient encodings [26, 33, 34], better ways of scheduling the SAT solvers [33] or modifying the SAT solver's heuristics to be more suitable for solving planning problems [31].

In this thesis we will deal with the problem of encoding a planning task to SAT in order to efficiently find its solution (a plan). We will also show how SAT and MaxSAT techniques can be used to improve the quality of an already obtained plan by removing useless (redundant) actions.

1.1 Contributions

The main contribution of this thesis is the definition of new encoding schemes for planning as SAT and their theoretical and experimental evaluation. These encodings are used for two purposes. One is finding plans and the other is improving plans found by other planning systems by removing redundant actions from them.

For the first purpose we introduce two new encoding schemes. One is called the Reinforced encoding and it slightly improves upon the existing so called ∀-step semantics based encodings. The other is a new innovative encoding called the Relaxed Relaxed ∃-step encoding which works well on planning problems that were previously very difficult to solve using SAT based techniques. We also define a simple rule, that given a planning problem instance can predict which encoding scheme is more suitable for solving the instance. Parts of these results are already published in a conference paper [2].

For the second purpose we introduce a propositional encoding for the problem of plan redundancy. We use this encoding to generate SAT and MaxSAT formulas which allows us to efficiently solve NP-hard plan optimization problems, which were previously only addressed by using heuristic algorithms. We also introduce our own heuristic algorithm which improves upon the existing ones. The results are already accepted for publication [5] and submitted to a conference and being reviewed at the time of writing of this thesis [6].

1.2 Overview by Chapters

The thesis is organized in the following way.

- The second chapter contains the preliminary definitions and basic propositions used in the rest of the text. Most of the definitions are well known

but we also provide some new ones regarding redundant action elimination and parallel plans.

- Chapter three deals with the problem of finding plans using a SAT solver. We describe several ways of encoding a planning task into a series of SAT formulas. We review a basic encoding scheme [27] and adapt it for the SAS+ planning formalism. Then we describe the well known SASE encoding [26], which was the first planning encoding scheme using the SAS+ formalism. After that, as the main contribution of the chapter, we present two new original encoding schemes and also provide a rule which helps us to choose between them for a specific planning task. The chapter ends with a theoretical and experimental comparison of the described encodings with each other and state-of-the art encodings of Rintanen [33].

- The fourth chapter deals with post-planning optimization, i.e., improving plans which were obtained by an arbitrary planning algorithm. We focus on removing redundant actions from plans. First, we review a heuristic algorithm used for this purpose [30]. Then we improve this algorithm and provide several new algorithms for redundancy elimination based on using SAT and MaxSAT solvers. Our new algorithms are capable of optimally solving all the variants of the plan redundancy elimination problem, which are all NP-hard. The chapter ends with an experimental comparison of the proposed and existing methods.

- Finally, the conclusion contains a summary of the thesis and proposes some research directions for future work.

2. Preliminaries

In this Chapter we give the basic definitions and properties related to satisfiability, maximum satisfiability and planning.

2.1 Satisfiability (SAT)

SAT is one the most important problems in computer science. It was the first problem proven to be NP-hard [17]. Despite its complexity there are very efficient SAT solvers [9] which make it possible to design successful algorithms for hard problems by translating them to SAT. In this section we review the basic definitions and properties related to satisfiability, which can be found in any SAT related textbook (for example The Handbook of Satisfiability [10]).

The input of the SAT problem is a CNF formula. The definition follows.

Definition 1 (CNF Formula)**.** *A* Boolean variable *is a variable with two possible values* True *and* False*. A* literal *of a Boolean variable x is either x or $\neg x$, i.e.,* positive *or* negative literal*. A* clause *is a disjunction (OR) of literals. A* conjunctive normal form (CNF) formula *is a conjunction (AND) of clauses.*

Next we define what is a satisfying assignment.

Definition 2 (Satisfying Assignment)**.** *A* truth assignment *ϕ of a formula F assigns a* truth value *to its variables. The assignment ϕ satisfies*

- *a positive literal if it assigns the value True to its variable,*
- *a negative literal if it assigns the value False to its variable,*
- *a clause if it satisfies at least one of its literals,*
- *a CNF formula if it satisfies each one of its clauses.*

If ϕ satisfies a CNF formula F, then ϕ is called a satisfying assignment *of F.*

The definition of satisfiability follows.

Definition 3 (Satisfiability)**.** *A formula F is said to be* satisfiable *if there is a truth assignment ϕ that satisfies F, i.e. ϕ is a satisfying assignment of F. Otherwise, the formula ϕ is* unsatisfiable*.*

The problem of satisfiability (SAT) *is to determine whether a given formula F is satisfiable or unsatisfiable.*

A *SAT solver* is a procedure that solves the SAT problem. For satisfiable formulas we also expect a SAT solver to return a satisfying assignment. An example of a satisfiable CNF formula with its satisfying assignment follows.

Example 1. *$F = (x_1 \lor x_2 \lor \neg x_4) \land (x_3 \lor \neg x_1) \land (\neg x_1 \lor \neg x_2)$ is CNF formula with 3 clauses $\{(x_1 \lor x_2 \lor \neg x_4), (x_3 \lor \neg x_1), (\neg x_1 \lor \neg x_2)\}$ and 6 literals $\{x_1, \neg x_1, x_2, \neg x_2, x_3, \neg x_4\}$ on 4 variables $\{x_1, x_2, x_3, x_4\}$. F is satisfiable with $\phi = \{x_1 \to False, x_2 \to True, x_3 \to True, x_4 \to True\}$ being a satisfying truth assignment of F.*

The are some special kinds of clauses that are interesting. A clause with only one literal is called a *unit clause* and with two literals a *binary clause*. A *Horn* clause is clause with at most one positive literal. A CNF formula consisting of only Horn clauses is called a *Horn formula* and a formula with only unit and binary clauses is called a *quadratic formula*. The satisfiability of Horn and quadratic formulas can be determined in polynomial time [10].

We will often need to express implications in our formula. It is easy to verify, that an implication of the form $x \Rightarrow (y_1 \lor \cdots \lor y_k)$ is equivalent to the clause $(\neg x \lor y_1 \lor \cdots \lor y_k)$.

2.2 Maximum Satisfiability (MaxSAT)

In this section we review the definitions of a family of SAT related optimization problems called maximum satisfiability. The presented definitions and basic properties are well know and can be found in any SAT related textbook [10].

Definition 4 (Maximum Satisfiability)**.** *The problem of* Maximum Satisfiability (MaxSAT) *is the problem of finding a truth assignment of a given CNF formula that satisfies the maximum number of its clauses.*

A MaxSAT solver determines what is the maximum number of clauses that can be satisfied in a given CNF formula and finds a truth assignment that satisfies that many clauses. If the input CNF formula is satisfiable, then the MaxSAT solvers returns a satisfying assignment.

In MaxSAT we have no control over which clauses are satisfied in an optimal solution. It might be the case, that for some of the clauses it is essential, that they are satisfied under the found assignment. This issue is addressed by the Partial MaxSAT problem.

Definition 5 (Partial MaxSAT)**.** *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 problem *(PMaxSAT) is to find a truth assignment for a given PMaxSAT formula that satisfies all the hard clauses and as many soft clauses as possible.*

There are two special cases, one is that all the clauses are hard, the other is that all the clauses are soft. In the first case PMaxSAT is equivalent to SAT, in the second to MaxSAT.

In the situation, that not all the clauses are equally important, we can assign weights to them and we obtain the weighted MaxSAT problem.

Definition 6 (Weighted MaxSAT)**.** *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 for a given WCNF formula that maximizes the sum of the weights of satisfied clauses.*

If all the clauses have the same weight, then WMaxSAT is equivalent to MaxSAT. A WMaxSAT solver can also be used to solve PMaxSAT problems if the weights of the clauses are properly chosen. This is achieved by setting the weights of the soft clauses to 1 and the weights of the hard clauses to a value higher than the number of soft clauses.

The last of the MaxSAT problems is the weighted partial MaxSAT problem (WPMaxSAT). Although the situation it represents can be easily expressed as a WMaxSAT problem, we define it separately for convenience.

Definition 7 (Weighted Partial MaxSAT)**.** *A* weighted partial maximum satisfiability (WPMaxSAT) formula *is a CNF formula consisting of two kinds of clauses called* hard *and* soft *clauses. Additionally, each soft clause has a non-negative integer weight assigned to it. The* Weighted Partial MaxSAT (WPMaxSAT) *problem is to find a truth assignment for a given WPMaxSAT formula that satisfies all its hard clauses and maximizes the sum of the weights of satisfied soft clauses.*

2.3 Planning

In the introduction we briefly described what planning is, in this section we give the formal definitions. We will use the multivalued SAS+ formalism [1] instead of the classical STRIPS formalism [19] based on propositional logic.

Definition 8 (Planning Task)**.** *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* $\mathrm{dom}(x_i) \subset \mathbb{N}$.
- *O is a set of actions (or operators). An 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* $v \in \mathrm{dom}(x_i)$*. The actions may 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.* $s_I \in S$ *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 can be applied to a state if it its preconditions are satisfied. For example if $x_i = v$ is in the preconditions of an action, then the action can be

applied only to states where the state variable x_i is equal to v. Similarly, the effects of an applied action change the world state. If $x_i = v$ is an effect of an action, then after the application of this action the value of x_i will become v. A formal definition follows.

Definition 9 (Applicable Actions). *An action a is* applicable *in a given state s if* $\text{pre}(a) \subseteq s$. *By $s' = \text{apply}(a, s)$ 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* $\text{eff}(a)$ *which replace the corresponding (same variable) assignments in s. If $A = [a_1, \dots, a_k]$ is a sequence of actions, then* $\text{apply}(A, s) = \text{apply}(a_k, \text{apply}(a_{k-1} \dots \text{apply}(a_2, \text{apply}(a_1, s)) \dots))$.

Now we are ready to give the definition of a plan, which is a solution for a planning task.

Definition 10 (Sequential Plan). *A* sequential plan *P of* length *k for a planning task $\Pi = \{X, O, s_I, s_G\}$ is a sequence of actions $P = [a_1, \dots, a_k]; a_i \in O$ such that $s_G \subseteq \text{apply}(P, s_I)$.*

We will denote by $|P| = k$ the length of a plan P and by $P[i]$ we will mean the i-th action of P, i.e., $P[i] = a_i$. If P contains actions with costs, then we define the *cost of a plan*, $C(P)$, to be the sum of the costs of the actions in it, i.e., $C(P) = \sum\{C(P[i]);\ i \in 1 \dots |P|\}$.

The quality of a plan is measured by the number and the cost of its actions. Shorter plans with lower cost actions are preferable. An optimal plan for a given planning task is such a plan, that there exist no strictly better plans for that task.

Definition 11 (Optimal 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 $|P'| < |P|$. Similarly, a plan P is called* cost optimal *if there is no other plan P' such that $C(P') < C(P)$.*

The following example illustrates the above defined terms.

Example 2. *In this example we will model a simple package delivery scenario. We have a truck that needs to deliver two packages to the location C from the locations A and B. In the beginning the truck is located in A. The locations A, B, and C are connected by roads (see Figure 2.1).*

We will model the planning task using the following variables:

- *Truck location T, dom(T)* $= \{A, B, C\}$
- *Package locations P_1 and P_2, dom(P_1) = dom(P_2)* $= \{A, B, C, L\}$ *($P_i = L$ represents that the package i is inside the truck).*

Now, having defined the variables $X = \{T, P_1, P_2\}$ and their respective domains, we can define the initial state s_I and the goal conditions s_G.

$$s_I = \{T = A, P_1 = A, P_2 = B\} \qquad s_G = \{P_1 = C, P_2 = C\}$$

Finally, we need to define the set of actions with their preconditions and effects. The action templates are described in the following table.

Figure 2.1: The initial and the goal state for the logistics planning example

Action	*Preconditions*	*Effects*	*Description*
$move(l1, l2)$	$T = l1$	$T = l2$	*move the truck from location* $l1$ *to* $l2$
$loadP_1(l)$	$T = l, P_1 = l$	$P_1 = L$	*load* P_1 *on the truck at location* l
$loadP_2(l)$	$T = l, P_2 = l$	$P_2 = L$	*load* P_2 *on the truck at location* l
$unloadP_1(l)$	$T = l, P_1 = L$	$P_1 = l$	*unload* P_1 *from the truck at location* l
$unloadP_2(l)$	$T = l, P_2 = L$	$P_2 = l$	*unload* P_2 *from the truck at location* l

To get the actual actions we need to substitute $l, l1, l2$ *with* A, B *and* C.

One of the possible solutions is the following plan $P = [loadP_1(A), move(A, C), unloadP_1(C), move(C, B), loadP_2(B), move(B, C), unloadP_2(C), move(C, A)]$. *The plan* P *is valid, but it is not an optimal plan. It contains 8 actions while the following (optimal) plan has only 6 actions:* $P^* = [loadP_1(A), move(A, B), loadP_2(B), move(B, C), unloadP_1(C), unloadP_2(C)]$.

2.3.1 Redundant Actions

Suboptimal plans often contain actions that can be removed without affecting their validity. Such actions are called redundant. A plan reduction is a subsequence of a plan with some redundant actions removed. In this subsection we review the definitions of redundancy related problems (first defined in [20, 30]).

Definition 12 (Plan Reduction)**.** *Let* P *be a plan for a planning task* Π. *Let* P' *be a 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. *The actions in* P *that are not present in* P' *are called* redundant actions*.*

Note, that adding new actions into the plan or reordering actions is not allowed. Using these operations would in many cases allow us to improve plans even further. Nevertheless, plan reduction is restricted to just removing redundant actions.

Definition 13 (Redundant and Perfectly Justified Plans)**.** *We say that a plan* P *for a planning task* Π *is* redundant *if and only if there exists a plan reduction of* P. *A plan which is not redundant is called a* perfectly justified plan.

Example 3. *Using the package delivery planning problem of Example 2 we can observe, that the last action of P ($move(C, A)$) is redundant. By removing this action from the plan we get a perfectly justified plan P' which is a plan reduction of P ($P' \preceq P$). Nevertheless, P' is not optimal, since it contains 7 actions, while the optimal plan P^* from Example 2 contains only 6 actions. Note, that P^* is not a plan reduction of P, which means we could not get to P^* from P by just removing actions.*

From the example it is apparent, that a perfectly justified plan is not necessarily an optimal plan. On the other hand, an optimal plan is always perfectly justified. The example also shows, that an optimal plan may not be reachable from a given plan just by removing redundant actions from it. The following example demonstrates that even if an optimal plan is a plan reduction of the input plan and we reach a perfectly justified plan, we might still end up with a non-optimal plan.

Example 4. *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 complete 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).

Let us consider action costs and a case, where each move action has a cost 1 except for the $move(v_1, v_n)$ action with a cost $n + 1$. In this case the cost optimal plan reduction is achieved by removing the first two actions.

The example shows, that it matters very much in what order we remove the redundant 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. We can formally define this problem in the following way.

Definition 14 (Minimal Length Plan Reduction)**.** *Let P be a plan for a planning task Π. We say that P' is a* minimal length plan reduction *of P if and only if $P' \preceq P$ and there is no P'' such that $P'' \preceq P$ and $|P''| < |P'|$.*

Analogously, we define the problem of *minimal plan reduction*, which is the problem of removing the most costly subsequence of redundant actions from a plan.

Definition 15 (Minimal Plan Reduction)**.** *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 $P' \preceq P$ and there is no P'' such that $P'' \preceq P$ and $C(P'') < (P')$.*

The decision version of each of these problems (given a plan P and a number L, is there a plan reduction of length/cost at most L?) is NP-complete [20, 30]. From this it also follows, that just checking if a given plan is perfectly justified is NP-complete, which is equivalent to the question whether at least one action can be removed.

2.3.2 Parallel Plans

One of the most important aspects of planning as SAT is the usage of parallel plans. Intuitively, the idea of parallel plans is that the plan is a sequence of sets of actions, such that the actions inside one set can be executed in parallel (at the same time together) [26, 11]. These sets are called *parallel steps* and the number of parallel steps in a parallel plan is called the *makespan*.

In modern SAT encodings [33, 38, 2] this concept is generalized, in particular, the requirement, that actions inside a set can be executed in parallel is removed. Our generalized definition follows.

Definition 16 (Action Ordering Function)**.** *An* action ordering function $\trianglelefteq$ *transforms a set of actions A into a sequence of actions $\trianglelefteq(A)$ in a way that each action of A appears exactly once in $\trianglelefteq(A)$.*

Definition 17 (Parallel Plan)**.** *A sequence of sets of actions $P = [A_1, \ldots, A_k]$ is a* parallel plan *for a planning task Π if there is an action ordering function $\trianglelefteq$ such that $[\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)]$ is a (sequential) plan for Π, where $\oplus$ denotes the concatenation of sequences. The sets A_i are called* parallel steps *and k is called the* makespan *of P.*

The reason, why parallel plans are important for SAT based planning, is that it allows us to solve a planning task with fewer SAT solver calls. We will require only as many solver calls as the makespan of the resulting plan instead of its length (see Figure 3.1).

Various parallel plan semantics define which actions can be together inside a parallel step. In the following set of definitions we will denote by s_j the world state in between the parallel steps A_j and A_{j+1}, which is obtained by applying the sequence $\trianglelefteq(A_j)$ on s_{j-1} (except for $s_0 = s_I$).

Definition 18 ($\forall$-Step semantics)**.** *A parallel plan $P = [A_1, \ldots, A_k]$ satisfies the $\forall$-Step semantics [11, 33] if*

- *each action in A_j is applicable in the state s_j,*
- *the effects of all the actions in A_j are applied in s_{j+1}, and*
- *$[\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)]$ is a valid plan for each ordering function $\trianglelefteq$.*

We will show, that to ensure, that each ordering of the sets of actions in a parallel plan leads to a valid sequential plan, it is sufficient to check that the actions in each set are pairwise independent. The definition of action independence follows.

Definition 19 (Independent Actions)**.** *We say that two actions* a_1 *and* a_2 *are* independent *if they do not share common variables, i.e.,* $\text{scope}(a_1) \cap \text{scope}(a_2) = \emptyset$*, where* $\text{scope}(a) \subseteq X$ *is a set of all state variables that appear in* $\text{pre}(a)$ *and* $\text{eff}(a)$*.*

Proposition 1. *Let* A *be a set of actions such that* $\forall a_i \neq a_j \in A$ *the actions* a_i *and* a_j *are independent. If there is an ordering* $\trianglelefteq$ *such that* $\trianglelefteq(A)$ *transforms the state* s_1 *to* s_2 *then all the possible orderings of* A *transform* s_1 *to* s_2*.*

Proof. The pair-wise independence of the actions in A implies, that each state variable is both required and changed only once and only by one action during any application of the actions in A. Therefore the actions are applicable in any order and the resulting state is the same for each ordering. □

Note, that the pairwise independence of actions is a sufficient but not a necessary condition for the parallel steps in a $\forall$-Step semantics plan, as the following examples demonstrates.

Example 5. *Let* a_1 *and* a_2 *be two actions such that* $\text{pre}(a_1) = \text{pre}(a_2) = \{x = 1\}$, $\text{eff}(a_1) = \{y = 2\}$*, and* $\text{eff}(a_1) = \{z = 2\}$*. Clearly,* a_1 *and* a_2 *are not independent (they share the variable* x*), however, they can be ordered arbitrarily to achieve the same changes between two given states.*

The following semantics – the $\exists$-Step[33] – weakens the requirement on the ordering of the actions, and only requires, that there exists at least one 'good' action ordering function.

Definition 20 ($\exists$-Step semantics)**.** *A parallel plan* $P = [A_1, \dots, A_k]$ *satisfies the* $\exists$*-Step semantics [33] if*

- *each action in* A_j *is applicable in the state* s_j*,*
- *the effects of all the actions in* A_j *are applied in* s_{j+1}*, and*
- *there exists an action ordering function* $\trianglelefteq$ *such that* $[\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)]$ *is a valid sequential plan.*

The next semantics is the Relaxed $\exists$-Step semantics [38], which removes the requirement that the actions in A_j must be applicable in s_j. Thus some of the action in A_j might become applicable only after some other actions in A_j are applied. However, the effects of all the action are still applied in the next state.

Definition 21 (Relaxed $\exists$-Step semantics)**.** *A parallel plan* $P = [A_1, \dots, A_k]$ *satisfies the Relaxed* $\exists$*-Step semantics [38] if*

- *the effects of all the actions in* A_j *are applied in* s_{j+1}*, and*
- *there exists an action ordering function* $\trianglelefteq$ *such that* $[\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)]$ *is a valid sequential plan.*

Lastly, our newly defined semantics removes the last requirement which can be removed, i.e., that the effects of the actions must be applied in the next state. Therefore we only require, that the actions in a parallel step can be ordered properly.

Definition 22 (Relaxed Relaxed ∃-Step semantics)**.** *A parallel plan* $P = [A_1, \ldots, A_k]$ *satisfies the Relaxed Relaxed ∃-Step (*R^2*∃-Step) semantics if*

- *there exists an action ordering function* $\trianglelefteq$ *such that* $[\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)]$ *is a valid sequential plan.*

The following example demonstrates the differences between the four semantics and how the more relaxed semantics can allow shorter makespan plans.

Example 6. *Using the package delivery planning problem of Example 2 let us examine the optimal sequential plan* $P^* = [loadP_1(A),\ move(A, B),\ loadP_2(B),\ move(B, C),\ unloadP_1(C),\ unloadP_2(C)]$. *The four semantics allow four different parallel plans:*

- *The* R^2*∃-Step semantics allows the single step plan* $[\{loadP_1(A),\ move(A, B),\ loadP_2(B),\ move(B, C),\ unloadP_1(C),\ unloadP_2(C)\}]$.

- *The Relaxed ∃-Step semantics forbids having both move operations inside one step, since the effect of the first move action is not applied after the step. Therefore the shortest possible Relaxed ∃-Step plan is the following two step parallel plan* $[\{loadP_1(A),\ move(A, B),\ loadP_2(B)\},\ \{move(B, C),\ unloadP_1(C),\ unloadP_2(C)\}]$.

- *The ∃-Step semantics does not allow the second load action to be inside the first step, since its precondition is not satisfied in the initial state. Similarly, the unload actions cannot be in the same step as the second move action. Therefore the plan needs to have three parallel steps* $[\{loadP_1(A),\ move(A, B)\},\ \{loadP_2(B),\ move(B, C)\},\ \{unloadP_1(C),\ unloadP_2(C)\}]$.

- *To satisfy the ∀-Step semantics all actions, except for the unload actions, must be in separate parallel steps. The following five step plan is possible.* $[\{loadP_1(A)\},\ \{move(A, B)\},\ \{loadP_2(B)\},\ \{move(B, C)\},\ \{unloadP_1(C),\ unloadP_2(C)\}]$.

  The two unload actions can be together in the last step despite the fact that they are not independent. However, if we follow the rule, that only independent actions can be together in a step, then the plan has to have six parallel steps.

3. Finding Plans using SAT

The basic idea of solving planning as SAT is the following [27]. We construct (by encoding the planning task) a series of SAT formulas $F_1, F_2, \dots$ such that F_i is satisfiable if there is a parallel plan of makespan $\leq i$. Then we solve them one by one starting from F_1 until we reach the first satisfiable formula F_k. From the satisfying assignment of F_k we can extract a plan of makespan k. The pseudo-code of this algorithm is presented in Figure 3.1

```
SP1  PlanningAsSat (Π)
SP2    k := 0
SP3    repeat
SP4      k := k + 1
SP5      F := encodePlanningTaskWithMakespan(Π, k)
SP6    until isSatisfiable(F)
SP7    P := extractPlan(getSatAssignment(F))
SP8    return P
```

Figure 3.1: Pseudo-code of the basic planning as satisfiability algorithm.

The method was first introduced by Kautz and Selman [27] and is still very popular and competitive. This is partly due to the power of SAT solvers, which are getting more efficient year by year. Since then many new improvements have been made to the method, such as new compact and efficient encodings [26, 33, 34], better ways of scheduling the SAT solvers [33] or modifying the SAT solver's heuristics to be more suitable for solving planning problems [31]. Clever ways of solver scheduling [33] can significantly improve the performance of the planning algorithm at the cost of possibly longer makespan plans. Nevertheless, we will use the basic one-by-one scheduling since we are interested only in comparing the properties of encodings, i.e., the construction of the formulas F_i.

The following four sections will each describe a way of encoding a planning task into SAT. In the fifth section we will discuss how to select the proper encoding for a given planning task. The sixth section will compare the encodings by examining the properties of the formulas they produce. The last section of the chapter will experimentally compare the encodings with each other and two state-of-the-art encodings on benchmark problems from the international planning competition.

3.1 The Direct Encoding

The simplest and most straightforward way of encoding a planning task into SAT is following the definition of the planning problem and translating it into propositional logic. This encoding was the first one used in SAT based planners [28, 11]. Originally it was described in the context of the propositional planning formalism

(STRIPS [19]). Here we adapt it for the multivalued SAS+ formalism [1]. This encoding will use the ∀-Step parallel planning semantics.

Our goal is to, given a planning task $\Pi = \{X, O, s_I, s_G\}$ and an integer k, construct a CNF formula F_k such that F_k is satisfiable only if there is a parallel plan of at most k steps for Π. We also want to construct F_k in a way, that in the case it is satisfiable, we can easily extract a plan from its satisfying assignment.

In the Direct encoding we will use two kinds of Boolean variables.

- *Action variables* a_i^t indicating whether the i-th action is used in the t-th step. We will have one such variable for each action $a \in O$ from the description of the planning task and each of the k parallel steps.
- *Assignment variables* $b_{x=v}^t$ indicating whether the value of the variable x is equal to v in the beginning of the t-th step (before applying the actions of the t-th step). We will have one such Boolean variable for each state variable $x \in X$ and each value $v \in \text{dom}(x)$ for each of the k parallel steps and one extra set for the special $(k+1)$-th step which contains no actions and is used to represent the goal state.

Now we will describe the clauses of F_k. The first two sets of clauses will ensure that the assignment variables represent a valid planning state, i.e., each variable has exactly one value. The following clauses ensure that each assignment variable has at least one value.

$$\begin{gathered}(b_{x=v_1}^t \vee b_{x=v_2}^t \vee \cdots \vee b_{x=v_d}^t) \\ \forall x \in X,\ \text{dom}(x) = \{v_1, v_2, \ldots, v_d\},\ \forall t \in \{1, \ldots, k+1\}\end{gathered} \tag{3.1}$$

The next set of binary clauses will enforce, that at most one value is assigned to each state variable $x \in X$.

$$\begin{gathered}(\neg b_{x=v_i}^t \vee \neg b_{x=v_j}^t) \\ \forall x \in X,\ v_i \neq v_j,\ \{v_i, v_j\} \subseteq \text{dom}(x),\ \forall t \in \{1, \ldots, k+1\}\end{gathered} \tag{3.2}$$

These two sets of clauses guarantee, that any satisfying assignment of a formula containing them represents a valid assignment of values to the state variables.

Next we will define the clauses that connect the assignment and action variables. Following the definition of ∀-Step semantics, we need to ensure, that if an action is in the plan at step k, then all of its precondition assignments must hold at the beginning of the k-th step.

$$\begin{gathered}(\neg a^t \vee b_{x=v}^t) \\ \forall a \in O,\ \forall (x = v) \in \text{pre}(a),\ \forall t \in \{1, \ldots, k\}\end{gathered} \tag{3.3}$$

Similarly, we add clauses to force the effects of the actions in the next time step.

$$\begin{gathered}(\neg a^t \vee b_{x=v}^{t+1}) \\ \forall a \in O,\ \forall (x = v) \in \text{eff}(a),\ \forall t \in \{1, \ldots, k\}\end{gathered} \tag{3.4}$$

We also need to ensure, that the values of the state variables have not changed between two neighboring parallel steps, unless some actions changed them.

$$(\neg b^{t+1}_{x=v} \vee b^t_{x=v} \vee a^t_{s_1} \vee \cdots \vee a^t_{s_j}) \\ \forall x \in X,\ \forall v \in \mathrm{dom}(x),\ \mathrm{support}(x=v) = \{a_{s_1}, \dots, a_{s_j}\},\ \forall t \in \{1, \dots, k\} \tag{3.5}$$

By $\mathrm{support}(x = v) \subseteq O$ we mean the set of *supporting actions* of the assignment $x = v$, i.e., the set of actions that have $x = v$ as one of their effects. The clause (in 3.5) represents the implication, that if x has the value v at time $t + 1$ then it either already had the value v at time t or that one of the supporting actions of the assignment $x = v$ is in the k-th step.

Next we need to deal with the interfering actions inside a parallel step. According to Proposition 1 it is sufficient to ensure, that only pair-wise independent actions are together in step. We will achieve this by disabling all pairs of non-independent (interfering) actions. To extrude interfering actions from the parallel steps we will add binary clauses for all the interfering action pairs.

$$(\neg a^t_i \vee \neg a^t_j) \\ \forall \{a_i, a_j\} \subseteq O,\ a_i, a_j \,\text{not independent},\ \forall t \in \{1, \dots, k\} \tag{3.6}$$

There might be a plenty of interfering action pairs producing a lot of clauses. But if we look carefully at the clauses we have already described, we can see, that most of the interfering actions cannot occur together anyway as we will show via the following notion of compatible actions.

Definition 23 (Compatible Actions)**.** *Two sets of conditions (assignments) are compatible if they assign the same values to the variables they share.*

Two actions a_1 and a_2 are compatible *if the preconditions of a_1 are compatible with the preconditions of a_2 and also the effects of a_1 are compatible with the effects of a_2.*

Due to the clauses that enforce, that actions imply their preconditions 3.3 and effects 3.4, and the clauses that forbid a state variable to have more than one value 3.2, actions that are not compatible cannot be in a parallel step together. Therefore it is enough to only suppress compatible interfering action pairs.

$$(\neg a^t_i \vee \neg a^t_j) \\ \forall \{a_i, a_j\} \subseteq O,\ a_i, a_j \,\text{compatible and not independent},\ \forall t \in \{1, \dots, k\} \tag{3.7}$$

The last two sets of clauses we need to define are encodings of the initial state and the goal conditions. Both of these sets consist of unit clauses on the assignment variables. For the initial state we add unit clauses saying that the variables are assigned to their initial values at the beginning of the first parallel step.

$$(b^1_{x=v}) \\ \forall (x = v) \in s_I \tag{3.8}$$

Similarly, to express the goal conditions we require the corresponding assignments after the last parallel step (at time $k+1$).

$$\begin{array}{c}(b^{k+1}_{x=v})\\ \forall(x=v)\in s_G\end{array} \tag{3.9}$$

The final formula F_k is the conjunction of the clauses defined in the equations 3.1, 3.2, 3.3, 3.4, 3.5, 3.7, 3.8, and 3.9. If F_k is satisfiable, then a parallel plan P_ϕ can be extracted from its satisfying truth assignment ϕ in the following way.

Definition 24 (Plan Extraction)**.** *Let ϕ be a satisfying assignment of F_k. P_ϕ is a sequence of action sets such that its t-th set contains those actions $a_i \in O$ for which $\phi(a_i^t) = True$.*

We conclude this section by the following proposition about the correctness of the encoding.

Proposition 2. *If the formula F_k obtained using the Direct encoding of the planning task Π is satisfied by a truth assignment ϕ then P_ϕ is a valid $\forall$-Step parallel plan of makespan k for the planning task Π.*

Proof. The requirements for the action sets given by the $\forall$-Step semantics are clearly satisfied:

- the preconditions of actions are satisfied due to 3.3
- the effects are propagated thanks to 3.4
- the actions can be ordered arbitrarily since non-independent pairs are disabled due to 3.7 which is sufficient thanks to Proposition 1. Also the non-compatible action pairs do not need to be disabled as discussed in the text above.

It remains to prove that $P^S_\phi = [\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)]$ is a (sequential) plan for Π, where $\oplus$ denotes the concatenation of sequences and $\trianglelefteq$ is an arbitrary ordering of an action set.

A sequential plan is valid if all the actions are applicable in the given order and the goal conditions are satisfied in the end.

First, let us observe, that the values of the state variables are consistent at each step (due to 3.1 and 3.2) and do not change between two neighboring steps without an action changing them (thanks to 3.5). Since the action variables imply their precondition and effect variables (3.3, 3.4) the actions must be applicable if their action variable is True and also their effects must hold in the next state.

Thanks to 3.8 the state variables are set to the initial state values before the first action and due to 3.9 the goal conditions hold after the last set of actions. This fact together with the consistency of the state variables during all the k steps implies the validity of P^S_ϕ for Π. $\square$

The reversed implication, which is that if P_ϕ is a valid ∀-Step parallel plan of makespan k, then ϕ satisfies F_k, does not hold. This is because P_ϕ may contain a non-independent pair of actions in one of its steps and still be a valid ∀-Step plan (see Example 5). Such a pair of actions would make one of the clauses of type 3.7 unsatisfied.

3.2 The SASE Encoding

The encoding we review in this section was historically the first SAT encoding of a planning task based on the SAS+ formalism [26]. The encoding we present is slightly modified compared to the original SASE paper [26]. In particular, we use a different encoding of the initial state and goal conditions as well as the interference of transitions is defined in a more strict manner (see our recent paper [4] for more information about transition interference). The encoding again uses the ∀-step parallel planning semantics and we will again describe it by showing how the formula F_k is constructed.

In contrast to the Direct encoding presented in the previous section, the SASE encoding uses transition variables instead of assignment variables. The definition of a transition follows.

Definition 25 (Transition)**.** *A* transition *represents a change of a state variable* $x \in X$ *from one value to another from its domain* $\mathrm{dom}(x)$ *or from an arbitrary value to a specific value. There are the following three kinds of transitions.*

- *An* active *transition changes the value of the variable* x *from* d *to* e *such that* $d \neq e$*,* $\{d, e\} \subseteq \mathrm{dom}(x)$*, it is denoted by* $\delta_{x\colon d \to e}$*. An action* a *has an active transition* $\delta_{x\colon d \to e}$ *if* $(x = d) \in \mathrm{pre}(a)$ *and* $(x = e) \in \mathrm{eff}(a)$*.*

- *A* prevailing *transition conserves the value of the variable* x *(if it was* d*, then it remains* d*,* $d \in \mathrm{dom}(x)$*), it is denoted by* $\delta_{x\colon d \to d}$*. An action* a *has a prevailing transition* $\delta_{x\colon d \to d}$ *if* $(x = d) \in \mathrm{pre}(a)$ *and there is no assignment related to* x *in* $\mathrm{eff}(a)$*.*

- *A* mechanical *transition changes the value of the variable* x *from any value to the value* d *(*$d \in \mathrm{dom}(x)$*), it is denoted by* $\delta_{x\colon * \to d}$*. An action* a *has a mechanical transition* $\delta_{x\colon * \to d}$ *if* $(x = d) \in \mathrm{eff}(a)$ *and there is no assignment related to* x *in* $\mathrm{pre}(a)$*.*

Example 7. *The action* a *with preconditions* $\mathrm{pre}(a) = \{x = 1, y = 3\}$ *and effects* $\mathrm{eff}(a) = \{y = 1, z = 2\}$ *has one active transition (*$\delta_{y\colon 3 \to 1}$*), one prevailing transition (*$\delta_{x\colon 1 \to 1}$*), and one mechanical transition (*$\delta_{z\colon * \to 2}$*).*

The *transition set* of an action a is the set of all transitions that a has, it is denoted by Δ_a. By Δ_p we will mean the set of all possible prevailing transitions of a planning task, i.e., $\Delta_p = \{\delta_{x\colon d \to d} \mid x \in X, d \in \mathrm{dom}(x)\}$. The set of all transitions Δ is the union of all the prevailing transitions and the transition sets

of all the actions $\Delta = \Delta_p \cup \{\Delta_a \mid a \in O\}$. By $\Delta_x \subseteq \Delta$ where $x \in X$ we will denote the set of all transitions related to the variable x.

In the SASE encoding we will have one Boolean variable for each transition in Δ and one for each action $a \in O$. Therefore, we have the following two kinds of Boolean variables.

- *Action variables* a_i^t indicating whether the i-th action is used in the t-th step (same as the action variables in the Direct encoding).
- *Transition variables* c_δ^t (or $c_{x:\,d\to e}^t$ where $\delta = \delta_{x:\,d\to e}$) indicating whether the transition δ occurred during the t-th step. We will have one such variable for each $\delta \in \Delta$ for each of the k parallel steps.

Now, we will describe the clauses in F_k. We will start with clauses to ensure that exactly one transition can happen for each variable at each time step. The following clauses say that at least one transition for each variable must happen. Bear in mind, that the sets Δ_x contain all the prevailing transitions, therefore the values of the variables do not have to change necessarily during each parallel step.

$$\begin{gathered}(c_{\delta_1}^t \vee \cdots \vee c_{\delta_j}^t) \\ \forall x \in X, \Delta_x = \{\delta_1, \ldots, \delta_j\}, \forall t \in \{1, \ldots, k\}\end{gathered} \tag{3.10}$$

The following set of clauses ensures that at most one transition is allowed for each state variable.

$$\begin{gathered}(\neg c_{\delta_1}^t \vee \neg c_{\delta_2}^t) \\ \forall x \in X, \forall \{\delta_1, \delta_2\} \subseteq \Delta_x, \delta_1 \neq \delta_2, \forall t \in \{1, \ldots, k\}\end{gathered} \tag{3.11}$$

In the original SASE paper [26] a more complex and less strict definition of interfering transitions was used. The difference is, that according to the original definition, two transitions $\delta_1, \delta_2 \in \Delta_x$ do not interfere if one of the transitions is mechanical and both δ_1 and δ_2 transfer x to the same value. We have shown in our paper [4] that this less strict definition is not consistent with the $\forall$-Step parallel planning semantics and it slightly decreases the performance of the SASE encoding. Therefore, we will use the simpler and more strict definition, that two transitions interfere if they are on the same state variable.

Next we describe the clauses that connect the action variables with the transition variables. If an action a is selected, then all the transitions in its transition set Δ_a must be selected as well. This implication is expressed via the following clauses.

$$\begin{gathered}(\neg a^t \vee c_\delta^t) \\ \forall a \in O, \forall \delta \in \Delta_a, \forall t \in \{1, \ldots, k\}\end{gathered} \tag{3.12}$$

Also we need to make sure, that transitions (except for prevailing transitions) cannot happen without actions that have them in their transition sets. The

following set of clauses will ensure this.

$$
\begin{aligned}
&(\neg c_\delta^t \vee a_{s_1}^t \vee \cdots \vee a_{s_m}^t) \\
\forall \delta \in (\Delta \setminus \Delta_p), &\operatorname{support}(\delta) = \{s_1, \ldots, s_m\}, \forall t \in \{1, \ldots, k\}
\end{aligned}
\tag{3.13}
$$

By $\operatorname{support}(\delta)$ we mean the set of indices of actions that have δ in their transition set, i.e., $\operatorname{support}(\delta) = \{i \mid a_i \in O;\ \delta \in \Delta_{a_i}\}$.

The next set of clauses connects the transitions of two neighboring parallel steps and ensures that the values of state variables cannot change arbitrarily between them. We will encode the property that if a transition $\delta_{x:\, d \to e}$ happens during the $(t+1)$-th step, then there has to be a transition in the t-th step that changes x from some value to d.

$$
\begin{aligned}
&(\neg c_{\delta_{x:\, d\to e}}^{t+1} \vee c_{\delta_{x:\, v_1 \to d}}^t \vee \cdots \vee c_{\delta_{x:\, v_m \to d}}^t) \\
\forall x \in X,\ \forall \delta_{x:\, d \to e} \in \Delta_x, &\operatorname{dom}(x) = \{v_1, \ldots v_m\}\ \forall t \in \{1, \ldots, k\}
\end{aligned}
\tag{3.14}
$$

Similarly to the Direct encoding we need to disable the interfering action pairs. The definition of interfering actions remains the same as well as the fact that only compatible actions need to be suppressed explicitly. Therefore the clauses defined in equation 3.7 will also work for the SASE formula.

Lastly, we add the clauses that enforce the initial state to hold in the beginning and the goal conditions to be satisfied in the end. As for the initial state, we will disable all the transitions that are not compatible with the initial state, i.e., if a variable x has the value d in the initial state, then all the transitions that change x from a value other than d are disabled by using a unit clause. Note, that mechanical transitions are always compatible with the initial state (or any other state) and therefore no mechanical transition is disabled.

$$
\begin{aligned}
&(\neg c_{\delta_{x:\, d\to e}}^1) \\
\forall \delta_{x:\, d \to e} &\in \Delta, (x = d) \notin s_I
\end{aligned}
\tag{3.15}
$$

Similarly, we will disable all the transitions that change a variable to a value different from its goal value. Here, we must be careful not to disable transitions for variables that do not appear in the goal conditions.

$$
\begin{aligned}
&(\neg c_{\delta_{x:\, d\to e}}^k) \\
\forall \delta_{x:\, d \to e} \in \Delta, &(x = e) \notin s_G, \exists v\ (x = v) \in s_G
\end{aligned}
\tag{3.16}
$$

In the original SASE encoding [26] the initial state and the goal conditions were encoded using one clause for each variable that represented the set of possible transitions, i.e., transitions that change the state variable from its initial value to some value or change from some value to a goal value. Thus the initial state clauses would be the following.

$$
\begin{aligned}
&(c_{\delta_{x:\, d\to v_1}}^1 \vee \cdots \vee c_{\delta_{x:\, d\to v_m}}^1) \\
\forall x \in X, &(x = d) \in s_I, \operatorname{dom}(x) = \{v_1, \ldots v_m\}
\end{aligned}
$$

Similarly, the goal condition clauses would be the following.

$$(c^k_{\delta_{x:\, v_1 \to d}} \vee \cdots \vee c^k_{\delta_{x:\, v_m \to d}})$$
$$\forall x \in X, (x = d) \in s_G, \mathrm{dom}(x) = \{v_1, \ldots v_m\}$$

Thus the original SASE encoded F_k contained no unit clauses at all. We believe it is important to have unit clauses in F_k in order to speed up SAT solving. There are even special preprocessing techniques for SAT that require the presence of unit clauses in formulas [7].

We have described all the clauses required for the construction of F_k using the SASE encoding. The formula F_k is the conjunction of the clauses in equations 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, and 3.7 (from the Direct encoding). A parallel plan P_ϕ can be extracted from the satisfying assignment ϕ of F_k (if it is satisfiable) in the same way as for the Direct encoding, i.e., as defined in Definition 24. The following proposition (analogous to Proposition 2) holds for the SASE encoding.

Proposition 3. *If the formula F_k obtained using the SASE encoding of the planning task Π is satisfied by a truth assignment ϕ then P_ϕ is a valid $\forall$-Step parallel plan of makespan k for the planning task Π.*

Proof. This proof is very similar to the proof of Proposition 2. The only difference is that the values of the state variables are not expressed directly using assignment variables but via transitions.

The requirements for the action sets given by the $\forall$-Step semantics are clearly satisfied:

- the preconditions of actions are satisfied due to 3.12
- the effects are propagated also due to 3.12
- the actions can be ordered arbitrarily for the same reasons as for the Direct encoding, since the clauses 3.7 are also included in the SASE encoding (see the proof of Proposition 2).

It remains to prove that $P^S_\phi = [\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)]$ is a valid (sequential) plan for Π, where $\oplus$ denotes the concatenation of sequences and $\trianglelefteq$ is an arbitrary ordering of an action set.

First, let us observe, that the transitions of the state variables are consistent at each step, i.e., exactly one transition is allowed for each state variable (due to 3.10 and 3.11) and a transition cannot happen without an action that has it (thanks to 3.13). Furthermore, the transitions between the parallel steps must be compatible due to 3.14.

Since the action variables imply the proper transition variables thanks to 3.12, the actions must be applicable if their action variable is True and also the transition connected to the action must happen.

Thanks to 3.15 only transitions compatible with the initial state can happen in the first step and because of 3.16 only transitions that change the variables

to their goal values are allowed in the last step. This fact together with the consistency of the transitions during all the k steps implies the validity of P^S_ϕ for the planning task Π. $\square$

Note, that the reversed implication does not hold for the same reasons as for the Direct encoding (see the discussion in the last paragraph of the previous section).

3.3 The Reinforced Encoding

In this section we introduce a new encoding which is a combination of the Direct and SASE encodings described in the previous two sections. The encoding contains all three kinds of variables (action, assignment, and transition) and shares many of the clauses used in the Direct and SASE encodings. The name of the encoding comes from the idea of reinforcing one encoding with the strengths of the other. In other words, we are 'reinforcing' the direct encoding by using transition variables, or alternatively, we are 'reinforcing' the SASE encoding by using assignment variables.

Using more variables will also reduce the number of clauses. For example, there could be as many as $O(\sum_{x \in X} |\operatorname{dom}(x)|^4)$ clauses in the SASE encoding to ensure that only one transition is allowed for each variable (equation 3.11). It follows from the fact, that if $d = |\operatorname{dom}(x)|$ then there could be as many as $O(d^2)$ transitions on the variable x and we need to disable each pair, which leads to $O(d^4)$ binary clauses (equation 3.11). The reinforced encoding avoids these clauses by using assignment variables and clauses that connect transitions to assignments.

Now we describe how the formula F_k is constructed using the Reinforced encoding. The $\forall$-Step parallel planning semantics will be used again. As already stated, F_k will have the following three kinds of Boolean variables, which are the union of the variables used in the previous two encodings.

- *Action variables* a_i^t indicating whether the i-th action is used in the t-th step. We will have one such variable for each action from the description of the planning task and for each of the k parallel steps.

- *Assignment variables* $b^t_{x=v}$ indicating whether the value of the variable x is equal to v in the end of the t-th step (after applying the actions of the t-th step). This is different from the Direct encoding, where these variables described the beginning of the steps. We will have one such Boolean variable for each state variable $x \in X$ and each value $v \in \operatorname{dom}(x)$ for each of the t parallel steps.

- *Transition variables* c^t_δ (or $c^t_{x:\, d \to e}$ where $\delta = \delta_{x:\, d \to e}$) indicating whether the transition δ occurred during the t-th step. We will have one such variable for each $\delta \in \Delta$ for each of the k parallel steps.

Many of the clauses in the reinforced encoding are recycled from the previous encodings. In particular the following clauses are recycled.

- The initial state is encoded the same way as in the SASE encoding – using the unit clauses defined in equation 3.15 that disable all the transitions that are not compatible with the initial state.
- The clauses that ensure that each state variable has at most one value (equation 3.2) are used from the Direct encoding.
- Action interference clauses are the same as in both the Direct and SASE encodings (equation 3.7).
- The clauses enforcing that the actions imply their transitions (equation 3.12) are recycled from the SASE encoding.
- Similarly, the clauses representing the implications, that all the non prevailing transitions must be supported by their supporting actions (equation 3.13) are used from the SASE encoding.

Additionally, we need to add three new kinds of clauses that connect the assignment variables with the transition variables. The first set of clauses ensures that each transition $\delta_{x:\, d \to e}$ (including prevailing transitions $\delta_{x:\, e \to e}$ and mechanical transitions $\delta_{x:\, * \to e}$) implies that $x = e$ at the end of each step.

$$\begin{gathered}(\neg c^t_{\delta_{x:\, d \to e}} \vee b^t_{x=e}) \\ \forall \delta_{x:\, d \to e} \in \Delta, \forall t \in \{1, \ldots, k\}\end{gathered} \tag{3.17}$$

Similarly, we need to add clauses for each transition $\delta_{x:\, d \to e}$ (except for mechanical transitions) to enforce that $x = d$ holds at the end of the previous step, except for the first step, where we explicitly disable all the transitions that are not compatible with the initial state (using the clauses from equation 3.15).

$$\begin{gathered}(\neg c^t_{\delta_{x:\, d \to e}} \vee b^{t-1}_{x=d}) \\ \forall \delta_{x:\, d \to e} \in \Delta, d \neq *, \forall t \in \{2, \ldots, k\}\end{gathered} \tag{3.18}$$

The third kind of clauses is needed to guarantee, that if a variable x has the value v then there is a transition which changes the the value of x to v.

$$\begin{gathered}(\neg b^t_{x=v} \vee c^t_{\delta_1} \vee \cdots \vee c^t_{\delta_m}) \\ \forall x \in X, v \in \mathrm{dom}(x), \delta_1, \ldots, \delta_m \,\text{transform}\, x \,\text{to}\, v, \forall t \in \{1, \ldots, k\}\end{gathered} \tag{3.19}$$

Finally we add a set of clauses to ensure that the goal conditions will hold in the end. The goal conditions are encoded very similarly to the Direct encoding, i.e., using unit clauses with assignment variables (equation 3.9). The difference is that here we require these assignments in the end of the k-th step instead of the beginning of the $(k+1)$-th step.

$$\begin{gathered}(b^k_{x=v}) \\ \forall (x = v) \in s_G\end{gathered} \tag{3.20}$$

The formula F_k for the Reinforced encoding is a conjunction of the clauses defined in equations 3.2, 3.7, 3.12, 3.13, 3.15, 3.17, 3.18, 3.19, and 3.20. A $\forall$-step parallel plan can be extracted from any satisfying assignment of F_k exactly the same way as in the case of Direct and SASE encodings – see Definition 24. As we advertised in the beginning of this section, the clauses in equations 3.17, 3.18 and 3.2 allow us to get rid of the numerous transition interference clauses used in SASE (equation 3.11). Similarly to Proposition 2 and 3 the following Proposition holds for the Reinforced encoding.

Proposition 4. *If the formula F_k obtained using the Reinforced encoding of the planning task Π is satisfied by a truth assignment ϕ then P_ϕ is a valid $\forall$-Step parallel plan of makespan k for the planning task Π.*

Proof. This proof is again very similar to the proof of Proposition 2 and Proposition 3.

The requirements for the action sets given by the $\forall$-Step semantics are clearly satisfied:

- the preconditions of actions are satisfied due to 3.12, 3.17, and 3.18
- the effects are propagated also due to 3.12, 3.17, and 3.18
- the actions can be ordered arbitrarily for the same reasons as for the Direct and SASE encoding, since the clauses 3.7 are again included. (see the proof of Proposition 2).

It remains to prove that $P_\phi^S = [\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)]$ is a valid (sequential) plan for Π, where $\oplus$ denotes the concatenation of sequences and $\trianglelefteq$ is an arbitrary ordering of an action set.

Let us observe, that the transitions of the state variables are consistent at each step, i.e., exactly one transition is allowed for each state variable (due to 3.17, 3.18, and 3.2) and a transition cannot happen without an action that has it (thanks to 3.13). Furthermore, the transitions between the parallel steps must be compatible due to 3.17, 3.18, 3.19 and 3.2.

Since the action variables imply the proper transition variables thanks to 3.12, the actions must be applicable if their action variable is True and also the transition connected to the action must happen.

Thanks to 3.15 only transitions compatible with the initial state can happen in the first step and because of 3.18 and 3.20 only transitions that change the variables to their goal values are allowed in the last step. This fact together with the consistency of the transitions during all the k steps implies the validity of P_ϕ^S for the planning task Π. □

The reversed implication again does not hold for the same reasons as for the Direct and SASE encoding (see the discussion below Proposition 2).

3.4 The Exist Step Encoding

All the previous encodings of this chapter were connected to the $\forall$-Step parallel planning semantics. In this section we present a $R^2\exists$-Step encoding of planning into SAT, which was introduced in our recent paper [2]. Similarly to the previous sections, we will construct a formula F_k such that if F_k is satisfiable and ϕ is its satisfying assignment, then we can construct a $R^2\exists$-step parallel plan from ϕ.

In the encoding we will again explicitly represent which actions are in which step of the parallel plan as well as the world states in between the parallel steps. The first such state s_1 will represent the initial state, the second s_2 will represent the world state after applying the actions of the first step (in a proper order) and so on.

3.4.1 Variables and Plan Extraction

Similarly to the Reinforced encoding we have three kinds of Boolean variables.

- Action variables a_i^t for each action $a_i \in O$ and time step t. We will require that $a_i^t = True \Leftrightarrow a_i \in A_t$, i.e., a is in the parallel plan in step t. We have a set of such variables for each time step $t \in \{1 \ldots k\}$.
- Assignment variables $b_{x=v}^t$ for each state variable $x \in X$, and each $v \in \mathrm{dom}(x)$. We will require that $b_{x=v}^t = \mathrm{True} \Leftrightarrow s_t(x) = v$, i.e., in the state s_t at the beginning of time step t the variable x is assigned to the value v. We have these variables for each of the k parallel steps and one extra set for the time $k+1$ representing the goal state.
- Auxiliary variables $h_{i,j}^t$. More details about these variables will be provided later.

By checking the truth values of all the action variables in a satisfying assignment of the formula F_k, we can easily construct a parallel plan in the same way as we did for the Direct, SASE, and Reinforced encodings (see Definition 24).

To prove that such a plan is a valid $R^2\exists$-Step parallel plan we will also need to provide an ordering function $\trianglelefteq$ for the action sets. We will do this by assigning a unique rank $r(a)$ to each action $a \in O$ and the ordering function $\trianglelefteq$ will then simply sort the actions in the parallel steps according to their ranks. The ranks will be used during the construction of the formula to ensure the correctness of this method. The details of action ranking are discussed in the following subsection.

3.4.2 Action Ranking

The *ranking of actions* is the assignment of a unique integer rank $r(a)$ to each action $a \in O$. For the sake of correctness of the encoding the ranking can be arbitrary, however the selection of the ranking affects the performance of the planning process dramatically. Intuitively, the ranking should be such, that the actions are ranked according to their order in a valid plan for the given planning

task. The problem is, of course, that we do not know the plan in advance. Therefore, we should try to at least guess the order in which the given actions could appear in a plan.

Another problem is the evaluation of rankings, i.e., deciding which of a given set of rankings is better. We have been unable to identify an 'easy-to-check' property of a ranking which would indicate its quality. Having such a method would be very useful. We could generate several rankings and select the best one before encoding and solving the planning task. Currently, our only method of comparing different rankings is to use them in our $R^2\exists$-Step encoding and measure the time required to find a plan.

We have tested several ranking algorithms on the problems of the International Planning Competition, the details of the experiment and its results can be found in Subsection 3.7.2. We have compared the following ranking methods.

- *Random Ranking.* The ranks are assigned randomly to actions, each action has a unique rank. The purpose of this method is to serve as baseline for comparison with the other methods.
- *Input Ranking.* The ranks are assigned from 1 to $n = |O|$ to the actions in the order in which they are listed in the input, i.e., in the definition of the planning task. The intuition behind this methods is, that the author of the problem may have defined the actions in the order they are expected to appear in the plans.
- *Inverted Input ranking.* First we run Input Ranking, then we invert the ranks: $r(a) := n - r(a)$, where n is the number of actions. If Input Ranking is a 'good' ranking, then this ranking should be bad.
- *Topological Ranking.* This method is based on topologically ordering the action enabling graph while ignoring cycles. It is explained in detail below.
- *Inverted Topological Ranking.* We invert the result of Topological Ranking in the same way we invert Input Ranking.

The motivation behind Topological Ranking is that supporting actions should be before (have a lower rank than) the actions they support. We are trying to achieve this by topologically sorting the enabling graph of the set of actions. First, let us define the enabling graph.

Definition 26 (Enabling Graph)**.** *The* enabling graph *G for a set of actions O is a directed graph where vertices represent actions and there is an edge (a, a') if a supports a', i.e., $G = (O, \{a \to a' | a, a' \in O; \mathrm{eff}(a) \cap \mathrm{pre}(a') \neq \emptyset\})$.*

An example of an enabling graph is displayed in Figure 3.2. The $\exists$-Step [33] and the Relaxed $\exists$-Step [38] encodings use similar graphs called disabling and disabling-enabling graphs.

As apparent from the example, the enabling graph may contain cycles and therefore topological ordering is not defined. We break the cycles by randomly

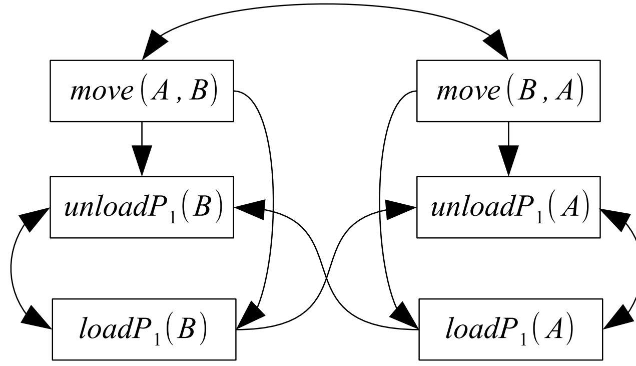


Figure 3.2: The enabling graph for a subset of actions of the planning task from Example 2. We selected the actions for the locations A and B and package P_1.

```
    topologicalRanking(O)
T1    global lastRank := 0
T2    global visited := {False, ..., False}
T3    foreach a ∈ O do
T4      rankAction(a)

    rankAction(a)
R1    if visited[a] = False then
R2      visited[a] := True
R3      foreach s ∈ supportingActions(a) do
R4        rankAction(s)
R5      r(a) := lastRank
R6      lastRank := lastRank + 1
```

Figure 3.3: Pseudo-code of the topological sorting based action ranking algorithm.

removing edges and use the topological ordering of the reduced enabling graph to assign action ranks. This is implemented by modifying the depth-first-search topological sorting algorithm [37] to ignore cycles. The pseudo-code of the topological ranking algorithm is displayed in Figure 3.3. Note, that the enabling graph is constructed 'on-the-fly' during the procedure when the edges are needed.

The weakness of this ranking method is that only the actions of a planning task are used. For example the initial state and the goal conditions are not considered at all. Designing better ranking algorithms is a matter of future work together with methods for comparing rankings.

3.4.3 Basic Clauses

Here we start defining the clauses that are contained in F_k. Many of the clauses used here are the same as in the Direct (and Reinforced) encoding. In particular, the following clauses are reused.

- The unit clauses for the initial state (equation 3.8).
- The unit clauses for the the goal conditions (equation 3.9).
- The binary clauses enforcing the uniqueness of variable values (equation 3.2).
- The clauses ensuring that the values of the variables do not change, unless some action changes them (equation 3.5).

Next we will describe the clauses connecting the action variables to assignments. In the previous encodings we required that actions imply their preconditions. This is also used in the ∃-Step encodings [33]. In our (and also in the Relaxed-∃-Step [38]) encoding we require that the preconditions of actions are either satisfied by the proper assignment or by some other action in the same time step. If we encoded this idea in a straightforward way it would cause trouble, since the actions could support each other in a cyclic manner with disregard to the initial state of the time step. The solution is to use the action ranks ($r(a)$). Then we can encode our relaxed condition by clauses requiring that each precondition of an action is either satisfied by the assignment from the beginning of this parallel step or by an action with a lower rank in this parallel step. The action-precondition clauses will be the following.

$$\begin{gathered}(\neg a_i^t \vee b_{x=v}^t \bigvee \{a_j^t \mid r(a_j) < r(a_i), (x=v) \in \text{eff}(a_j)\}) \\ \forall a_i \in O, \forall (x=v) \in \text{pre}(a_i), \forall t \in \{1 \ldots k\}\end{gathered} \tag{3.21}$$

The ranking solves the problem of cyclic supporting, however there still remains one problem. It may happen that an action is relying on another action (with a lower rank) which sets its precondition but there may be another action between them which destroys the precondition. For example, if we have three actions $r(\mathit{move}(A,B)) < r(\mathit{move}(B,A)) < r(\mathit{loadP}_1(B))$ (from Example 2) and $\mathit{loadP}_1(B)$ relies on $\mathit{move}(A,B)$ then $\mathit{move}(B,A)$ must not be in the plan. Actions like $\mathit{move}(B,A)$ are referred to as *threats* in Plan Space Planning [22]. Dealing efficiently with threats is a bit more complicated and we will address it separately in the next subsection.

Next we describe the clauses that distinguish our R^2∃-Step encoding from the Relaxed ∃-Step encoding and ensure that the effects of the actions are properly propagated. The idea is similar as in the case of preconditions. We will add clauses that ensure, that each effect $(x = v)$ of an action is either projected to the next parallel step or there is an action with a higher rank in this step, that will take the responsibility of setting the value of the variable x for the next parallel step.

$$\begin{gathered}(\neg a_i^t \vee b_{x=v}^{t+1} \bigvee \{a_j^t \mid r(a_j) > r(a_i), x \in \text{scope}(a_j)\}) \\ \forall a_i \in O, \forall (x=v) \in \text{effprev}(a_i), \forall t \in \{1 \ldots k\}\end{gathered} \tag{3.22}$$

By effprev(a) we mean the set of effects and prevailing assignments of a. *Prevailing assignments* are such precondition assignments that have no matching effect

assignment, e.g., an assignment to the same variable. For example the action $(\{x_1 = 1, x_2 = 1\}, \{x_2 = 3\})$ has a prevailing assignment $x_1 = 1$. A prevailing condition must be treated like an effect, otherwise the value of its variable might not propagate to the next parallel step which could lead to invalid plans. By $\text{scope}(a)$ we mean the set of state variables that appear in $\text{effprev}(a)$.

3.4.4 Action Interference Clauses

The last and the most complicated set of clauses we need to add to our formula are the clauses solving the above mentioned problem of lower ranking actions destroying the preconditions of higher ranking actions in the same parallel step. We start by some notation regarding the relationships between actions and assignments that will simplify the following descriptions.

The relationship status of an action a and an assignment $x = v$ is one or more of the following

- a is *supporting* $(x = v)$ if $(x = v) \in \text{eff}(a)$
- a is *opposing* $(x = v)$ if $\exists v' \neq v : (x = v') \in \text{eff}(a)$
- a is *requiring* $(x = v)$ if $(x = v) \in \text{pre}(a)$
- a is *unrelated* to $(x = v)$ if $(x = v) \notin \text{pre}(a) \cup \text{eff}(a)$

Note that an action can be both requiring and opposing an assignment (that precondition would be non-prevailing).

We want to ensure that if an opposing action a_1 destroys an assignment then there is no action with a higher rank a_2 requiring it unless there is some supporting action a_3 between a_1 and a_2 that sets it up again. But then again there might be an action between a_3 and a_2 that destroys the assignment again. The situation seems to be quite intricate but fortunately there is an elegant solution. The solution is inspired by the $\exists$-Step encoding [33], particularly their encoding of action interference constraints (also used in the Relaxed $\exists$-Step encoding [38]).

In [33] and [38] the authors want to ensure that if an action destroys an assignment then no other action with a higher rank requiring that assignment can be in the same parallel step. Using auxiliary variables they build a chain of implications for every possible assignment over the opposing and requiring actions of that assignment. A formal description of such a chain for one assignment follows.

Let $a_1, a_2, \ldots, a_k$ be a list of all n opposing and m requiring actions of a given assignment sorted by the ranks of the actions. Let $o_1, \ldots o_n$ be the indices of the opposing actions and $r_1, \ldots, r_m$ the indices of requiring actions ($k \leq m + n$). Let $\text{next}(i) > i$ be the index of the closest requiring action after the i-th action in the sequence $a_1, a_2, \ldots, a_k$. Let $h_{r_1}, h_{r_2}, \ldots, h_{r_m}$ be new auxiliary variables. The chain is composed of the following implications.

- $h_j \Rightarrow h_{\text{next}(j)}; \forall j \in \{r_1, \ldots, r_{m-1}\}$

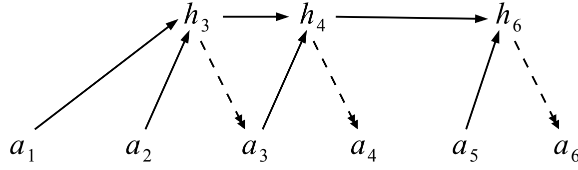


Figure 3.4: A graphic representation of the implications chain. In the example on the picture a_1, a_2, a_3, and a_5 are opponents of the given assignment and a_3, a_4, and a_6 require it. Notice that a_3 both requires and opposes the assignment. The solid lines represent the implications and the dashed lines negative implications.

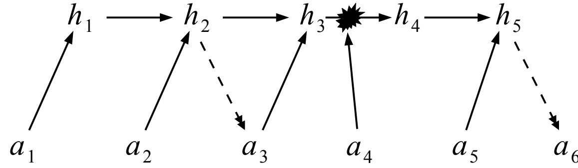


Figure 3.5: A graphic representation of the extended implications chain. In the example on the picture a_1, a_2, a_3, and a_5 are opponents of the given assignment, a_4 is a supporter and a_3, and a_6 require it. Being a supporter, a_4 can break the chain between h_3 and h_4. The solid lines represent the implications, the dashed lines negative implications and the explosion denotes the chain breaking.

- $h_j \Rightarrow \neg a_j; \forall j \in \{r_1, \ldots, r_m\}$
- $a_j \Rightarrow h_{\text{next}(j)}; \forall j \in \{o_1, \ldots, o_n\}$

See Figure 3.4 for a graphic representation of an example. It is easy to see that such a chain of implications will ensure the required constraints. Whenever an opposing action gets selected into the plan it blocks all the actions with a higher rank that require the given assignment.

We will extend the chain definition to accommodate the situation where an intermediate action can restore the destroyed assignment. First we need to add the supporting actions into the chain and then relax the corresponding $h_j \Rightarrow h_{\text{next}(j)}$ implications to allow the supporting actions to break the chain. A more formal description follows.

Let $a_1, a_2, \ldots, a_k$ be a list of all n opposing, m requiring, and l supporting actions of a given assignment sorted by the ranks of the actions. Let $o_1, \ldots o_n$ be the indices of the opposing actions, $r_1, \ldots, r_m$ the indices of requiring actions, and $s_1, \ldots, s_l$ the indices of supporting actions. Let $h_1, h_2, \ldots, h_{k-1}$ be new auxiliary variables. The chain is composed of the following implications.

- $h_{j-1} \Rightarrow h_j; \forall j \in \{2, \ldots, k-1\} \setminus \{s_1, \ldots, s_l\}$
- $h_{j-1} \Rightarrow \neg a_j; \forall j > 1; j \in \{r_1, \ldots, r_m\}$
- $a_j \Rightarrow h_j; \forall j \in \{o_1, \ldots, o_n\}$

- $h_{j-1} \Rightarrow (h_j \vee a_j); \forall j > 1; j \in \{s_1, \dots, s_l\}$

An example with a graphic representation of the extended implication chain is given in Figure 3.5. The extended chain is very similar to the original one and it is easy to verify from the definition of the implications that a supporting action can indeed break the chain of implications which would prevent an action whose precondition has been restored. Next we give the clausal representation of the implications above.

$$\begin{aligned}(\neg h^t_{x=v,\text{prevN}(i,x,v)} \vee h^t_{x=v,i}); \forall i \in (R_{x=v} \cup O_{x=v})\\ (\neg h^t_{x=v,\text{prevN}(i,x,v)} \vee \neg a^t_i); \forall i \in R_{x=v}\\ (\neg a^t_i \vee h^t_{x=v,i}); \forall i \in O_{x=v}\\ (\neg h^t_{x=v,\text{prevN}(i,x,v)} \vee h^t_{x=v,i} \vee a^t_i); \forall i \in S_{x=v}\\ \forall x \in X, \forall v \in \text{dom}(x), \forall t \in \{1 \dots k\}\end{aligned} \tag{3.23}$$

Where

- $h^t_{x=v,i}$ are new auxiliary Boolean variables, i represents action indices.
- $R_{x=v}$ is the set of indices of actions that require $x = v$
- $O_{x=v}$ is the set of indices of actions that oppose $x = v$
- $S_{x=v}$ is the set of indices of actions that support $x = v$
- $\text{prevN}(i, x, v)$ is the index of the action preceding a_i in $N_{x=v}$, where $N_{x=v}$ is the sorted list (by action ranks) of all supporting, opposing and requiring actions of $x = v$.

The encoding of the chain can be improved by removing some of the auxiliary variables and changing the related implications accordingly.

This concludes the description of our encoding. The final formula F_k is a conjunction of all the clauses defined in equations 3.2, 3.5, 3.8, 3.9, 3.21, 3.22, and 3.23.

3.4.5 Correctness

Following up on Propositions 2, 3, and 4 we now present a similar proposition for the $R^2\exists$-Step encoding.

Proposition 5. *Let F_k be a $R^2\exists$-Step encoded formula for a planning task $\Pi = (X, O, s_I, s_G)$. If F_k is satisfiable then a parallel plan of length k can constructed from the satisfying assignment of F_k.*

Proof. Let $P_\phi = [A_1, \dots, A_k]$ be a sequence of actions sets extracted from the satisfying assignment of F_k as defined in Definition 24. Let $\trianglelefteq$ be an ordering function, which orders the actions in A in the increasing order of their ranks. It is enough to prove that each $\trianglelefteq(A_i)$; $i \in \{1, \dots, k\}$ is a valid sequential plan for a planning task (X, O, s_i, s_{i+1}), where $s_0 = s_I$ and $s_{i+1} = \text{apply}(\trianglelefteq(A_i), s_i)$. Then $\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)$ is a valid sequential plan for the task Π since $s_0 = s_I$ (thanks to the clauses 3.8 and 3.2), $s_{k+1} \subseteq s_G$ (thanks to the clauses 3.9 and 3.2), and the values of state variables are correctly propagated between the end of A_i and the beginning of A_{i+1} (thanks to 3.5 and 3.22).

We are going to prove the validity of $\trianglelefteq(A_i); i \in \{1, \dots, k\}$ for the task (X, O, s_i, s_{i+1}) by contradiction. Let us assume that $\trianglelefteq(A_i)$ is an invalid plan. Then either the goal conditions are not satisfied or there is at least one action in $\trianglelefteq(A_i)$ that cannot be applied because of unsatisfied precondition(s) in the given state.

However, the goal conditions must be satisfied because of the clauses defined in 3.2, 3.22 and 3.5 – they are either copied from the state s_i or set by an action in A_i (thanks to 3.22 and 3.5) and no action later in $\trianglelefteq(A_i)$ (with a higher rank) could set them to another value (thanks to 3.2 and 3.22).

A precondition of an action a in $\trianglelefteq(A_i)$ also cannot be unsatisfied, since thanks to 3.21 they are either satisfied by the previous world state or another action a' in $\trianglelefteq(A_i)$ which is before a (because of a lower rank) and thanks to 3.23 the precondition is not destroyed by other action between a' and a without being restored by some other action.

Therefore $\trianglelefteq(A_i); i \in \{1, \dots, k\}$ must be valid sequential plans for the planning tasks (X, O, s_i, s_{i+1}) and $\trianglelefteq(A_1) \oplus \cdots \oplus \trianglelefteq(A_k)$ a valid sequential plan for the planning task $\Pi = (X, O, s_I, s_G)$. □

The implication, that if there is an $R^2\exists$-Step parallel plan P_k of makespan k then the formula F_k constructed as described above is satisfiable, does not hold. The reason is that we would need to know the order of the actions in P_k and use it to rank the actions properly for the construction of F_k. Since we cannot guarantee that we will rank the actions properly, our F_k might be unsatisfiable even if P_k is a valid $R^2\exists$-Step parallel plan of makespan k.

However, in general, if there exists a plan for a given planning task, then we will eventually find it. Regardless of the ranking of actions there can always be at least one action in each parallel step. Therefore, if there is a sequential plan of length k for a given planning task, then the formula F_k constructed using our encoding with any action ranking will be satisfiable.

3.5 Selective Encoding

While doing experiments with our new encodings we noticed, that on some problems the $R^2\exists$-Step encoding works very well and significantly outperforms the other encodings. But there are also problems, where the situation is reversed, i.e., the $R^2\exists$-Step encoding performs very poorly.

Our goal is, of course, to have an encoding that works well for all the problems, or at least for as many as possible. This can be achieved by combining two or more different encodings into one to form a selective encoding.

A *selective encoding* consists of a set of encodings E and a *selection rule*. The task of the selection rule is to select an encoding from E that should be used to solve a given planning task based on the planning task itself and other available information (for example the makespan of the formula that we want to construct). A good selection rule should be simple, so it be can evaluated quickly and clever, so it can select the proper encoding for a planning task.

Based on experimental observations we have constructed the following selective encoding which uses our two new encodings – the Reinforced and $R^2\exists$-Step encoding. The selection rule is based on our observation, that the $R^2\exists$-Step encoding performs best, when the number of transitions is low. The selection rule decides based on the number of transitions ($|\Delta|$) divided by the number of state variables ($|X|$) in the given planning task. If $|\Delta|/|X| > 10$ then the Reinforced encoding is selected, otherwise we select the $R^2\exists$-Step encoding. Furthermore, when the $R^2\exists$-Step Encoding is used, we alternate between two action ranking algorithms (see Subsection 3.4.2). For even makespans we use the Topological Ranking method (see Figure 3.3) and for the odd makespans the Input Ranking method (rank the actions in the order they are defined in the input problem). The proposed rule can be evaluated very quickly and, as the experiments will demonstrate, works very well on the available benchmark problems.

3.6 Properties of the Encoded Formulas

In this section we will examine the formulas F_k constructed by the four encodings described in the previous sections. We will compute upper bounds on the number of their variables and clauses. We will differentiate between unit (1 literal), binary (2 literals), and Horn (at most one positive literal) clauses.

3.6.1 Planning Task Parameters

The size of the formulas will of course depend on the parameters of the planning task being encoded. We will use the following quantitative properties of a planning task $\Pi = (X, O, s_I, s_G)$ to compute the upper bounds.

- n - The number of actions ($n = |O|$).
- v - The number of state variables ($v = |X|$).
- d - The maximum domain size ($d = \max_{x \in X}\{|\operatorname{dom}(x)|\}$).
- p - The maximum number of preconditions or effects an action has ($p = \max_{a \in O}\{|\operatorname{pre}(a)|, |\operatorname{eff}(a)|\}$)

Typically, the number of actions is much higher than the other parameters. From these values we can compute the following upper bounds related to the planning task.

- The number of assignments is at most vd.
- The number of transitions is at most $v(d^2 + d)$ since there are at most $d(d-1)$ active, d prevailing, and d mechanical transitions for each variable.
- The number of auxiliary nodes in the implication chains used by the $R^2\exists$-Step encoding is at most vdn since we have a chain for each assignment and the total number of all supporting, opposing, and requiring actions of an assignment is at most n.

3.6.2 Number of Variables

The following table contains an overview of the maximum number of Boolean variables in F_k (a formula for makespan k) for the four described encodings.

Var. Type	Direct	SASE	Reinforced	$R^2\exists$-Step
action	kn	kn	kn	kn
assignment	$(k+1)vd$	-	kvd	$(k+1)vd$
transition	-	$kv(d^2+d)$	$kv(d^2+d)$	-
chain	-	-	-	$kvdn$

The values correspond to the maximum number of actions, assignments, transitions and implication chain nodes multiplied by k, except for the Direct and $R^2\exists$-Step encodings, which have one extra set of assignment variables.

If we assume, that $n > d$, then the maximum number of variables is increasing in the following order: Direct, SASE, Reinforced, and $R^2\exists$-Step.

3.6.3 Number of Clauses

We will examine the four encodings one by one and count the number of clauses in them by their type.

The formula F_k obtained by the Direct encoding is the conjunction of the clauses defined in equations 3.1, 3.2, 3.3, 3.4, 3.5, 3.7, 3.8, and 3.9.

- There are $(k+1)v$ clauses of the type 3.1 – $k+1$ sets for for each variable.
- There are at most $(k+1)vd^2$ clauses of the type 3.2 – $k+1$ sets for each variable and two different values from its domain. These clauses are binary and Horn.
- There are at most knp clauses of both type 3.3 and type 3.4 – one for each step, action and each of its preconditions / effects. These clauses are binary and Horn.

- There are at most kvd clauses of the type 3.5 – one for each step, variable and a value from its domain.
- There are at most kn^2 clauses of the type 3.7 – one for each step, and each pair of compatible interfering actions (at most each pair of actions). These clauses are binary and Horn.
- There are at most v clauses of both type 3.8 and type 3.9 – one for each variable value in the initial state / goal condition. These clauses are unit.

In total we have $k(n^2 + 2np + vd^2 + vd + v) + vd^2 + 3v$ clauses, from which $2v$ are unit clauses and $k(n^2 + 2np + vd^2) + vd^2$ are both binary and Horn clauses.

The formula F_k obtained by the SASE encoding is the conjunction of the clauses defined in equations 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, and 3.7.

- There are kv clauses of the type 3.10 – one for each variable and step.
- There are at most $kv(d^4 + 2d^3 + d^2)$ clauses of the type 3.11 – one for each step and variable and all pairs of transitions related to this variable ($(d^2 + d)^2$ pairs since there are at most $d^2 + d$ transitions for each variable). These clauses are binary and Horn.
- There are at most $2knp$ clauses of the type 3.12 – one for each step, action and each of its transitions (there are at most $2p$ transitions connected to each action). These clauses are binary and Horn.
- There are at most $kv(d^2 + d)$ clauses of both type 3.13 and type 3.14 – one for each step and transition.
- There are at most $v(d^2+d)$ clauses of both type 3.15 and type 3.16 – one for each transition that is not compatible with the initial state / goal conditions (at most all the transitions). These are unit clauses.
- There are at most kn^2 clauses of the type 3.7 – one for each step, and each pair of compatible interfering actions (at most each pair of actions). These clauses are binary and Horn.

In total we have $k(n^2 + 2np + vd^4 + 2vd^3 + 3vd^2 + 2vd + v) + 2vd^2 + 2vd$ clauses, from which $2vd^2 + 2vd$ are unit clauses and $k(n^2 + 2np + vd^4 + 2vd^3 + vd^2)$ are both binary and Horn clauses.

The formula F_k obtained by the Reinforced encoding is the conjunction of the clauses defined in equations 3.2, 3.7, 3.12, 3.13, 3.15, 3.17, 3.18, 3.19, and 3.20.

- There are at most kvd^2 clauses of the type 3.2 – one for each step and variable and two different values from its domain. These clauses are binary and Horn.
- There are at most kn^2 clauses of the type 3.7 – one for each step, and each pair of compatible interfering actions (at most each pair of actions). These clauses are binary and Horn.

- There are at most $2knp$ clauses of the type 3.12 – one for each step, action and each of its transitions (there are at most $2p$ transitions connected to each action). These clauses are binary and Horn.
- There are at most $kv(d^2 + d)$ clauses of the type 3.13 – one for each step and transition.
- There are at most $v(d^2 + d)$ clauses of type 3.15 – one for each transition that is not compatible with the initial state (at most all the transitions). These are unit clauses.
- There are at most $kv(d^2 + d)$ clauses of the both type 3.17 and type 3.18 – one for each step and transition. These clauses are binary and Horn.
- There are at most kvd clauses of the type 3.19 – one for each step and assignment.
- There are at most v clauses of the type 3.20 – one for each goal condition. These clauses are unit.

In total we have $k(n^2 + 2np + 4vd^2 + 4vd) + vd^2 + vd + v$ clauses, from which $vd^2 + vd + v$ are unit clauses and $k(n^2 + 2np + 3vd^2 + 2vd)$ are both binary and Horn clauses.

Finally, the formula F_k obtained by the $R^2\exists$-Step encoding is the conjunction of the clauses defined in equations 3.2, 3.5, 3.8, 3.9, 3.21, 3.22, and 3.23.

- There are at most $(k + 1)vd^2$ clauses of the type 3.2 – $k + 1$ sets for each variable and two different values from its domain. These clauses are binary and Horn.
- There are at most kvd clauses of the type 3.5 – one for each step, variable and a value from its domain.
- There are v clauses of both type 3.8 and type 3.9 – one for each variable value in the initial state / goal condition. These clauses are unit.
- There are $2knp$ clauses of both type 3.21 and type 3.22 – one for each step, action and its precondition / effect / prevailing conditions (at most $2p$ in total).
- There are $4knvd$ clauses of the types defined in Equation 3.23. This reflects the worst case that each action requires, supports and also opposes each assignments. Since an action cannot both support and oppose an assignment this is not a tight upper bound. Most of these clauses ($3knvd$) are binary and Horn.

In total we have $k(4nvd + 2np + vd^2 + vd) + vd^2 + 2v$ clauses, from which $2v$ are unit clauses and $k(3nvd + vd^2) + vd^2$ are both binary and Horn clauses.

The following table contains a summary of the number of clauses.

Encoding	Number of Clauses
Direct	$k(n^2 + 2np + vd^2 + vd + v) + vd^2 + 3v$
SASE	$k(n^2 + 2np + vd^4 + 2vd^3 + 3vd^2 + 2vd + v) + 2vd^2 + 2vd$
Reinforced	$k(n^2 + 2np + 4vd^2 + 4vd) + vd^2 + vd + v$
$R^2\exists$-Step	$k(4nvd + 2np + vd^2 + vd) + vd^2 + 2v$

Bear in mind, that these are all upper bounds for the worst-case scenario and the number of actual clauses is usually much lower. The size of the $R^2\exists$-Step formula will also depend on the action ranking. Some of the clauses which we did not count as binary or Horn can be binary or Horn for a concrete planning task. For example if an assignment has only one supporting action, then the related clause defined in Equation 3.13 will be binary. We also did experimental measurements of the formula sizes. For the results refer to Subsection 3.7.4.

3.7 Experiments

To compare the encodings to each other and to other state-of-the-art encodings we did experiments using all the benchmark problems from the optimization track of the 2011 International Planning Competition (IPC) [16]. We measured the time required to solve the instances, the number of SAT solver calls (makespans), and the number of problems solved within a given time limit. We also investigated the size and composition of the encoded formulas.

First we compared five different ranking methods for our $R^2\exists$-Step encoding with a 10 minutes runtime limit. Next, we run experiments with a 30 minutes time limit using the following six encoding implementations.

- Direct Encoding (Dir). We implemented a Java encoder to encode the planning task using the Direct encoding as described in Section 3.1
- SASE Encoding (SASE). Our Java encoder for the modified SASE encoding described in Section 3.2
- Reinforced Encoding (Reinf). The Java implementation of our new Reinforced encoding from Section 3.3
- $R^2\exists$-Step Encoding ($R^2\exists$). The implementation for our new $R^2\exists$-Step encoding in Java as described in Section 3.4. We have used the topological sorting based action ranking method (see Figure 3.3).
- Selective Encoding (Sel). This encoding is a combination of the two previous encodings – the Reinforced and the $R^2\exists$-Step as defined in Section 3.5.
- Rintanen's $\forall$-Step Encoding (R$\forall$). An efficient C++ implementation of the $\forall$-Step semantics by Rintanen [33] used in his well known state-of-the-art SAT based planning system Madagascar [32].
- Rintanen's $\exists$-Step Encoding (R$\exists$). An efficient C++ implementation of the $\exists$-Step semantics encoding by Rintanen [33] also used in Madagascar [32].

We were unable to do experiments with the Relaxed ∃-Step encoding [38], since there is no implementation compatible with the benchmark problems of the 2011 IPC.

3.7.1 Experimental Setting

To compare the performance of the encodings we created a simple script, which iteratively constructed and solved the formulas for time steps $1, 2, \ldots$ until a satisfiable formula was reached (see Figure 3.1). For each encoding we used the same SAT solver – Lingeling[9] (version ats).

The time limit was 5 or 30 minutes for the SAT solving part, i.e., the total time the SAT solver could spend solving the formulas $F_1, F_2, \ldots$ for each problem instance was 5 or 30 minutes. The time required for the generation of $F_1, F_2, \ldots$ is ignored. Hence the overall planning time could exceed the time limit for a problem instance.

The experiments were run on a computer with Intel i7 920 CPU @ 2.67 GHz processor and 6 GB of memory. As already mentioned, our five encoding procedures were implemented in Java. To obtain the state-of-the-art ∀-Step formulas and ∃-Step formulas we used Rintanen's planner Madagascar[32] (version 0.99999 21/11/2013 11:54:15 amd64 1-core).

The benchmark problems of the IPC are organized into domains. Each domain contains 20 problems and there are 14 domains which results in a total of 280 problems. The benchmark problems are provided in the PDDL format which is accepted by Madagascar, however our encodings require input in the SAS+ format. We used Helmert's translation tool, which is a part of the Fast Downward planning system [23], to obtain the SAS+ files from the PDDL files. The translation is very fast requiring only a few seconds for all domains.

3.7.2 Ranking Comparison

Before presenting the results of the main experiment let us look at the comparison of the ranking methods described in Subsection 3.4.2. We have run the SAT planning algorithm with our R^2∃-Step encoding using the five different ranking methods from Subsection 3.4.2 with a time limit of five minutes for SAT solving.

The number of solved problems and the total makespan of the found plans is presented in Table 3.1. The total SAT solving time for the solved problems is in Table 3.2.

The Topological Ranking (TSort) method solved the highest number of problems while the Inverted Input and Random methods were the least successful. Nevertheless, there are five domains, where TSort was outperformed. Most notable among these is the openstacks domain, where the Input ranking solved almost twice as many problems while its total makespan and also its runtime remained low.

Focusing on the makespans, we can observe, that in several cases TSort maintains the smallest total makespan and total runtime while solving as many or more

Table 3.1: The number of solved problems (#P) with their total makespan (Mks) that were solved under 5 minutes by our $R^2\exists$-Step Encoding using different action ranking methods.

Domain	TSort		TSort^{-1}		Input		Input^{-1}		Random	
	#P	Mks	#P	Mks	#P	Mks	#P	Mks	#P	Mks
barman	**4**	36	2	29	**4**	60	2	28	1	11
elevators	**20**	85	**20**	99	**20**	106	**20**	79	**20**	75
floortile	17	158	**18**	185	16	149	**18**	178	**18**	167
nomystery	3	14	4	20	3	13	**6**	33	3	14
openstacks	12	75	13	66	**20**	59	5	43	10	57
parcprinter	**20**	30	**20**	249	**20**	88	**20**	186	**20**	140
parking	0	0	0	0	0	0	0	0	0	0
pegsol	**19**	158	18	155	12	147	16	142	18	152
scanalyzer	6	11	**9**	16	7	12	6	13	6	12
sokoban	**1**	17	**1**	19	**1**	18	**1**	17	**1**	19
tidybot	**1**	1	**1**	1	**1**	1	**1**	1	**1**	1
transport	5	20	6	40	8	44	**9**	57	4	19
visitall	**20**	34	12	113	9	55	9	49	12	80
woodworking	**20**	33	**20**	57	**20**	58	**20**	30	**20**	40
Total	**148**		144		141		133		134	

problems than the other methods. This is most apparent for the parcprinter and visitall domains.

Low total makespan does not always correlate with good runtime. For example in the elevators and woodworking domains, where all the methods solved all the problems, we can observe, that TSort has a low total makespan, but high runtime compared to the other methods.

Overall, we can conclude, that the action ranking affects the performance of planning very significantly and different ranking methods work well for different domains. Nevertheless, TSort appears to be the best choice in general and therefore we will use it when comparing the $R^2\exists$-Step encoding to other encodings.

3.7.3 Performance Results

In this subsection we analyze the results of the main experiment – the comparison of the seven encoding methods on the IPC problems with a 30 minutes time limit for each problem.

The number of solved instances in presented in Table 3.3. Looking at the results from the perspective of the domains, we can observe, that the elevators, parcrpinter, and woodworking domains are entirely solved by every encoding. On

Table 3.2: The total SAT solving time required to solve the problems that were solved under 5 minutes by our $R^2\exists$-Step Encoding using different action ranking methods.

Domain	TSort	TSort^{-1}	Input	Input^{-1}	Random
barman	641.64	1229.25	934.80	577.67	276.00
elevators	273.74	134.15	68.28	70.86	217.31
floortile	984.49	1259.40	1404.30	1380.39	1189.03
nomystery	293.92	954.26	278.33	409.84	173.22
openstacks	791.33	1417.90	440.23	499.20	508.37
parcprinter	4.35	54.97	7.44	59.44	28.42
parking	0	0	0	0	0
pegsol	817.88	1356.88	1632.52	837.86	937.63
scanalyzer	228.57	1390.03	262.36	151.30	274.30
sokoban	49.69	74.21	52.29	51.76	68.33
tidybot	1.63	1.92	1.42	1.73	2.13
transport	282.67	2377.54	656.24	912.33	300.06
visitall	13.98	564.88	23.21	23.58	255.73
woodworking	137.63	84.57	95.91	51.88	152.27

the other hand, the parking domain is so difficult that not even a single problem is solved by any of the encodings. The openstacks domain is also very difficult for all but our new $R^2\exists$-Step encoding (and Selective, which also uses it). The sokoban and tidybot domains are very hard for all of our encodings, while the encodings of Rintanen can handle them much better, especially the tidybot domain.

If we compare the encodings, we can observe that the best results are obtained by the Selective encoding followed the two Rintanen encodings. The Selective encoding very successfully combines the Reinforced and $R^2\exists$-Step encodings and is never worse for any domain than any of its components. The strength of the Selective encoding is that it can properly select the Reinforced encoding for the domains and problems, where the $R^2\exists$-Step is not efficient. Also its ability to use two action ranking methods instead of just one seems to be very helpful.

The Reinforced encoding itself succeeds in its goal to combine the strengths of the Direct and SASE encodings. Except for the visitall domain it is newer worse than the Direct or SASE encoding.

Comparing the two encodings of Rintanen we can notice, that they do not differ too much from each other. In contrast, the variability of our encodings appears to be much higher. We consider higher variability an advantage, since it allows for techniques such as our Selective encoding to be more successful.

The times required to solve the problems are presented in Table 3.4. If we look at the results for the domains, where each encoding solved all the problems, i.e.,

Table 3.3: The number of problems in each domain that the encodings solved within the time limit (30 minutes for SAT solving).

Domain	Dir	SASE	Reinf	$R^2\exists$	Sel	R∀	R∃
barman	4	4	4	8	**9**	8	4
elevators	**20**	**20**	**20**	**20**	**20**	**20**	**20**
floortile	16	11	18	18	18	16	**20**
nomystery	**20**	10	**20**	6	**20**	**20**	**20**
openstacks	0	0	0	15	**20**	0	0
parcprinter	**20**	**20**	**20**	**20**	**20**	**20**	**20**
parking	0	0	0	0	0	0	0
pegsol	10	6	10	**19**	**19**	11	12
scanalyzer	14	12	15	9	15	17	**18**
sokoban	2	2	2	2	2	**6**	**6**
tidybot	2	2	2	2	2	13	**15**
transport	16	17	18	13	**19**	18	18
visitall	12	9	10	**20**	**20**	11	11
woodworking	**20**	**20**	**20**	**20**	**20**	**20**	**20**
Total	156	133	159	172	**204**	180	184

the elevators, parcprinter, and woodworking, we can notice, that except for the parcprinter problems, the runtime of the $R^2\exists$-Step encoding is much higher than the runtime of the other methods. If we also look at Table 3.5, which contains the total makespan of the found plans, we can deduce, that lower makespan, i.e., fewer SAT calls does not necessarily mean faster planning, especially not in the case of these easy domains. Nevertheless, for the domains, where $R^2\exists$-Step significantly outperformed the other methods (except for Selective) – openstacks, pegsol, and visitall, the makespans are much lower than the makespans of the other methods, despite the fact, that they solved fewer problems.

In Figure 3.6 we provide a so called cactus plot, that visually compares the five strongest encodings. The plot represents how many problems can be solved withing a given time limit. It is interesting, that the lines for the $R^2\exists$-Step and the 'Rintanen ∀' encoding cross each other several times, which means, that for certain time limits $R^2\exists$-Step would solve more instances than the 'Rintanen ∀' encoding. The cactus plot confirms, that the Selective method significantly outperforms all the other methods, furthermore, this holds for any reasonable time limit.

Table 3.4: The time in seconds required to solve all the problems that were solved within the time limit. The presented time is the sum of times the SAT solver alone required, formula generation time is not included.

Domain	Dir	SASE	Reinf	$R^2\exists$	Sel	R∀	R∃
barman	2041.44	1549.31	1680.53	2999.30	3562.01	4368.98	693.80
elevators	33.96	55.72	38.10	288.29	39.63	29.76	5.73
floortile	7327.15	2083.57	1206.18	1380.63	1070.39	2243.53	154.85
nomystery	3798.45	1377.23	1894.65	1927.40	1669.42	5104.46	494.10
openstacks	-	-	-	2679.68	440.12	-	-
parcprinter	10.07	28.25	12.15	4.24	6.12	5.32	5.51
parking	-	-	-	-	-	-	-
pegsol	3796.11	2096.72	4971.97	830.72	824.83	5365.69	6965.38
scanalyzer	1401.21	831.80	1435.19	1118.89	938.14	575.33	843.79
sokoban	592.03	1337.87	857.04	550.18	441.44	4373.59	4255.80
tidybot	75.68	74.02	118.09	480.85	759.08	2953.78	4254.63
transport	1404.23	3554.90	3203.62	7418.01	4004.49	3373.99	366.83
visitall	2380.76	683.25	726.53	14.24	14.45	1606.21	1702.49
woodworking	2.57	2.04	3.84	138.61	180.07	0.66	0.77

3.7.4 Properties of the Formulas

In Section 3.6 we calculated upper bounds on the number of variables and clauses in the encoded formulas. In this section we will examine the properties of the formulas generated for the IPC benchmark problems by our four encodings as well as by the two encodings of Rintanen. We generated the formulas for makespan 3, i.e., F_3 for the first problem of each of the domains and counted the number of variables, clauses and the percentage of binary and Horn clauses.

Table 3.6 contains the number of variables in F_3 for each domain and encoding. We can observe, that our $R^2\exists$-Step encoding has conspicuously the highest number of variables in each case, followed by the Reinforced, SASE, and Direct encoding. This is the same order as we would expect from the calculated upper bounds on the number of variables. We can also observe, that the two Rintanen encodings usually have fewer variables than our encodings. Overall, the number of variables is rather similar between our ∀-Step encodings and the Rintanen encodings, except for the tidybot domain, where the Rintanen encodings have much fewer variables. Tidybot is also the domain, where all our encodings are significantly outperformed by R∀ and R∃ in the number of solved instances.

The number of clauses is presented in Table 3.7. Our $R^2\exists$-Step encoding has the highest number of clauses in most of the cases, but for some domains, like parking and tidybot, our ∀-Step encodings produce much more clauses. There is again a big difference between our and Rintanen's encodings for the tidybot

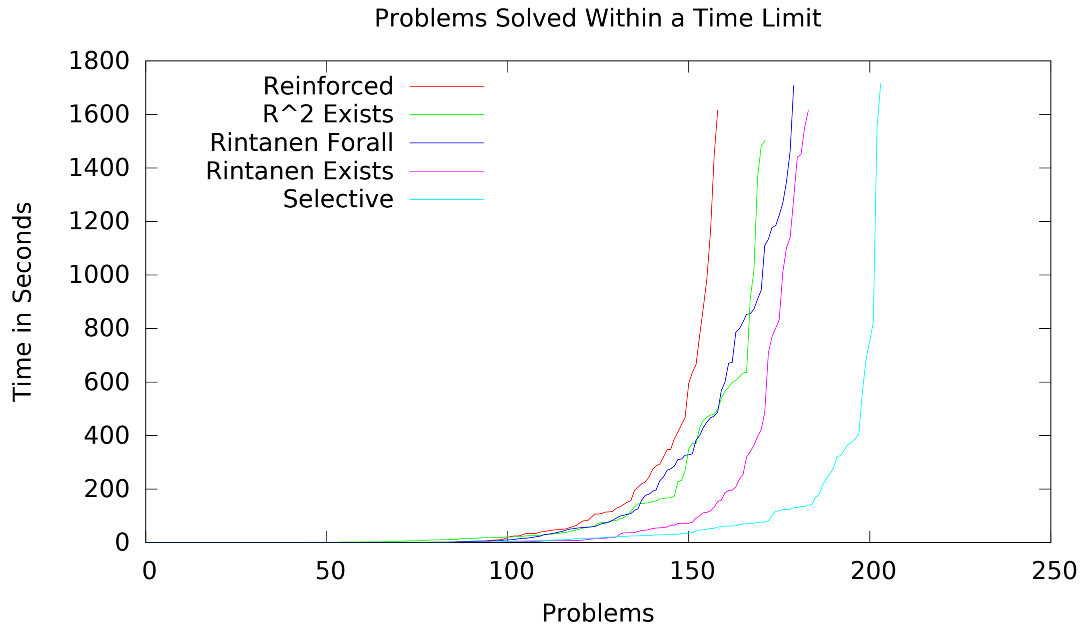


Figure 3.6: This plot represents the number of problems that a SAT based planner using the given encoding can solve under a given time limit. It is obtained by sorting the SAT solving times for the solved problems for each encoding.

Table 3.5: The sum of makespans of the plans found within the time limit for each domain.

Domain	Dir	SASE	Reinf	$R^2\exists$	Sel	R∀	R∃
barman	121	121	121	84	102	282	100
elevators	190	190	190	85	190	190	130
floortile	302	181	344	169	344	290	230
nomystery	347	119	347	30	347	347	217
openstacks	-	-	-	93	60	-	-
parcprinter	261	261	261	30	40	261	261
parking	-	-	-	-	-	-	-
pegsol	222	131	222	158	166	245	270
scanalyzer	83	61	95	17	91	114	119
sokoban	60	60	60	27	60	396	396
tidybot	14	15	15	6	7	310	344
transport	221	242	262	55	277	262	191
visitall	223	110	146	34	39	188	188
woodworking	68	68	68	33	40	66	66

Table 3.6: Number of variables for makespan 3, first problem of each domain

Domain	Dir	SASE	Reinf	$R^2\exists$	R$\forall$	R$\exists$
barman	1606	2536	3068	16180	1226	1986
elevators	1394	2684	2992	25264	1394	1394
floortile	736	1360	1664	7872	688	688
nomystery	1270	2416	2636	54148	1171	1171
openstacks	904	1324	1568	13328	904	1664
parcprinter	513	836	1172	3944	393	393
parking	12824	32560	33720	412948	12304	12304
pegsol	955	1972	2372	29548	841	841
scanalyzer	1620	2336	2432	30688	1604	1604
sokoban	828	1648	2080	18000	748	748
tidybot	14881	20576	21684	253568	288	288
transport	2136	3552	3840	43136	2136	2136
visitall	64	116	156	208	52	52
woodworking	1293	1988	2420	14580	726	726

domain.

Table 3.8 contains the percentage of binary and Horn clauses in the generated formulas. The percentage of Horn and binary clauses is very similar for each encoding and each domain, there are usually slightly more Horn clauses. Our $R^2\exists$-Step encoding tends to have the smallest percentage of both binary and Horn clauses. The highest values for each encoding are for the parking domain, which is the hard domain, where none of the encodings solved any of the problems. This counters the hypothesis, that formulas with a high percentage of binary or Horn clauses are easier to solve. Also the values for the tidybot domain and our $\forall$-Step encodings are very high, while the values for the Rintanen encoding are much lower. Both Rintanen's encodings significantly outperform our encodings in solving the problems of this domain.

3.7.5 Discussion

The experimental results revealed, that our new $R^2\exists$-Step encoding is very efficient for some domains, that other encodings cannot handle. On the other hand, its performance highly depends on the ranking of actions and it does not perform well on domains with a high number of transitions per variable. Fortunately, we could design a simple selective approach which can decide when the $R^2\exists$-Step encoding should be used and by trying several action rankings the problem of proper ranking selection was partially solved. This selective encoding approach could significantly outperform all the other methods including the state-of-the-art

Table 3.7: Number of clauses for makespan 3 for the first problem of each domain.

Domain	Dir	SASE	Reinf	$R^2\exists$	R∀	R∃
barman	28249	34530	33855	36760	13127	11279
elevators	7981	34994	9918	54725	13046	9716
floortile	4107	13568	6035	17413	4261	3613
nomystery	7037	84868	8498	112541	13075	8179
openstacks	7104	9436	8020	27593	8057	6767
parcprinter	2111	5241	3257	8081	2043	2091
parking	2233859	5763267	2975355	890069	359197	109717
pegsol	6799	67402	8849	63150	7321	7411
scanalyzer	13916	12688	11156	69840	18544	18544
sokoban	5847	25377	9086	38943	6298	6298
tidybot	2124122	2742541	2742373	574056	1804	1780
transport	14676	33767	15019	94232	20332	19084
visitall	139	485	329	359	169	169
woodworking	9507	11332	10520	29218	4941	4869

Table 3.8: The percentage of binary (B) and Horn(H) clauses in the formula for makespan 3 for the first problem of each domain.

Domain	Dir		SASE		Reinf		$R^2\exists$		R∀		R∃	
	B	H	B	H	B	H	B	H	B	H	B	H
barman	98.4	98.6	95.6	96.0	96.6	97.0	57.9	58.1	96.0	96.6	95.4	96.1
elevators	97.0	97.1	94.6	95.8	88.9	91.7	84.3	84.3	95.8	96.5	94.4	95.2
floortile	93.8	94.4	92.7	94.0	89.5	91.8	78.3	78.5	91.2	93.0	89.7	91.7
nomystery	97.6	97.7	98.0	98.3	86.4	89.2	93.3	93.3	97.1	97.5	95.4	96.1
openstacks	97.8	98.3	94.1	95.2	95.0	96.0	83.0	83.1	96.4	97.3	95.8	96.8
parcprinter	90.0	92.0	88.1	90.1	86.4	89.4	80.5	81.1	87.8	91.0	88.1	91.2
parking	99.9	99.9	99.6	99.8	99.8	99.9	81.9	81.9	99.5	99.6	98.4	98.6
pegsol	94.6	95.6	97.5	97.9	88.6	91.4	89.9	90.0	92.9	92.0	92.9	92.1
scanalyzer	99.5	99.6	95.7	96.5	96.9	97.4	71.0	71.0	99.3	99.4	99.3	99.4
sokoban	94.0	94.5	94.7	95.5	89.9	92.3	87.1	87.1	91.3	92.8	91.3	92.8
tidybot	99.9	99.9	99.9	99.9	99.9	99.9	56.3	56.3	90.6	88.0	90.5	87.9
transport	98.4	98.5	94.3	95.6	92.1	93.8	82.3	82.3	97.5	97.9	97.3	97.7
visitall	79.9	84.9	81.9	84.9	83.3	87.8	69.4	71.3	74.6	80.5	74.6	80.5
woodworking	96.8	97.4	90.2	91.8	92.2	93.5	69.1	69.3	91.2	93.0	91.1	92.9

encodings of Rintanen.

By looking at the quantitative properties of the formulas we learned, that our encodings in many cases use much more variables and clauses than the state-of-the-art encodings. We also noticed, that a high percentage of binary or Horn clauses does not indicate that an encoding will be successful. This is interesting, since formulas that contain only Horn clauses (Horn formulas) and formulas with only binary clauses (2SAT formulas) can be solved in polynomial time [10].

4. Improving Plans

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 [25], Fast Downward [23] or LPG [21] 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.

The first section of this chapter contains a review of the related work. In the following sections we restrict ourselves to optimizing plans by only removing redundant actions from them.

In the second section we review a polynomial heuristic algorithm called Action Elimination, which often removes most of the redundant actions from a plan, but cannot guarantee to remove them all, i.e., to achieve perfect justification. This algorithm does not take into account action costs. We extend it into a new polynomial heuristic algorithm called Greedy Action Elimination, which tries to remove costly redundant actions from a plan in order to improve its cost.

In the third section we will introduce our propositional encoding of the problem of plan redundancy. This encoding will be used in the fourth and fifth sections to construct SAT, Partial MaxSAT, and Weighted Partial MaxSAT formulas that will be used to design algorithms that can achieve perfect justification and solve the problems of minimal length plan reduction and minimal plan reduction.

The last section in this chapter will present experimental results proving the practical usefulness of the proposed techniques on plans acquired by current state-of-the-art satisficing planners for planning tasks from the 2011 International Planning Competition.

4.1 Related Work

Various techniques have been proposed for post-planning plan optimization. Westerberg and Levine [39] 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 [30] 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 [18] improves quality of plans by identifying suboptimal subsequences of actions according to heuristic estimation (distances 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

```
    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 4.1: Pseudo-code of the Action Elimination algorithm as presented in [30].

sequence of actions connecting the given states. In our previous work we have used a similar approach for optimizing parallel plans [3]. A recent technique [36] 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 subcategory of post-planning plan optimization. An influential work [20] 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 [30] 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 [30] is a special case of Perfectly Justified plans having minimal cost of the plan. Both Perfect Justification and Minimal reduction are NP-complete [20, 30]. 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 [13, 14].

```
    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(a[i], s) then
R08       s := apply(P[i], s)
R09       P′ := append(P′, P[i])
R10   return P′
```

Figure 4.2: Pseudo-code of the evaluateRemove and remove functions used in the Greedy Action Elimination algorithm (Figure 4.3).

4.2 Greedy Action Elimination

There are several heuristic approaches [13, 20, 30] , which can identify most of the redundant actions in plans in polynomial time. One of the most efficient of these approaches was introduced in [20] under the name Linear Greedy Justification. It was reinvented in [30] and called Action Elimination. In this thesis we use the latter name and extend the algorithm to take into account the action costs. We begin by describing the original Action Elimination algorithm.

Action Elimination (see Figure 4.1) 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.

```
    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 4.3: 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.

The Action Elimination algorithm ignores the cost of the actions and eliminates a set of redundant actions as soon as it discovers it. In this thesis we modify Action Elimination to be less 'impatient'. Before removing any set of redundant actions, we will identify each such set and 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 call this new algorithm *Greedy Action Elimination*.

Greedy Action Elimination uses two functions: evaluateRemove and remove (see Figure 4.2). The function evaluateRemove tests if the k-th action and the following actions that depend on it can be removed, i.e., whether it is greedily justified. 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 4.3) 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 cycle runs at most n times (each time at least one action is eliminated) and each cycle calls n times evaluateRemove and once remove. Both these functions run in $O(np)$ time, therefore the total runtime is $O(n(n^2p + np)) = O(n^3p)$.

There are plans, where Action Elimination cannot eliminate all redundant actions [30]. 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 [16] and how much do the reduced plans differ from the minimal plan reduction, i.e., the best possible outcome of action elimination. To find out, first we need to design an algorithm that can check perfect justification and also an algorithm for solving minimal plan reduction. As already mentioned, these problems are NP-complete and therefore we find it reasonable to solve them using SAT and MaxSAT based approaches. In the next section we will introduce an encoding of the problem of redundant actions into propositional logic.

4.3 Propositional Encoding of Plan Redundancy

This section is devoted to describing how a given planning task Π and a valid plan P for Π can be encoded into 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* for 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 $\text{Opps}(c, i, j)$ the set of ranks of opposing actions of the condition c which have their rank between i and j ($i \leq j$). Similarly, by $\text{Supps}(c, i)$ 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 Boolean 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 $P[i]$ (the i-th action of P, i.e., 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 $P[j]$ of c_i with the rank $j < i$ and there is no opposing action of c_i with its 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 $(i-1) < |P|$ of the second kind for each condition and each time i. 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 a 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.

$$F_{c,i,0} = \bigwedge_{j \in \text{Opps}(c,0,i)} (\neg y_{c,i,0} \vee \neg a_j) \tag{4.1}$$

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.

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

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

$$F_{c,i,j} = (\neg y_{c,i,j} \vee a_j) \bigwedge_{k \in \text{Opps}(c,j,i)} (\neg y_{c,i,j} \vee \neg a_k) \tag{4.2}$$

These clauses are equivalent to the implications that if the given option is true, then the given supporting action variable is true and all the variables of 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.

$$F_{c,i} = (y_{c,i,0} \bigvee_{j \in \text{Supps}(c,i)} y_{c,i,j}) \wedge F_{c,i,0} \bigwedge_{j \in \text{Supps}(c,i)} F_{c,i,j} \tag{4.3}$$

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 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 variable a_i the following clauses will enforce, that if it is true, then all the preconditions of $P[i]$ must hold.

$$F_{a_i} = \bigwedge_{c \in \text{pre}(a_i)} (\neg a_i \vee F_{c,i}) \tag{4.4}$$

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

$$F_A = \bigwedge_{a_i \in P} F_{a_i} \tag{4.5}$$

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.

$$F_G = \bigwedge_{c \in s_G} F_{c,n} \tag{4.6}$$

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

$$F_{\Pi,P} = F_G \wedge F_A \tag{4.7}$$

From a satisfying assignment of this formula we can produce a plan reduction of P. A plan obtained using a truth assignment ϕ will be denoted as P_ϕ. Its formal definition follows.

Definition 27 (Subsequence Extraction)**.** *Let P be a plan for a planning task Π. Let ϕ be a truth assignment for the formula $F_{\Pi,P}$. We define P_ϕ to be a subsequence of P such that $P[i]$ is present in P_ϕ if and only if $\phi(a_i) = True$.*

4.3.1 Correctness and Size

In this subsection we will prove the correctness of the encoding, i.e., that a a truth assignment ϕ satisfies $F_{\Pi,P}$ if and only if P_ϕ is a plan reduction of P ($P_\phi \preceq P$). We will also provide an upper bound on the size of the formula $F_{\Pi,P}$.

In order to prove the correctness we split the construction of P_ϕ from P and ϕ into two steps. First, we replace each action $P[i]$ such that $\phi(a_i) = False$ in P by a special dummy action denoted by $\sqcup$. A sequence obtained this way will be called a *sparse sequence* and denoted by $P_\phi^\sqcup$. The dummy action $\sqcup$ has no preconditions or effects and its only purpose is to take up some positions in a sequence. A formal definition follows.

Definition 28 (Dummy Action, Sparse Sequence)**.** *A* dummy action $\sqcup$ *is an action such that* $\text{pre}(\sqcup) = \text{eff}(\sqcup) = \emptyset$*.*

Let P be a plan for a planning task Π. Let ϕ be a truth assignment for the formula $F_{\Pi,P}$. A sparse sequence $P_\phi^\sqcup$ *is a sequence of length $|P|$ of actions such that*

$$P_\phi^\sqcup[i] = \begin{cases} P[i] & \text{if } \phi(a_i) = True \\ \sqcup & \text{if } \phi(a_i) = False \end{cases} \tag{4.8}$$

The second step of the construction of P_ϕ is the removal of the dummy actions from $P_\phi^\sqcup$. Clearly, P_ϕ obtained by this method is a plan for a planning task Π if and only if $P_\phi^\sqcup$ is a plan for Π. Therefore it is enough to prove the correctness property for $P_\phi^\sqcup$. Let us start with the following lemma.

Lemma 1. *Let P be a plan for a planning task Π. Let ϕ be a truth assignment for the formula $F_{\Pi,P}$ and $P_\phi^\sqcup$ a sparse sequence as defined in Definition 28. Let s_i be the state before the i-th action in $P_\phi^\sqcup$, i.e., $s_0 = s_I$, $s_{i+1} = \text{apply}(P_\phi^\sqcup[i], s_i)$. For each assignment/condition c and each time $i \in 1, \ldots, |P|$ the clauses $F_{c,i}$ defined in Equation 4.3 are satisfied only if c holds in the i-th state during the execution of $P_\phi^\sqcup$, i.e., $c \in s_i$.*

Proof. In order to satisfy $F_{c,i}$ at least one of the option variables $y_{c,i,*}$ must be True. This option variable can either represent the option that the condition c_i is satisfied in s_i since the initial state or there is a supporting action for c with a rank smaller than i and there is no opposing action in the way to destroy the condition. In the first case, the clauses in Equation 4.1 will ensure that no opposing action is in $P_\phi^\sqcup$ with a rank smaller than i and therefore the condition still holds in s_i. In the second case, the clauses in Equation 4.2 ensure that a supporting action for c is present in $P_\phi^\sqcup$ before the i-th step and there is no opposing action between that supporting action and the state s_i. Therefore the condition c must hold in s_i in both cases. □

By applying this lemma on the goal conditions and the preconditions of the effects we can prove the correctness of the encoding.

Proposition 6. *Let P be a plan for a planning task Π. Let ϕ be a truth assignment for the formula $F_{\Pi,P}$. The assignment ϕ satisfies $F_{\Pi,P}$ if and only if P_ϕ is a plan reduction of P, i.e., $P_\phi \preceq P$.*

Proof. From Definition 27 it is clear, that P_ϕ is a subsequence of P, therefore it remains to prove that P_ϕ is a valid plan for Π if and only if ϕ satisfies $F_{\Pi,P}$.

Let us start by showing that if ϕ satisfies $F_{\Pi,P}$, then P_ϕ is valid plan for Π. First, we prove, that the sequence $P_\phi^\sqcup$ is a plan for Π. A plan is valid if its i-th action is applicable in the state s_i (where $s_0 = s_I$ and $s_{i+1} = \text{apply}(P_\phi^\sqcup, s_i)$) and after the application of the last action the goal conditions are satisfied. The clauses defined in Equation 4.6 ensure that the clauses $F_{c,n}$ must be satisfied for each goal condition $c \in s_G$. Using Lemma 1, this implies, that the goal conditions are satisfied after the execution of $P_\phi^\sqcup$. If an action $P_\phi^\sqcup[i]$ is a dummy action, then it is applicable in any state since it has no preconditions. Otherwise the Boolean variable a_i must be True under ϕ. Due to the clauses in Equation 4.4 the clauses $F_{c,i}$ must be satisfied for each $c \in \text{pre}(P[i])$. Again, using Lemma 1, this implies, that the preconditions are satisfied in s_i, i.e., before the application of $P_\phi^\sqcup[i]$. Therefore we can conclude, that $P_\phi^\sqcup$ is valid plan for Π which implies, that P_ϕ is a valid plan for Π as well. This is due to the fact, that removing dummy actions cannot affect the execution of any plan and P_ϕ can be obtained from $P_\phi^\sqcup$ this way.

On the other hand, let P_ϕ be a plan reduction of P. We will prove that ϕ satisfies $F_{\Pi,P}$, i.e., each clause is satisfied under ϕ. For the sake of contradiction let us assume, that a clause C is not satisfied. We will show, that this leads to a contradiction with the validity of the plan P_ϕ for Π. There are four kinds of clauses in $F_{\Pi,P}$:

- $(\neg y_{c,i,k} \vee \neg a_j)$ (used in 4.1 and 4.2). If this clause is unsatisfied it means that both a_j and $y_{c,i,k}$ are True, i.e., the option must be satisfied but the opposing action is present.

- $(\neg y_{c,i,k} \vee a_j)$ (used in 4.2). Similarly to the previous case, if this clause is not satisfied it means that the option is true, but the corresponding supporting action is not in the plan reduction.

- $(y_{c,n,0} \vee \cdots \vee y_{c,n,k})$ (used for goal conditions $c \in s_G$). The clause is unsatisfied if all the option variables are false, i.e., none of the options to satisfy c is selected.

- $(\neg a_i \vee y_{c,i,0} \vee \cdots \vee y_{c,i,k})$ (used for preconditions of actions). The clause is unsatisfied if a_j is True ($P[i]$ is in the plan), but none of the options to satisfy the precondition c of $P[i]$ is selected.

We have shown that if any of the clauses in $F_{\Pi,P}$ is unsatisfied under ϕ then P_ϕ cannot be a valid plan. □

The following observations follow directly from the Proposition. The formula $F_{\Pi,P}$ is always satisfiable for any planning task Π and its valid 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 subsection by computing the following upper bound on the size of the formula $F_{\Pi,P}$.

Proposition 7. *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. 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. □

4.4 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, i.e., a plan without redundant actions.

The idea is very similar to the standard planning as SAT approach (from the previous chapter), 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.

First we need to construct a CNF formula which is satisfiable if and only if a plan P is redundant for the planning task Π. This can be achieved easily by adding the following clause to the formula $F_{\Pi,P}$.

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

This clause is satisfied if at least one of the actions in the plan is omitted. Therefore it is easy to see (using Proposition 6), that the formula $F_{\Pi,P} \wedge F_R$ is satisfied if and only if P is a redundant plan for Π.

The pseudo-code of the redundancy elimination algorithm is presented in Figure 4.4. 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 $F_{\Pi,P} \wedge F_R$ 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).

```
    RedundancyElimination (Π, P)
I1    F := encodeRedundancy(Π, P)
I2    while isSatisfiable(F) do
I3      φ := getSatAssignment(F)
I4      P := P_φ
I5      F := encodeRedundancy(Π, P)
I6    return P
```

Figure 4.4: Pseudo-code of the SAT based redundancy elimination algorithm. It returns a perfectly justified plan.

```
     IncrementalRedundancyElimination (Π, P)
II01   solver = new SatSolver
II02   solver.addClauses(encodeRedundancy(Π, P))
II03   while solver.isSatisfiable() do
II04     φ := solver.getSatAssignment()
II06     C := ⋁{¬a_i| φ(a_i) = True}
II07     solver.addClause(C)
II08     foreach a_i ∈ P do if φ(a_i) = False then
II09       solver.addClause({¬a_i})
II10     P := P_φ
II11   return P
```

Figure 4.5: Pseudo-code of the incremental SAT based redundancy elimination algorithm.

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.

The incremental algorithm is presented in Figure 4.5. 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 this is not necessary. The algorithm also adds unit clauses to enforce that the already eliminated actions cannot be reintroduced.

The worst-case time complexity of this algorithm is $O(2^n)$, where n is the length of the input plan. This follows from the exponential worst-case time complexity of the SAT solving part.

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, i.e., the result depends on the order in which the redundant actions

```
      MinimalLengthReduction (Π, P)
MRE1    F := encodeMinimalLengthReduction(Π, P)
MRE2    φ := partialMaxSatSolver(F)
MRE3    return P_φ
```

Figure 4.6: Pseudo-code of the minimal length plan reduction algorithm.

are removed (see Example 4). This issue is addressed in the next section.

4.5 Minimal Length Reduction and Minimal Reduction

In this section we describe how to do the best possible redundancy elimination for a plan. By best we can either mean removing the maximum number of redundant actions (minimal length plan reduction (MLR)) or removing redundant actions with the maximal total cost (minimal plan reduction (MR)).

The plans resulting from MLR and MR are 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 or MR (see Example 4).

The solution we propose for MLR 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 of $F_{\Pi,P}$.

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

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

The algorithm (Figure 4.6) 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 (see Definition 27).

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

A WPMaxSAT formula also consists of two kinds of clauses – soft and hard. The soft clauses have a non-negative integer weight assigned to them. The task

```
    MinimumReduction (Π, P)
MR1   F := encodeMinimumReduction(Π, P)
MR2   φ := weightedPartialMaxSatSolver(F)
MR3   return P_φ
```

Figure 4.7: Pseudo-code of the minimum reduction algorithm.

of a WPMaxSAT solver is to find a satisfying assignment that satisfies all the hard clauses and maximizes the sum of the weights of satisfied soft clauses.

Our WPMaxSAT formula is very similar to the PMaxSAT formula used for MLR. 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. Therefore the maximization of the total weight of these clauses is equivalent to removing actions with a maximal total cost. The validity of the plan obtained from the satisfying assignment is guaranteed thanks to Proposition 6 and the fact that all the hard clauses must be satisfied. The algorithm is analogous to MLR, it is displayed in Figure 4.7.

The worst-case time complexity of both MR and MLR is $O(2^n)$, where n is the length of the input plan. This follows from the exponential worst-case time complexity of the PMaxSAT and WPMaxSat solving part.

4.6 Experiments

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 SAT, PMaxSAT, and WPMaxSat based algorithms – SAT reduction, 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 2011 International Planning Competition [16] and compared the algorithms with each other and with a plan optimization tool which focuses on redundant inverse actions elimination (IAE) [13].

4.6.1 Experimental Setting

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 [23], 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 [23], Metric FF [24], and Madagascar [32]. Each of these planners was configured to find plans as fast as possible and ignore plan quality.

We tested six redundancy elimination methods:

- *Inverse action elimination (IAE)* is a polynomial heuristic redundancy elimination algorithm that focuses on removing the most common set of redun-

dant actions – pairs and groups of inverse actions [13]. It is implemented in C++.

- *Action Elimination (AE)* is our own Java implementation of the Action Elimination algorithm as displayed in Figure 4.1.
- *Greedy Action Elimination (GAE)* is our Java implementation of our new Greedy Action Elimination algorithm as displayed in Figure 4.3.
- *SAT Reduction (SAT)* is our Java implementation of our new incremental SAT reduction algorithm (see Figure 4.5). We used the Java SAT solver Sat4j [8] since it is incremental and is very convenient to use in a Java application. This algorithm guarantees to achieve perfect justification, i.e., it outputs a plan that contains no redundant actions.
- *Minimal Length Plan Reduction (MLR)* is a Partial MaxSAT reduction based algorithm displayed in Figure 4.6. We implemented the translation in Java and used the QMaxSAT [29] state-of-the-art MaxSAT solver written in C++ to solve the instances. We selected QMaxSAT due to its good availability and very good results in the 2013 MaxSAT competition. This algorithm also guarantees to achieve perfect justification. Furthermore, it guarantees to remove the highest possible number of redundant actions from the initial plan.
- *Minimal Reduction (MR)* is a Weighted Partial MaxSAT reduction based algorithm displayed in Figure 4.7. The translation is implemented in Java and we used the Toysat [35] Weighted MaxSAT solver written in Haskell to solve the instances. Although Toysat did not place very well in the 2013 MaxSAT competition, it was able to significantly outperform all the other available solvers on our formulas. Like the previous two, this algorithm guarantees to achieve perfect justification as well. Furthermore, it guarantees to remove the most costly set of redundant actions from the initial plan.

For these methods we measured the total runtime and the total number and 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 runtime for the optimization was unlimited, however it never took more than 5 minutes for any problem. The benchmark problems are taken from the satisficing track of IPC 2011 [16].

4.6.2 Number of Removed Actions

First, let us take a look at the number of removed redundant actions. Looking at the number of removed actions in Table 4.1 we can make several interesting observations. For example, in the nomystery and pegsol domains no redundant

Table 4.1: The number of removed actions. The table contains the number of found plans, their total length and the total number of eliminated actions by the six redundancy elimination methods.

	Domain	Plans	Length	IAE	AE	GAE	SAT	MLR	MR
Metric FF	elevators	20	4273	79	79	79	79	79	79
	floortile	2	81	9	10	10	10	10	10
	nomystery	5	107	0	0	0	0	0	0
	parking	18	1546	118	124	124	124	124	124
	pegsol	20	637	0	0	0	0	0	0
	scanalyzer	18	571	0	30	30	30	30	30
	sokoban	13	2504	0	6	6	6	6	6
	transport	6	1329	145	164	165	164	165	165
Fast Downward	barman	20	3749	400	528	555	596	629	629
	elevators	20	4625	88	94	94	94	94	92
	floortile	5	234	22	22	22	22	22	22
	nomystery	13	451	0	0	0	0	0	0
	parking	20	1494	4	4	4	4	4	4
	pegsol	20	644	0	0	0	0	0	0
	scanalyzer	20	823	0	26	26	26	26	26
	sokoban	17	5094	0	244	236	458	460	414
	transport	17	4059	235	289	290	289	290	290
Madagascar	barman	8	1785	161	303	310	303	318	318
	elevators	20	11122	1461	2848	2927	3021	3138	2952
	floortile	20	1722	30	30	30	30	30	30
	nomystery	15	480	0	0	0	0	0	0
	parking	18	1663	152	152	152	152	152	152
	pegsol	19	603	0	0	0	0	0	0
	scanalyzer	18	1417	0	232	236	232	236	236
	sokoban	1	121	22	22	22	22	22	20
	transport	4	1446	246	508	535	532	553	553

actions were found in plans obtained by any planner. All the other domains contain some redundant actions, most notably the plans found by Madagascar for the elevators domain contain over three thousand redundant actions.

The performance of the IAE method is the poorest. Clearly, this is because of the IAE method is specific, i.e., only pairs or pairs of nested inverse actions are considered. Nevertheless, there are examples of domains, where this method removes the same number of redundant actions as any other method (floortile

Table 4.2: The number of times that AE and GAE achieve perfect justification, #P denotes the number of found plans by the given planner/domain combination. The cases where perfect justification is not achieved for each plan are emphasized by bold text.

Domain	Metric FF			Fast Downward			Madagascar		
	#P	AE	GAE	#P	AE	GAE	#P	AE	GAE
barman	0	0	0	20	**17**	**17**	8	8	8
elevators	20	20	20	20	20	20	20	**11**	**14**
floortile	2	2	2	5	5	5	20	20	20
nomystery	5	5	5	13	13	13	15	15	15
parking	18	18	18	20	20	20	18	18	18
pegsol	20	20	20	20	20	20	19	19	19
scanalyzer	18	18	18	20	20	20	18	18	18
sokoban	13	13	13	17	**15**	**15**	1	1	1
tidybot	17	17	17	16	16	16	16	16	16
transport	6	6	6	17	17	17	4	**1**	4
visitall	2	2	2	20	20	20	0	0	0
woodworking	19	19	19	20	20	20	20	20	20

and parking plans by Fast Downward and Madagascar).

The AE method outperforms IAE in 14 cases and achieves minimal length plan reduction in 18 cases out of 26, which is a very good result for a polynomial heuristic algorithm. GAE improves upon AE in 7 cases.

The SAT method has very similar results to the AE method. Although it guarantees perfect justification, this is often (in 8 cases) not enough to remove the highest number of redundant actions. For the transport problems (for each planner) SAT is outperformed by the polynomial GAE. This is possible since removing redundant actions is not confluent (see Example 4).

As expected, the MLR method removes the highest (or equal) number of actions in each case. In 4 cases it gives shorter plans than the MR method. This demonstrates, that when optimizing the cost of plans, not necessarily the highest number of redundant actions needs to be removed (see also Example 4).

One of the questions we asked before in the text was how often AE and GAE achieve perfect justification, i.e., a plan that has no redundant actions. We tested the plans obtained by AE and GAE using our SAT encoding whether they are still redundant. The results are displayed in Table 4.2. The data indicates that in the majority of cases AE and GAE achieved perfect justification. There are 17 plans for AE and 11 for GAE where perfect justification was not achieved.

Table 4.3: The cost of removed actions. The table contains the number of found plans (#P), their total cost and the total cost of eliminated actions by the six redundancy elimination methods.

	Domain	#P	Cost	IAE	AE	GAE	SAT	MLR	MR
Metric FF	elevators	20	25618	2842	2842	2842	2842	2842	2842
	floortile	2	195	29	30	30	30	30	30
	nomystery	5	107	0	0	0	0	0	0
	parking	18	1546	118	124	124	124	124	124
	pegsol	20	300	0	0	0	0	0	0
	scanalyzer	18	1137	0	62	62	62	62	62
	sokoban	13	608	0	2	2	2	2	2
	transport	6	29674	2650	3013	3035	3013	3035	3035
Fast Downward	barman	20	7763	436	753	780	893	926	926
	elevators	20	28127	1068	1218	1218	1218	1218	1218
	floortile	5	572	66	66	66	66	66	66
	nomystery	13	451	0	0	0	0	0	0
	parking	20	1494	4	4	4	4	4	4
	pegsol	20	307	0	0	0	0	0	0
	scanalyzer	20	1785	0	78	78	78	78	78
	sokoban	17	1239	0	58	58	102	102	102
	transport	17	74960	4194	5259	5260	5259	5260	5260
Madagascar	barman	8	3360	296	591	598	591	606	606
	elevators	20	117641	7014	24096	24728	26702	28865	28933
	floortile	20	4438	96	96	96	96	96	96
	nomystery	15	480	0	0	0	0	0	0
	parking	18	1663	152	152	152	152	152	152
	pegsol	19	280	0	0	0	0	0	0
	scanalyzer	18	1875	0	232	236	232	236	236
	sokoban	1	33	0	0	0	0	0	0
	transport	4	20496	4222	6928	7507	7444	7736	7736

4.6.3 Cost of Removed Actions

The total cost of the removed actions by the six described methods is displayed in Table 4.3. If we compare these results to the length results in Table 4.1 we can observe that for the sokoban and pegsol problems the cost of the plans is smaller than their length. This implies that they contain actions with zero cost. If we look back at Table 4.1 we can indeed observe that for the sokoban problems MR

removes less actions than the other methods, since redundant actions with zero cost are ignored.

The IAE method is again the weakest followed by AE and GAE. The AE algorithm, although it ignores the action costs, performs rather well. Except for 8 planner/domain combinations it achieves minimal reduction, i.e., the best possible result.

The GAE algorithm improves upon AE in 7 cases (the same as for length) and achieves minimal reduction in all but 5 planner/domain pairs.

The MLR method is guaranteed to remove the maximum number of redundant actions (not considering their cost) and this is also enough to achieve minimal plan reduction in each case except for the Madagascar plans for the elevators domain.

As expected, MR provides the best results, however these results are often not strictly better than the results of the polynomial methods.

4.6.4 Runtime

Table 4.4 contains the total time required by the elimination algorithms to improve the plans for each domain/planner combination.

We can immediately notice that the runtime of all of our 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 MLR on the Fast Downward plans for the sokoban domain.

Surprisingly, the runtime of IAE is high despite the complexity results [13]. 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 runtime of the GAE is not significantly increased compared to AE in most of the cases except for the Madagascar plans for the elevators domain. Considering the better results obtained by GAE we can say, that using GAE instead of just AE pays off in most of the cases.

Note, that the runtime of the MLR method is often the smallest contrary to the fact, that it is the only one which guarantees eliminating the maximum number of redundant actions. This can be explained by the excellent performance of the partial MaxSAT solver we used – QMaxSAT [29]. 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 (MR) is guaranteed to achieve minimal plan 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 usually need to solve this difficult domain.

Table 4.4: The runtime of the elimination algorithms. The table contains the number of found plans, and the total runtime for the entire domain of the six redundancy elimination methods in seconds.

	Domain	Plans	IAE	AE	GAE	SAT	MLR	MR
Metric FF	elevators	20	1,34	0,70	0,78	3,27	0,17	1,77
	floortile	2	0,00	0,01	0,02	0,10	0,00	0,00
	nomystery	5	0,17	0,01	0,01	0,18	0,00	0,00
	parking	18	0,19	0,10	0,30	1,64	0,03	0,26
	pegsol	20	0,00	0,06	0,06	1,23	0,02	0,29
	scanalyzer	18	0,00	0,04	0,07	0,94	0,01	0,16
	sokoban	13	0,72	0,31	0,31	2,62	0,36	9,07
	transport	6	0,35	0,29	0,41	2,21	0,25	3,40
Fast Downward	barman	20	1,04	0,50	0,94	7,57	0,44	10,60
	elevators	20	1,58	0,70	0,93	3,57	0,19	2,00
	floortile	5	0,00	0,03	0,07	0,29	0,00	0,02
	nomystery	13	4,31	0,03	0,04	0,53	0,01	0,04
	parking	20	0,03	0,10	0,10	1,35	0,03	0,21
	pegsol	20	0,01	0,06	0,06	1,24	0,02	0,30
	scanalyzer	20	0,00	0,07	0,08	1,40	0,03	0,49
	sokoban	17	6,41	0,54	0,70	9,19	1,87	252,15
	transport	17	1,15	0,56	0,90	3,03	0,18	1,90
Madagascar	barman	8	1,02	0,22	0,44	3,71	0,27	5,88
	elevators	20	6,86	1,19	9,07	18,38	1,90	31,06
	floortile	20	0,07	0,30	0,33	1,38	0,03	0,24
	nomystery	15	2,62	0,03	0,03	0,56	0,00	0,02
	parking	18	0,14	0,13	0,33	1,86	0,05	0,32
	pegsol	19	0,00	0,06	0,05	1,17	0,01	0,26
	scanalyzer	18	0,06	0,20	0,45	1,67	0,04	0,31
	sokoban	1	0,02	0,02	0,04	0,26	0,01	0,19
	transport	4	0,24	0,21	0,49	1,69	0,16	8,14

4.6.5 Discussion

Clearly, the MR method is guaranteed to provide minimal reduction of plans and therefore cannot be outperformed (in terms of quality) by the other methods. Similarly, the MLR method cannot be outperformed in terms of plan lengths. Despite the exponential worst-case time complexity of these methods, 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, the measured runtimes often depend more on the efficiency of implementation of particular methods rather than the worst-case complexity properties.

Our results also show that in the most cases using the polynomial method (AE or GAE) provides minimal reduction, so the MR method usually does not lead to strictly better results. Guaranteeing in which cases (Greedy) AE provides minimal reduction is an interesting open question.

Conclusion

In this thesis we have shown how can satisfiability (SAT) and maximum satisfiability (MaxSAT) solvers be efficiently used to both find plans and improve them. Finding plans via SAT solving is not a new idea. It has been around for several decades and it is one of the most successful approaches to automated planning.

Our main contribution to the topic of planning as SAT is the introduction of two new encoding schemes, the Reinforced and the $R^2\exists$-Step encoding. These two encodings work well for different sets of planning problems (domains) but we were able to find a simple rule which allows the automatic selections of the best encoding for a given planning task. Using this rule we designed a combined encoding that can significantly outperform the existing state-of-the-art encodings.

As for the second problem – the improvement of plans we have focused on the special case of removing redundant (unnecessary) actions from plans, which is an NP-hard optimization problem. Prior to our work, there existed only heuristic algorithms that are not guaranteed to remove all the redundant actions. The most successful of these algorithms is called Action Elimination (AE). Based on the ideas of AE we have introduced our own heuristic algorithm – Greedy Action Elimination (GAE), which, contrary to AE, takes actions cost into account. GAE outperformed AE and the other existing heuristic approaches on benchmark problems.

Furthermore, we have introduced a SAT encoding for the problem of plan redundancy. Using this encoding we have proposed three new methods which can completely solve three optimization problems related to redundancy elimination. The first method uses a SAT solver to produce perfectly justified plans, i.e., plans without redundant actions. The second method uses a partial MaxSAT solver to remove the highest possible number of redundant actions from plans. Finally, the third method guarantees to remove the set of redundant actions with the highest total cost and uses a weighted partial MaxSAT solver. Thanks to the existence of powerful modern SAT and MaxSAT solvers, these methods work very well in practice with the current state-of-the-art planners and benchmark problems.

Future Work

An important topic for future work is finding new and better methods for action ranking, which is a key component of our new $R^2\exists$-Step encoding. The methods which we proposed in this thesis are very simple and use only a part of the information available for the planning problem. We believe, that there exist much better methods of action ranking. Discovering such methods would immediately improve the performance of the $R^2\exists$-Step encoding.

A related open problem is the evaluation of action rankings, i.e., deciding whether a given ranking will allow us to solve the planning task quickly. The evaluation method should be fast to be useful.

Another potentially promising direction for future research might be the incorporation of the Counter-example Guided Abstraction Refinement (CEGAR) [15] techniques into the area of planning as SAT.

As for the post-planning optimization, it would be interesting to modify our encoding to allow replacing and/or adding actions into the plan. To maintain the high performance of the method these replacement/addition operations should be limited by some parameter, which would gradually increase and allow more modifications. This idea is inspired by the plan neighborhood graph search method [30].

Bibliography

[1] Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11:625–656, 1995.

[2] Tomáš Balyo. Relaxing the relaxed exist-step parallel planning semantics. In *ICTAI*, pages 865–871. IEEE, 2013.

[3] Tomáš Balyo, Roman Barták, and Pavel Surynek. Shortening plans by local re-planning. In *ICTAI*, pages 1022–1028. IEEE, 2012.

[4] Tomáš Balyo, Roman Barták, and Daniel Toropila. Two semantics for step-parallel planning: Which one to choose? *Proceedings of the 29th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2011)*, pages 9–15, 2011.

[5] Tomáš Balyo and Lukáš Chrpa. Eliminating all redundant actions from plans using sat and maxsat. *Proceedings of Knowledge Engineering for Planning and Scheduling (KEPS), To appear*, 2011.

[6] Tomáš Balyo and Lukáš Chrpa. On different strategies for eliminating redundant actions from plans. *Proceedings of The Seventh Annual Symposium on Combinatorial Search (SOCS), Under review*, 2014.

[7] Tomáš Balyo, Andreas Froehlich, Marijn Heule, and Armin Biere. Everything you always wanted to know about blocked sets (but were afraid to ask). *to appear in SAT 2014*, 2014.

[8] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[9] Armin Biere. Lingeling and plingeling home page. http://fmv.jku.at/lingeling/, May 2014.

[10] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[11] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[12] Tom Bylander. The computational complexity of propositional strips planning. *Artif. Intell.*, 69(1-2):165–204, 1994.

[13] Lukáš Chrpa, Thomas Leo McCluskey, and Hugh Osborne. Determining redundant actions in sequential plans. In *Proceedings of ICTAI*, pages 484–491, 2012.

[14] Lukáš Chrpa, Thomas Leo McCluskey, and Hugh Osborne. Optimizing plans through analysis of action dependencies and independencies. In *Proceedings of ICAPS*, pages 338–342, 2012.

[15] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[16] Amanda Jane Coles, Andrew Coles, Angel García Olaya, Sergio Jiménez Celorrio, Carlos Linares López, Scott Sanner, and Sungwook Yoon. A survey of the seventh international planning competition. *AI Magazine*, 33(1), 2012.

[17] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971.

[18] Sam J. Estrem and Kurt D. Krebsbach. Airs: Anytime iterative refinement of a solution. In *Proceedings of FLAIRS*, pages 26–31, 2012.

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

[20] Eugene Fink and Qiang Yang. Formalizing plan justifications. In *In Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence*, pages 9–14, 1992.

[21] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research (JAIR)*, 20:239 – 290, 2003.

[22] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004.

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

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

[25] Joerg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[26] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A novel transition based encoding scheme for planning as satisfiability. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[27] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[28] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI 92: Tenth European Conference on Artificial Intelligence*, pages 359–363, Vienna, Austria, 1992.

[29] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver. *JSAT*, 8(1/2):95–100, 2012.

[30] Hootan Nakhost and Martin Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *ICAPS*, pages 121–128. AAAI, 2010.

[31] Jussi Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.

[32] Jussi Rintanen. Planning as satisfiability: state of the art. http://users.cecs.anu.edu.au/ jussi/satplan.html, July 2013.

[33] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006.

[34] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. Sat-based parallel planning using a split representation of actions. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*. AAAI, 2009.

[35] Masahiro Sakai. Toysolver home page. https://github.com/msakai/toysolver, May 2014.

[36] Fazlul Hasan Siddiqui and Patrik Haslum. Plan quality optimisation via block decomposition. In Francesca Rossi, editor, *IJCAI*. IJCAI/AAAI, 2013.

[37] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Inf.*, 6:171–185, 1976.

[38] Martin Wehrle and Jussi Rintanen. Planning as satisfiability with relaxed exist-step plans. In Mehmet A. Orgun and John Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 244–253. Springer, 2007.

[39] Carl Henrik Westerberg and John Levine. Optimising plans using genetic programming. In *Proceedings of ECP*, pages 423–428, 2001.