Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)

Encoding Lifted Classical Planning in Propositional Logic

Daniel Höller[1] and **Gregor Behnke**[2,3]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany,
[2] University of Freiburg, Freiburg, Germany,
[3] University of Amsterdam, ILLC, The Netherlands
hoeller@cs.uni-saarland.de, g.behnke@uva.nl

Abstract

Planning models are usually defined in lifted, i.e. first order formalisms, while most solvers need (variable-free) grounded representations. Though techniques for grounding prune unnecessary parts of the model, grounding might – nevertheless – be prohibitively expensive in terms of runtime. To overcome this issue, there has been renewed interest in solving planning problems based on the lifted representation in the last years. While these approaches are based on (heuristic) search, we present an encoding of lifted classical planning in propositional logic and use SAT solvers to solve it. Our evaluation shows that our approach is competitive with the heuristic search-based approaches in satisficing planning and outperforms them in a (length-)optimal setting.

1 Introduction

Planning models are usually defined in a lifted way on some kind of (usually function-free) first order language. While there are methods to solve such problems based on this representation (e.g. Penberthy and Weld (1992), Younes and Simmons (2003), or Ridder and Fox (2014)), by far most work in the last decades has been based on grounded (i.e. variable-free) models. The lifted model is therefore transformed by a process called *grounding*, which systematically replaces variables by all constants. To make this feasible, models usually incorporate typing and grounding systems apply techniques to exclude model parts for which they can show that they cannot be contained in any solution (e.g. delete-relaxed reachability). However, while this works well in practice on certain models, it cannot prevent a blowup of the model that might be exponential in the worst case.

Recently, several approaches have been presented that solve planning problems based on the lifted model. These systems are based on (heuristic) search. They maintain the input model lifted, but ground the explored parts of the search space during search. Corrêa et al. (2020) introduced an approach to generate successor states in this process efficiently. Heuristics to guide the search have been presented by Corrêa et al. (2021) and Lauer et al. (2021). The former computes the Add and the Max heuristic (Bonet and Geffner 2001) on the lifted model. It uses an approach based on datalog (just like the grounded computation integrated in the Fast Downward system (Helmert 2006)). While the input model is not pre-grounded, those parts that are reached during heuristic calculation are grounded by the heuristic function, making the computation hard (NP-hard in the lifted input size in worst case). Lauer et al. (2021) presented a relaxation that splits predicates into smaller ones of a fixed arity. When choosing arity *one*, their heuristic can be computed in polynomial time in the lifted input size. More preliminary work has been presented by Wichlacz, Höller, and Hoffmann (2021). They extract landmarks from the lifted model, which are then used to guide the search.

In this paper we present a schema to encode lifted classical planning in a state-less way inspired by approaches from plan space planning. We realize a translation to propositional logic to exploit the performance of modern SAT-solvers. While the encoding is quadratic in plan length in the worst case, we present a strategy to decrease its size in satisficing planning, which proves very effective in practice. We evaluate our resulting system on a benchmark set recently introduced by Corrêa et al. and Lauer et al. to test lifted systems. The results show that our system has a slightly higher normalized coverage than the systems from the literature in the satisficing setting, and that it has three times the normalized coverage in a length-optimal setting.

2 Formal Framework

We base our formalism on the one by Lauer et al. (2021). A lifted planning problem is a tuple $\Pi = (\mathcal{O}, \mathcal{P}, \mathcal{A}, \mathcal{I}, \mathcal{G})$. $\mathcal{O}$ is a set of objects, $\mathcal{P}$ a set of predicate symbols, and $\mathcal{A}$ a set of action symbols (all sets are finite). Each predicate symbol $P^n \in \mathcal{P}$ has a tuple of parameters $P(v_1, \ldots, v_n)$. As in this example, we annotate predicates with their arity (here n) when the parameters are not given explicitly. When relevant for a definition, we annotate each variable with a type (e.g. $v_i^{t_i}$). Let $\mathcal{O}^t$ be the subset of $\mathcal{O}$ of the type t. We assume a type hierarchy that is a tree[1] in which for a parent type tp and a child type tc holds that $\mathcal{O}^{tp} \supseteq \mathcal{O}^{tc}$. In particular, we assume that if $\mathcal{O}^{t_1} \cap \mathcal{O}^{t_2} \neq \emptyset$, then either t_1 is a parent-type of t_2 or vice versa. Let $\mathcal{T}$ be the set of types. A parameter $v_i^{t_i}$ of a predicate can be substituted by an object from $\mathcal{O}^{t_i}$. When all variables are substituted we call the predicate *ground* (also called *atom*).

[1] W.l.o.g. we assume a *single* tree to simplify our definitions.

Similar to predicates, an action symbol $A^n \in \mathcal{A}$ comes with a tuple of n typed parameters $A(v_1^{t_1}, \ldots, v_n^{t_n})$. The functions $prec$, add, and del map an action to its precondition, the add, and the delete effect. Each of these functions f maps an action $A^n \in \mathcal{A}$ to a set $f(A^n) = \{P(i_1, \ldots, i_m) \mid P^m \in \mathcal{P}, 1 \leq j \leq m : i_j \in \{1, \ldots, n\}\}$. I.e., each i_j is an index of a variable from the action.

Example 1. *Consider a* pickup *action from a simple transport domain: $Pick(x^v, y^l, z^p)$, where x is of type* vehicle*, y of type* location*, and y of type* package*. The precondition is $prec(Pick^3) \mapsto \{At(3,2), At(1,2)\}$, i.e., the* At *predicate must hold for the tuple consisting of the 3^{rd} and 2^{nd} parameter of the action (i.e. for $At(z,y)$), and for the 1^{st} and 2^{nd} parameter ($At(x,y)$), i.e., vehicle and package are at the same location. The effect is $add(Pick^3) \mapsto \{In(3,1)\}$ and $del(Pick^3) \mapsto \{At(3,2)\}$, the package is inside the vehicle.*

Let $A(v_1^{t_1}, \ldots, v_n^{t_n})$ be an action. We call it to be *well-defined* if for all elements $P(i_1, \ldots, i_m)$ in its preconditions and effects with $P(u_1^{s_1}, \ldots, u_m^{s_m}) \in \mathcal{P}$, it holds that $\mathcal{O}^{t_{i_j}} \subseteq \mathcal{O}^{s_j}$. I.e., the type of the action's variable must be a subtype (i.e., mapping to the same or a subset of the objects) of the variable type of the predicate. In the following, we assume that all actions in a problem definition are well-defined.

Similar to predicates, an action is *ground* when each of its variables $v_i^{t_i}$ is substituted by an object out of $\mathcal{O}^{t_i}$. We define a function ρ, which maps a ground action $A(o_1, \ldots, o_n) \in \mathcal{A}$ and a set $P' = \{P(i_1, \ldots, i_m) \mid P \in \mathcal{P}, i_j \in \{1, \ldots, n\}\}$ to the set $\{P(o_{i_1}, \ldots, o_{i_m}) \mid P(i_1, \ldots, i_m) \in P'\}$. I.e. it replaces indices in a precondition or effect definition by the objects from the action, resulting in a set of atoms.

Let $\mathcal{P}^{\mathcal{O}}$ and $\mathcal{A}^{\mathcal{O}}$ be the sets of ground actions and predicates over $\mathcal{O}$. $\mathcal{I}$ is the initial state, and $\mathcal{G}$ the goal definition; both are elements out of $2^{\mathcal{P}^{\mathcal{O}}}$. A state is represented by the atoms that hold in it ($s \in 2^{\mathcal{P}^{\mathcal{O}}}$), all other atoms do not hold.

Definition 1 (Applicability)**.** *A ground action a is applicable in a state s if and only if $\rho(a, prec(a)) \subseteq s$.*

Definition 2 (State Transition)**.** *The state s' resulting from applying an applicable ground action is denoted $\gamma(s,a)$ and defined as $\gamma(s,a) = (s \setminus \rho(a, del(a))) \cup \rho(a, add(a))$.*

Definition 3 (Solution)**.** *A solution to a problem Π is 1. a sequence of ground actions $\pi = (a_1, \ldots, a_n)$ such that 2. a_i is applicable in s_{i-1} with $s_0 = \mathcal{I}$ as well as $s_i = \gamma(s_{i-1}, a_i)$ for $i > 0$; and 3. for s_n it holds that $\mathcal{G} \subseteq s_n$.*

3 Lifted Planning in Propositional Logic

We first want to give an intuition of our encoding. It does not include intermediate states of a plan, only the initial state and an action sequence, which has strong similarities with plan-space planning (see Ghallab, Nau, and Traverso 2004, Ch. 5), where systems maintain a partially ordered set of actions, the *partial plan*, during search (resulting in the lack of a full state definition at this point). E.g. in POCL planning (Penberthy and Weld 1992), action preconditions are matched to an effect fulfilling them by *causal links*, while actions that delete such an atom are called *threats* for the link. Solutions are then not defined via state, but via the absence of *flaws*, where a flaw is either a precondition not linked, or a threat.

We will see that our encoding has much in common with this view on planning. However, the actual solving techniques differ severely. We use a translation instead of search, have no causal links, and commit to a totally ordered plan.

We will discuss the relation of our approach to other translations from classical planning to propositional logic in Section 4 after introducing our encoding.

3.1 A High-Level Description of Our Encoding

Our encoding is illustrated in Figure 1. First have a look at the sequence at the top that resembles a solution, i.e., a sequence of ground actions. Each box is a placeholder for an action symbol (gray) or an object (white). Each action is followed by n objects, where n is the maximum arity of an action from Π. We now define the constraints needed to make this sequence a solution to a given planning problem.

In the encoding we assign each placeholder a *value*. For actions, this is a number from 1 to $|\mathcal{A}|$. We assume an arbitrary but fixed ordering of the actions $\mathcal{A} = (a_1, \ldots, a_{\mathcal{A}})$. Let I be a function mapping the index of an action in the solution to its placeholder, i.e. $I(i)$ is the placeholder at index $(i-1) \times (n+1)$. We define I also for parameter j of action i, mapping $I(i,j)$ to the placeholder at position $(i-1) \times (n+1) + j$ (both i and j start with 1). In our encoding, the placeholder $I(i)$ will (at first) be an integer variable. As such, we write $I(i) = a$ to denote the fact that variable $I(i)$ is set to action $a \in \mathcal{A}$ and $I(i) = I(j)$ to denote that the values of the two placeholders need to be identical. We use the same semantics for the object placeholders, where $I(i,j) = o$ indicates that the j^{th} parameter of the i^{th} action is set to o.

Typing We first enforce that the types of the parameters fit the action symbol. We assign every object from the problem a unique number from $1 \ldots |\mathcal{O}|$ in a way such that objects belonging to a certain type get consecutive numbers. This is possible since we assert the type hierarchy to be a tree. For a type t, let $fi(t)$ be the smallest index of an object belonging to t, and $li(t)$ the largest one. Let L be the length of the action sequence as given in Figure 1.

$$\forall i \in \{1, \ldots, L\} : \forall A(v_1^{t_1}, \ldots, v_m^{t_m}) \in \mathcal{A} : \forall j \in \{1, \ldots, m\}$$
$$(I(a_i) = A) \Rightarrow (I(i,j) \geq fi(t_j))$$
$$(I(a_i) = A) \Rightarrow (I(i,j) \leq li(t_j))$$

When solving this constraint system, we end up with a sequence of action symbols followed by their parameters of the correct type. When an action has less than n parameters (let us assume k with $k < n$), we have no guarantee about the placeholders at the positions between k and n. However, these are not of interest when reconstructing the sequence of actions and we get the following lemma:

Lemma 1. *From a fulfilling assignment we can extract a ground action sequence $\pi = (a_1, \ldots, a_L)$.*

Preconditions Consider the (partial) action definitions given at the bottom of Figure 1. On the right is the *Pickup* action from Ex. 1, on the left a *Drive* action (for the former, only preconditions are given, for the latter only effects).

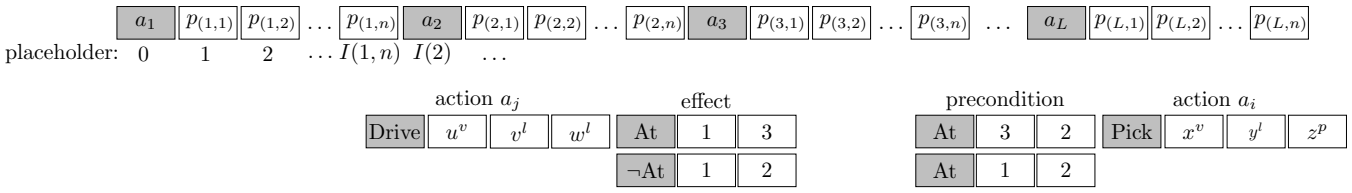

Figure 1: Schema of our encoding.

The *Pickup* action needs the *At* predicate to hold for the action parameter tuples $(3, 2)$ and $(1, 2)$, which translates into the vehicle (param. 1) and the package (param. 3) to be at the same location (param. 2). Consider the effect of the *Drive* action: the tuple of param. 1 (a vehicle) and 3 (a location) is added, while the tuple 1 and 2 (a location) is deleted. The first effect is a possible *achiever* for the second precondition of *Pickup*. An achiever is defined as follows:

Definition 4 (Achiever)**.** *An* achiever *for a (ground) atom $P(o_0, o_1, \ldots, o_m)$ in a (ground) plan $\pi = (a_1, \ldots, a_L)$ is*

1. *$\mathcal{I}$ if and only if it holds that $P(o_0, o_1, \ldots, o_m) \in \mathcal{I}$,* ***or***
2. *an action $a \in \pi$ with $P(o_0, o_1, \ldots, o_m) \in \rho(a, add(a))$.*

We define the term *position of the achiever* w.r.t. a sequence of actions as 0 (in the first case), or as the position of the action in the sequence (with 1 for the first action).

We know that the effect holds after executing the achiever. However, it might be deleted by another action. We call such an action a *destroyer*.

Definition 5 (Destroyer)**.** *A* destroyer *for a (ground) atom $P(o_0, o_1, \ldots, o_m)$ in a (ground) plan $\pi = (a_1, \ldots, a_L)$ is an action $a \in \pi$ that has it in its delete effect, i.e., $P(o_0, o_1, \ldots, o_m) \in \rho(a, del(a))$.*

The *position of the destroyer* w.r.t. a sequence of actions is its position in the sequence (with 1 for the first action).

In our example we can observe that we do not need to assign all parameters of the *Drive* action to know that it is an achiever for the precondition of our *Pickup* action. We need to set the param. 1 and 3 since only those are contained in effect 1. This makes perfect sense since we do not care *from* which location the vehicle comes, but where it is after the execution. Our second observation is that we do not need to pick the actual constants we assign to the parameters, it suffices to ensure that the respective parameters are *equal*. I.e., to check whether there is an achiever for precondition 2 of action a (the *Pick* action) we do not need to track the state nor to fully instantiate the achiever action's parameters. It is sufficient to check whether there is some j such that:

$$[I(j) = drive] \wedge [I(j, 1) = I(i, 1)] \wedge [I(j, 3) = I(i, 2)]$$

Definition 6 (Achiever Assignment Set)**.** *Given an action a and an element $P(i_1, \ldots, i_m)$ out of its precondition $prec(a)$. The function $\alpha(a, P(i_1, \ldots, i_m))$ maps the two elements to the following set:*

$$\alpha(a(v_1^{t_1}, \ldots, v_l^{t_l}), P(i_1, \ldots, i_m)) = \{(a', (j_1, \ldots, j_m)) \mid a'(u_1^{s_1}, \ldots, u_k^{s_k}) \in \mathcal{A}, P(j_1, \ldots, j_m) \in add(a'), \text{ for } 1 \leq h \leq m : \mathcal{O}^{t_{i_h}} \cap \mathcal{O}^{s_{j_h}} \neq \emptyset\}$$

The function maps one element of an action's precondition to a set of tuples containing (1) an action a' having the same predicate in its add effect (the achiever), and (2) the indices j_i from the action appearing in that effect. Be aware that these are just *indices* of the action's parameters. When we enforce parameter indexed j_1 of a' to be equal to the first parameter used in a's precondition, parameter j_2 of a' equal to the second parameter and so on, we know that a' is an achiever for the precondition. Notice that neither precondition nor effect are included in this equality tests (it is a test on parameters). We define the same for destroyers:

Definition 7 (Destroyer Assignment Set)**.** *Given an action a and an element $P(i_1, \ldots, i_m)$ out of its precondition $prec(a)$. The function $\beta(a, P(i_1, \ldots, i_m))$ maps the two elements to the following set:*

$$\beta(a(v_1^{t_1}, \ldots, v_l^{t_l}), P(i_1, \ldots, i_m)) = \{(a', (j_1, \ldots, j_m)) \mid a'(u_1^{s_1}, \ldots, u_k^{s_k}) \in \mathcal{A}, P(j_1, \ldots, j_m) \in del(a'), \text{ for } 1 \leq h \leq m : \mathcal{O}^{t_{i_h}} \cap \mathcal{O}^{s_{j_h}} \neq \emptyset\}$$

Definition 8 (Achiever/Destroyer Assignm. Set on Goals)**.** *We use analogous functions for $P(o_1, \ldots, o_n) \in \mathcal{G}$.*

$$\alpha/\beta(P(o_1, \ldots, o_m)) = \{(a, (j_1, \ldots, j_m)) \mid a(u_1^{s_1}, \ldots, u_k^{s_k}) \in \mathcal{A}, P(j_1, \ldots, j_m) \in add/del(a), \text{ for } 1 \leq h \leq m : o_h \in \mathcal{O}^{s_{j_h}}\}$$

To enforce the preconditions of all actions in the sequence, we include the constraints given in Figure 2. For each combination of placeholder i_c and action $A \in \mathcal{A}$, there is one implication. When the placeholder is equal to A, for each precondition there must be an index i_a containing an achiever, such that there is no destructor between i_a and i_c.

Lemma 2. *For each precondition $P(o_0, o_1, \ldots, o_m) \in \rho(a_i, prec(a_i))$ of each action $a_i = A(v_0, v_1, \ldots, v_L)$ in the action sequence extracted from the fulfilling assignment, the following holds:*

1. *There is an achiever for $P(o_0, o_1, \ldots, o_m)$ at some position j with $0 \leq j < i$ such that*
2. *there is no destroyer between j and i.*

Goal Condition The goal condition is defined in a similar way, given in Figure 3.

Lemma 3. *In the action sequence extracted from the fulfilling assignment, in each $P(o_0, o_1, \ldots, o_m) \in \mathcal{G}$:*

1. *There is an achiever for $P(o_0, o_1, \ldots, o_m)$ at some position j with $0 \leq j \leq L$ such that*
2. *there is no destroyer between j and L.*

From the combination of Lem. 1–3, we get the following:

$$\forall i_c \in \{1,\dots,L\} : \forall A \in \mathcal{A} : (I(i_c) = A) \Rightarrow \forall P^o(k_1,\dots k_m) \in prec(A) : \exists i_a : (0 \le i_a < i_c) :$$
$$\big((i_a = 0) \Rightarrow [\rho(A,\{P^o\}) \subset \mathcal{I}] \wedge (i_a \ge 1) \Rightarrow \big[\bigvee_{\substack{(A',(j_1,\dots,j_m))\\ \in\alpha(A,P^o)}} (I(i_a) = A') \wedge \bigwedge_{l=1}^{m} (I(i_c,k_l) = I(i_a,j_l))\big]\big) \wedge$$
$$\neg\exists i_d : (i_a < i_d < i_c) : \big[\bigvee_{\substack{(A',(j_1,\dots,j_m))\\ \in\beta(A,P^o)}} (I(i_d) = A') \wedge \bigwedge_{l=1}^{m} (I(i_c,k_l) = I(i_d,j_l))\big]$$

Figure 2: Constraints to enforce that all preconditions have achievers.

$$\forall P(o_1,\dots,o_n) \in \mathcal{G} : \exists i_a : (0 \le i_a \le L) : \big((i_a = 0) \Rightarrow [P(o_1,\dots,o_n) \in \mathcal{I}] \wedge$$
$$(i_a \ge 1) \Rightarrow \big[\bigvee_{\substack{A\in\mathcal{A},P(j_1,\dots,j_m)\\ \in\alpha(A,add(A))}} (I(i_a) = A) \wedge \bigwedge_{l=0}^{m} (o_l = I(i_a,j_l))\big]\big) \wedge \neg\exists i_d : (i_a < i_d \le L) : \big[\bigvee_{\substack{A\in\mathcal{A},P(j_1,\dots,j_m)\\ \in\beta(A,del(A))}} (I(i_d) = A) \wedge \bigwedge_{l=1}^{m} (o_l = I(i_d,j_l))\big]$$

Figure 3: Constraints to enforce goal achievers.

Theorem 1 (Correctness)**.** *The plan extracted by our encoding is a plan for the underlying planning problem.*

Proof. We need to show that we receive a sequence of ground actions that is applicable and leads to a goal. Assume that there is an action a_i that is not applicable though Lem. 1–3 hold. To be *not* applicable, there must be at least one atom contained in its preconditions not contained in the state right before the execution (see Def. 1). Let p be such an atom. According to Lem. 2.1, there is an achiever for p. Thus it was true right after the achiever has been executed (or in $\mathcal{I}$ if the achiever has position 0). The only means by which p can be false before a_i is by a delete effect of an action between the achiever's position j and i. Such an action would be a destroyer violating Lem. 2.2, which does not exist. The same reasoning holds for the goal. □

Theorem 2 (L-Completeness)**.** *Given π is a plan according to Def. 3 of length $L = |\pi|$ for problem Π, there is a satisfying assignment of our encoding.*

Proof. Similarly, assume a plan of length L satisfying Def. 3 but not Lem. 1–3. There must be some action a_i (or the goal) and precondition p of a_i that is not fulfilled. Since a_i is applicable in s_i, $p \in s_i$, it must have been in the initial state or been inserted by some action without a subsequent delete effect. Both cases would constitute the required achiever, leading to a contradiction. □

Theorem 3 (Size)**.** *Given a (lifted) planning problem Π, our encoding is quadratic in the length of the plan.*

Proof Sketch. The size is determined by the interplay of an action sequence of length L and the positions of possible achievers and destroyers. For an action a at position i, there are $j = i-1$ possible achievers. We can encode the selection of the achiever for a at position i in linear size.

The destroyers are critical as there are $k = i - j$ possible destroyers for every pair i, j. Whether a concrete action is a destroyer does not depend on the *exact* achiever position, but only on whether it is before the destroyer or not. As such, we can encode for every position k whether the achiever of a at i is before k – which is possible with a linear encoding. We then encode for every position k that its action is a destroyer if the achiever for any $i > k$ is before k. This leads to a quadratic encoding. For more details, we refer to the following encoding and the detailed proof in the appendix. □

3.2 An Encoding in Propositional Logic

We presented a schema for a state-less encoding that could be realized in different formalisms. In this paper we present a realization in propositional logic, which is described in this section. Other variants could e.g. use Integer Linear Programming or CSP. We create the formula for a given number of positions, denoted with L (the length of the plan).

We underline names of propositional variables to make formulae as clear as possible. Throughout this section, we generate parts of the formula for *every position* $p \in \{1,\dots,L\}$. To ease notation we omit the quantification.

We first define variables representing the placeholders:

- $\underline{a_i @ p}$ – for every $a_i \in \mathcal{A}$ and position $1 \le p \le L$.
- $\underline{o_i @ p, j}$ – for every $o_i \in \mathcal{O}$, position $1 \le p \le L$, and $1 \le j \le n$ where n is the maximum arity of an action.

These variables represent every possible value of each placeholder with a separate variable (one-hot encoding). To enforce this semantics, we have to assert that for every position $1 \le p \le L$ and argument index $1 \le j \le n$ at most one of the $\underline{a_i @ p}$ and $\underline{o_i @ p, j}$ is true. There are several ways of encoding such at-most-one constraints. For a set of variables V, we denote with $\mathbb{A}(V)$ a set of clauses that pose the constraint that at most one variable in V is true. If $|V| < 256$ we use the naïve quadratic encoding, for larger sets V we use the binary counter encoding (Sinz 2005). We postpone providing the exact constraints until after discussing typing.

Typing Next, we assure the correct parameter typing for the different actions. We do not encode typing constraints

directly, but we encode *more* object placeholders than the maximum arity n and consider the maximum arity separately per type. For every type t, we determine the maximum number n_t of parameters of this type an action has.

Example 2. *Consider a planning problem with the actions* $Pick(x^v, y^l, z^p)$, $Drop(x^p, y^l, z^v)$, *and* $Drive(x^l, y^v, z^l)$. *In this case, we have a maximum arity* $n = 3$ *and type arities of* $n_v = 1$, $n_p = 1$, *and* $n_l = 2$.

We then generate $N = \sum_{t \in \mathcal{T}} n_t$ many placeholders per position. For a fixed but arbitrary ordering of the types $(t_1, \ldots, t_t)$, we assign the first n_{t_1} placeholders to arguments of type t_1, the next n_{t_2} placeholders to arguments of type t_2 and so on. By this, we obtain a new mapping $\tau : \mathcal{A} \times \mathbb{N} \mapsto \mathbb{N}$ assigning every argument of an action to the index of its placeholder. Let $T : \mathbb{N} \mapsto \mathcal{T}$ define the type of each placeholder.

Example 3. *Continuing Example 2, we create* 4 *placeholders and set* $T(1) = v$, $T(2) = l$, $T(3) = l$, $T(4) = p$, *and* $\tau(Pick, 1) = 1$, $\tau(Pick, 2) = 2$, $\tau(Pick, 3) = 4$, $\tau(Drop, 1) = 4$, $\tau(Drop, 2) = 2$, $\tau(Drop, 3) = 1$, $\tau(Drive, 1) = 1$, $\tau(Drive, 2) = 1$, *and* $\tau(Drive, 3) = 3$.

We increase the number of placeholders, but eliminate variability in the order of arguments of actions, by grouping arguments with (ostensibly) similar semantics together. Further, the typing of object placeholders is now independent of the actually chosen action. This reduction will also be of use when encoding the equality of parameters. To ensure a correct typing and unique selection of actions and parameters, we add the following constraints to our formula, which ensure that of the possible values for each placeholder at most one is picked and no impossible value is chosen.

$$\mathbb{A}(\{\underline{a_i@p} \mid a_i \in \mathcal{A}\})$$
$$\forall j \in \{1, \ldots, N\} : \mathbb{A}(\{\underline{o_i@p, j} \mid o_i \in \mathcal{O}^{T(j)}\})$$
$$\forall j \in \{1, \ldots, N\} : \forall o \in \mathcal{O} \setminus \mathcal{O}^{T(j)} : \neg\underline{o@p, j}$$

For the time being, we also enforce that every position actually contains an action, i.e. when generating the formula for L positions, we can only find plans that contain exactly L actions. We therefore add the formula: $\bigvee_{a_i \in \mathcal{A}} \underline{a_i@p}$.

Equality For implementing the formula in Figure 2, we need to be able to refer to the equality of objects assigned to two parameter placeholders. Since we will use this construct frequently, we describe it now, prior to the actual implementation of preconditions and effects. We introduce propositional variables expressing equality as follows

- $\underline{b, i = p, j}$ – for each pair of positions $1 \leq b < p \leq L$ and $1 \leq i, j \leq N$.

$\underline{b, i = p, j}$ shall be true if and only if the i^{th} placeholder at position b has the same value (i.e. object) as the j^{th} placeholder at position p. Like for p, we will not explicitly state the quantification over b to improve readability. We always generate the clauses for all $1 \leq b < p \leq L$.

$$\forall i, j \in \{1, \ldots, N\}, \forall o_k \in \mathcal{O} :$$
$$\underline{b, i = p, j} \wedge \underline{o_k@p, j} \rightarrow \underline{o_k@b, i}$$
$$\underline{b, i = p, j} \wedge \underline{o_k@b, i} \rightarrow \underline{o_k@p, j}$$
$$\underline{o_k@p, j} \wedge \underline{o_k@b, i} \rightarrow \underline{b, i = p, j}$$

We only generate variables $\underline{b, i = p, j}$ if they are used in encoding the preconditions and effects of actions. Empirically, we often only encode a small portion of the possible equals variables. Typically, only variables of the same type need to be compared (e.g. locations with locations). Here we benefit from our decision to group arguments of similar types together as arguments of the same type could occur at widely different positions in the argument list.

Preconditions Next, we implement the formula in Figure 2. This poses two main issues: (1) the existential quantifier and (2) the equality constraints in the formula. We compile (1) into additional decision variables. Let P be the max. number of preconditions that any action has. We assume that the preconditions $prec(a)$ of an action a form a list s.t. we can refer to the i^{th} precondition with $prec(a)[i]$ (1-indexed).

- $\underline{\widehat{b, i@p}}$ – for every $0 \leq b < p \leq L$ and $1 \leq i \leq P$.
- $\underline{b, i, a_k(l_1, \ldots, l_m)@p, a_j}$ – for every $1 \leq b < p \leq L$, $1 \leq i \leq P$, $a_j \in \mathcal{A}$ and $(a_k, (l_1, \ldots, l_m)) \in \alpha(a_j, prec(a_j)[i])$.

$\underline{\widehat{b, i@p}}$ shall be true if the action at b is the achiever for the i^{th} precondition of the action at p. We allow $b = 0$ to model that it is supported by $\mathcal{I}$. We add the following implications assuring that an achiever is picked for every precondition:

$$\forall a_j \in \mathcal{A}, \forall i \in \{1, \ldots, |prec(a_j)|\} : \underline{a_j@p} \rightarrow \bigvee_{b=0}^{p-1} \underline{\widehat{b, i@p}}$$

For the following steps, we have to distinguish multiple cases. We start with the easiest case: $b > 0$ (i.e. the achiever is an action) and $|\alpha(a_j, prec(a_j)[i])| = 1$ (i.e. there is only one possible achiever apart from init). In this case, if we choose $\underline{\widehat{b, i@p}}$ and $\underline{a_j@p}$ to be true, then this directly implies certain parameter equalities. We encode this as follows:

$$\forall a_j \in \mathcal{A}, \forall i \in \{1, \ldots, |prec(a_j)|\}$$
let $prec(a_j)[i] = P(r_1, \ldots, r_m)$
and $(a_k, (l_1, \ldots, l_m)) \in \alpha(a_j, prec(a_j)[i])$, then add:
$$\underline{\widehat{b, i@p}} \wedge \underline{a_j@p} \rightarrow \underline{a_k@b}$$
$$\forall q \in \{1, \ldots, m\} :$$
$$\underline{\widehat{b, i@p}} \wedge \underline{a_j@p} \rightarrow \underline{b, \tau(a_k, l_q) = p, \tau(a_j, r_q)}$$

In the next case, when $|\alpha(a_j, prec(a_j)[i])| > 1$, we first need to select an achiever with the following clauses.

$$\forall a_j \in \mathcal{A}, \forall i \in \{1, \ldots, |prec(a_j)|\}$$
$$\underline{\widehat{b, i@p}} \wedge \underline{a_j@p} \rightarrow \bigvee_{(a_k, (l_1, \ldots, l_m)) \in \alpha(a_j, prec(a_j)[i])} \underline{b, i, a_k(l_1, \ldots, l_m)@p, a_j}$$

We can then use the same encoding for enforcing the required parameter equality as in the previous case – we simply have to loop over all achievers and replace $\underline{\widehat{b, i@p}} \wedge \underline{a_j@p}$ with $\underline{b, i, a_k(l_1, \ldots, l_m)@p, a_j}$.

The last case to consider is the case $b = 0$, i.e. when $\mathcal{I}$ is chosen as achiever. Consider the case that the precondition $prec(a_j)[i] = P(r_1, \ldots, r_m)$ is supported by a fact

that is true in $\mathcal{I}$. Let $\mathcal{I}^P$ be the atoms whose predicate is P. If $\underline{\widehat{0, i@p}}$ is true, we have to check whether the arguments $r_1, \ldots, r_m$ correspond to one of the atoms in $\mathcal{I}^P$. We could easily encode this directly by stating that for one atom in $\mathcal{I}^P$ all parameter variables must have the required constants: $\underline{a_j@p} \wedge \underline{\widehat{0, i@p}} \rightarrow \bigvee_{P(o_1,\ldots,o_m)\in\mathcal{I}^P} \bigwedge_{k=1}^{m} \underline{o_k@p, \tau(a_j, r_k)}$. To transform this formula into CNF, we would need to introduce additional variables. Instead we use a different encoding that is both more compact in practice and yields better runtime. We encode that if the first $k-1$ parameters match to one of the facts in the $\mathcal{I}^P$, then the kth parameter must be one of the possible continuations, i.e. a constant s.t. a fact in $\mathcal{I}^P$ agreeing on the first $k-1$ parameters has this constant as its kth parameter. This includes the case $k = 1$, i.e. we require that the first parameter is chosen s.t. a fact in $\mathcal{I}^P$ exists that has this object as its first parameter. Via induction, we get that for a satisfying valuation, we indeed selected a fact from $\mathcal{I}^P$. We encode this in the following way:

$$\forall i \in \{1, \ldots, |prec(a_j)|\} : \text{let } prec(a_j)[i] = P(r_1, \ldots, r_m)$$
$$\forall k \in \{1, \ldots, m\}$$
$$\forall (o_1, \ldots, o_{k-1}) \in \{(o_1, \ldots, o_{k-1}) \mid P(o_1, \ldots, o_m) \in \mathcal{I}^P\}$$
$$\underline{a_j@p} \wedge \underline{\widehat{0, i@p}} \wedge \bigwedge_{l=1}^{k-1} \underline{o_l@p, \tau(a_j, r_l)} \rightarrow \bigvee_{o_k\in\{o_k \mid (o_1,\ldots,o_k,\ldots,o_m)\in\mathcal{I}^P\}} \underline{o_k@p, \tau(a_j, r_k)}$$

We encode the goal with the same mechanics as the preconditions. We introduce the following achiever selection variables, assuming that the goal is a list $\mathcal{G} = (g_1, \ldots, g_{|\mathcal{G}|})$.

- $\underline{\widehat{b, i@G}}$ – for every $1 \leq p \leq L$ and $i \in \{1, \ldots, |\mathcal{G}|\}$
- $\underline{b, i, a_j(l_1, \ldots, l_m)@G}$ – for every $1 \leq p \leq L$, $i \in \{1, \ldots, |\mathcal{G}|\}$, and $(a_j, (l_1, \ldots, l_m)) \in \alpha(g_i)$.

We select the achiever position $\forall g_i \in \mathcal{G} : \bigvee_{b=0}^{L} \underline{\widehat{b, i@G}}$. Then we select an achiever if the position has been chosen.

$$\forall g_i \in \mathcal{G} : \underline{\widehat{b, i@G}} \rightarrow \bigvee_{(a_j(l_1,\ldots,l_m))\in\alpha(g_i)} \underline{b, i, a_j(l_1, \ldots, l_m)@G}$$

We then ensure the correct effect of the achiever.

$$\forall g_i \in \mathcal{G} : \forall (a_j, (l_1, \ldots, l_m)) \in \alpha(g_i) : \text{let } g_i = P(o_1, \ldots, o_m) : \forall q \in \{1, \ldots, m\} : \underline{b, i, a_j(l_1, \ldots, l_m)@G} \rightarrow \underline{o^k@p, \tau(a_j, l_q)}$$

Lastly, we consider the support of a goal by the initial state and disallow the support, if the fact does not hold in init.

$$\forall g_i \in \mathcal{G} \setminus \mathcal{I} : \neg \underline{0, i@G}$$

Destroyers Lastly, we need to integrate destroyers into our encoding. A central objective is to ensure that it does not become cubic. Please keep in mind that the question of whether the effects of an action actually inhibit an achiever does not depend on the concrete position of the achiever action, but only on the fact that the achiever action is ordered *before* the destroyer (proof of Thm. 3). As such, we first introduce a new type of decision variable that encode that the achiever b for a given precondition i is before a given position p.

- $\underline{\overleftarrow{b, i@p}}$ – for every $1 \leq b < p \leq L$ and $1 \leq i \leq P$.

We exclude support for $b = 0$ and treat destroyers for facts in $\mathcal{I}$ separately. Next we define the following clauses $\forall i \in \{1, \ldots, P\}$: $(\underline{\overleftarrow{b, i@p}} \rightarrow \underline{\overleftarrow{b+1, i@p}}) \wedge (\underline{\widehat{b, i@p}} \rightarrow \underline{\overleftarrow{b+1, i@p}})$. Referring to $b+1$ in the right-hand-side equation accounts for the precedence of add over delete effects.

Next we have to consider under which criteria a potential destroyer actually inhibits an achiever. This is the case if the delete effect of the destroyer is the exact same atom as the precondition the achiever shall achieve. As such, we only need to compare with this precondition. We then formulate the criterion under which the inhibition occurs (which will be a conjunction) and add its negation (a disjunction) as a single clause to the formula:

$$\forall a_j \in \mathcal{A} \forall i \in \{1, \ldots, |prec(a_j)|\} :$$
$$\forall (a_k, (l_1, \ldots, l_m)) \in \beta(a_j, prec(a_j)[i]) :$$
$$\text{let } P(r_1, \ldots, r_m) = prec(a_j)[i]$$
$$\neg \underline{a_j@p} \vee \neg \underline{a_k@b} \vee \neg \underline{\overleftarrow{b, i@p}} \vee \bigvee_{q=1}^{m} \underline{\neg b, \tau(a_k, l_q) = p, \tau(a_j, r_q)}$$

Now we consider destroyers for atoms in $\mathcal{I}$. For them, we again introduce new decision variables.

- $\underline{\overrightarrow{P(o_1, \ldots, o_m)@p}}$ – for every $1 \leq p \leq L$ and $\underline{P(o_1, \ldots, o_m)} \in \mathcal{I}$.

$\underline{\overrightarrow{P(o_1, \ldots, o_m)@p}}$ shall be true if the fact $P(o_1, \ldots, o_m)$ which was initially true in init was false at least once, i.e. init cannot be used for support any more. We define this with the following clauses:

$$\forall P(o_1, \ldots, o_m) \in \mathcal{I} :$$
$$\underline{\overrightarrow{P(o_1, \ldots, o_m)@p}} \rightarrow \underline{\overrightarrow{P(o_1, \ldots, o_m)@p+1}}$$
$$\forall (a_k, (l_1, \ldots, l_m)) \in \beta(P(o_1, \ldots, o_m)) :$$
$$\underline{a_k@p} \wedge \bigwedge_{q=1}^{m} \underline{o_q@p, \tau(a_k, l_q)} \rightarrow \underline{\overrightarrow{P(o_1, \ldots, o_m)@p}}$$

If a fact from init has been destroyed at some point, it cannot be used as an achiever any more:

$$\forall i \in \{1, \ldots, |prec(a_j)|\} : \text{let } prec(a_j)[i] = P(r_1, \ldots, r_m)$$
$$\forall P(o_1, \ldots, o_m) \in \mathcal{I}^P : \neg \underline{a_j@p} \vee \neg \underline{\widehat{0, i@p}} \vee \neg \underline{\overrightarrow{P(o_1, \ldots, o_m)@p-1}} \vee \bigvee_{q=1}^{m} \neg \underline{o_q@p, \tau(a_j, r_q)}$$

We omit this part of the encoding if $\beta(P(o_1, \ldots, o_m)) = \emptyset$.

Lastly, we have to consider destroyers for goal conditions. Here we use a direct, quadratic encoding. Note that the formula also handles a goal supported by $\mathcal{I}$ for $b = 0$.

$$\forall g_i \in \mathcal{G} : \forall (a_j, (l_1, \ldots, l_m)) \in \beta(g_i) : \text{let } g_i = P(o_1, \ldots, o_m) : \neg \underline{a_j@p} \vee \neg \underline{\widehat{b, i@G}} \vee \bigvee_{q=1}^{m} \neg \underline{o_q@p, \tau(a_j, l_q)}$$

A formula resulting from the translation of an example problem can be found in the appendix.

Algorithm 1: Length-Optimal Planning

for $l \in [0, 1, \ldots]$ **do**
  $p :=$ translation with length bounded by l
  if *SAT(p)* **then return** solved

3.3 From the Encoding to a Planning System

Since we presented a translation bounded in the plan length L, L is an obvious parameter to set before translation. Our first configuration starts with $L = 0$ and increases L by 1 when the resulting formula is proven unsolvable. This results in length-optimal solutions, which is cost-optimal if all actions have cost 1. The pseudocode is given in Alg. 1.

Besides L there is a second, less obvious parameter that proves very powerful in practice. The achiever/consumer structure seems to be very "local" in practice, i.e., the distance between consumer and achiever in a solution is very small. We exploit this insight by limiting the distance d between an action and the achievers fulfilling its precondition in our encoding, which makes it more compact. With $d = 1$, every precondition of an action a must be fulfilled by $\mathcal{I}$ or the action directly before a. While this is an extreme example, there are domains in the benchmark set where this suffices to solve the problems. In general, our experiments show that we at least do not need to include all actions back to $\mathcal{I}$ as possible achievers to solve many problems.

To implement this restriction to a distance d, we modify the selection of the achiever position to the following:

$$\forall a_j \in \mathcal{A} \forall i \in \{1, \ldots, |prec(a_j)|\} :$$

$$\underline{a_j@p} \rightarrow \widehat{\underline{0, i@p}} \vee \bigvee_{b=\max\{1,d-p\}}^{p-1} \widehat{\underline{b, i@p}}$$

We further do not generate clauses pertaining to achiever selection variables $\widehat{\underline{b, i@p}}$ that are not part of this disjunction anymore. This notably includes variables $\underline{b, i = p, j}$ expressing equality for $p-b > d$ and destroyer clauses for $p-b > d$. We further remove the constraint that every position must contain an action and only enforce that if a position contains an action, the previous position must as well.

$$\forall a_i \in \mathcal{A} \underline{a_i@p} \rightarrow \bigvee_{a_j \in \mathcal{A}} \underline{a_j@p-1}$$

The pseudocode is given in Alg. 2. We fixed a set of length bounds L. On each length we spend a slice of the overall runtime and start with a distance $d = 1$; until the time is up, we increase d. Using this configuration, we cannot provide a guarantee regarding optimality anymore, of course.

4 Related Work

Our approach is of course not the first SAT-based system in planning. However, most recently introduced encodings assume a grounded input, e.g., the original SAT planner (Kautz and Selman 1996), SATPLAN04 (Kautz and Selman 2006), Madagascar (Rintanen, Heljanko, and Niemelä 2006; Rintanen 2014), Aquaplanning (https://github.com/domschrei/aquaplanning), or SASE (Huang, Chen, and Zhang 2012).

Algorithm 2: Satisficing Planning

$L := [10, 25, 50, 100, 200]$
for $l \in L$ **do**
  with limit of $\frac{1}{|rem.\ L\ values|} \times$ *rem. runtime* **do**
    for *max achiever distance* $d \in [1, \ldots, l]$ **do**
      $p :=$ translation with length bound l and achiever distance bounded by d.
      if *SAT(p)* **then return** solved

Robinson et al. (2009) propose an approach that does not require full grounding. They split up the action schemata and only partially ground actions. Correctness is ensured via intra-operator mutex axioms. Especially in domains with action with high arity this approach is helpful. However, they still represent the state explicitly and as such have to ground the state space. We contacted the authors to get the code for our evaluation, but it is not available anymore.

Kautz, McAllester, and Selman (1996) introduce a "causal encoding" that also uses a split action representation and does not explicitly encode state. The main difference to our work is that it allows for a partial order of actions, while we do not. Committing to a total order allows for a more compact encoding of destroyers (causal threats), which we can encode quadratic, while they need a cubic encoding. Further, the total order allows for the "maximum distance of achiever" encoding that we use for satisfying planning, which further decreases the (practical) size of the encoding. Their encoding of preconditions is atom-centric (they consider support/needs for a lifted atom) while we encode this index-centric (support for the i^{th} precondition). Kautz, McAllester, and Selman further use a more complex means to handle equality as they – via a special lifted SAT system – allow for equality testing over terms not only over variables as we do. Lastly, we do not introduce intermediate variables for the fact that a timestep produces, needs, or destroys a given first-order atom, but encode the dependencies directly – reducing the number of decision variables.

5 Evaluation

We integrated our approach in the search-based *Powerlifted* (PWL) system (Corrêa et al. 2020). In the following, we will call it LiSAT (Lifted SAT-based planner)[2]. We also created and evaluated a version using incremental solving and will present its results. However, it did not increase performance and we do not give a description here due to lack of space.

We use a benchmark set dedicated to lifted planning[3]. Experiments ran on a single Intel Xeon Gold 6242 CPU core (2.80GHz) with 4GB memory and 30 minutes runtime. We ran a satisficing and a length-optimal evaluation. For the non-incremental encoding we use the SAT solver Kissat 2.0.1 (winner of the 2020 SAT Comp., main track) (Biere et al. 2020), for incremental solving Crypto-

[2]Source code is available online https://lisat-planning.github.io/
[3]https://github.com/abcorrea/htg-domains

	Lifted Systems						
	LiSAT (time slices)			Powerlifted		Unary Relaxation	
	Kissat (n.i.)	CMS (i.)	CMS (n.i.)	Add+po	GC	GC, ur-d	GC, ur
Blocksworld 40	**100.0** (40)	**100.0** (40)	**100.0** (40)	10.0 (4)	2.5 (1)	15.0 (6)	15.0 (6)
Childsnack 144	**100.0** (144)	**100.0** (144)	**100.0** (144)	45.8 (66)	16.0 (23)	60.4 (87)	41.7 (60)
GED 312	36.5 (114)	24.4 (76)	33.7 (105)	79.8 (249)	**100.0** (312)	**100.0** (312)	**100.0** (312)
Logistics 40	**100.0** (40)	97.5 (39)	**100.0** (40)	**100.0** (40)	47.5 (19)	0.0 (0)	0.0 (0)
Organic synt. 56	**92.9** (52)	**92.9** (52)	**92.9** (52)	83.9 (47)	82.1 (46)	80.4 (45)	80.4 (45)
Pipesworld 50	42.0 (21)	38.0 (19)	40.0 (20)	**50.0** (25)	44.0 (22)	22.0 (11)	24.0 (12)
Rovers 40	10.0 (4)	7.5 (3)	10.0 (3)	**77.5** (31)	2.5 (1)	37.5 (15)	32.5 (13)
Visitall MD 180	**98.3** (177)	92.8 (167)	94.4 (170)	78.9 (142)	35.6 (64)	81.7 (147)	55.6 (100)
862	**579.7** (592)	553.0 (540)	568.5 (574)	526.0 (604)	330.2 (488)	396.9 (623)	349.1 (548)

	Grounded Systems			
	Fast Downward		MpC	
	Add	FF	inv.	no inv.
Blocksworld 40	20.0 (8)	20.0 (8)	10.0 (4)	0.0 (0)
Childsnack 144	41.0 (59)	72.2 (104)	45.8 (66)	45.8 (66)
GED 312	82.4 (257)	**100.0** (312)	37.2 (116)	16.7 (52)
Logistics 40	10.0 (4)	10.0 (4)	0.0 (0)	0.0 (0)
Organic synt. 56	32.1 (18)	32.1 (18)	0.0 (0)	0.0 (0)
Pipesworld 50	28.0 (14)	28.0 (14)	20.0 (10)	18.0 (9)
Rovers 40	10.0 (4)	10.0 (4)	0.0 (0)	0.0 (0)
Visitall MD 180	40.0 (72)	40.0 (72)	6.7 (12)	25.6 (46)
862	263.5 (436)	312.4 (536)	119.7 (208)	106.1 (173)

Table 1: Coverage results for satisficing planning. LiSAT configurations marked "n.i." use non-incremental SAT solving, those marked "i." use incremental solving. MpC configurations marked "no inv." do not use MpC's invariant analysis in preprocessing.

	Lifted Systems				
	LiSAT (optimal)			Powerlifted	
	Kissat (n.i.)	CMS (i.)	CMS (n.i.)	BFS	A* Max
Blocksworld 40	**100.0** (40)	**100.0** (40)	**100.0** (40)	0.0 (0)	0.0 (0)
Childsnack 144	**51.4** (74)	33.3 (48)	33.3 (48)	2.1 (3)	0.7 (1)
GED 312	**21.8** (68)	17.3 (54)	19.2 (60)	13.1 (41)	13.8 (43)
Logistics 40	**75.0** (30)	**75.0** (30)	67.5 (27)	12.5 (5)	5.0 (2)
Organic synt. 56	**100.0** (56)	98.2 (55)	98.2 (55)	76.8 (43)	76.8 (43)
Pipesworld 50	**40.0** (20)	34.0 (17)	**40.0** (20)	22.0 (11)	14.0 (7)
Rovers 40	**10.0** (4)	**10.0** (4)	7.5 (3)	0.0 (0)	2.5 (1)
Visitall MD 180	**57.2** (103)	56.1 (101)	55.6 (100)	18.3 (33)	37.2 (67)
862	**455.4** (395)	424.0 (349)	421.3 (353)	144.8 (136)	150.0 (164)

	Grounded Systems			
	Fast Downward		MpC	
	A* Max	A* LM-Cut	opt, (inv.)	opt, (no inv.)
Blocksworld 40	2.5 (1)	20.0 (8)	10.0 (4)	12.5 (5)
Childsnack 144	3.5 (5)	5.6 (8)	0.7 (1)	0.0 (0)
GED 312	15.4 (48)	16.0 (50)	14.1 (44)	12.8 (40)
Logistics 40	2.5 (1)	10.0 (4)	0.0 (0)	0.0 (0)
Organic synt. 56	30.4 (17)	30.4 (17)	0.0 (0)	0.0 (0)
Pipesworld 50	16.0 (8)	16.0 (8)	14.0 (7)	12.0 (6)
Rovers 40	2.5 (1)	5.0 (2)	0.0 (0)	0.0 (0)
Visitall MD 180	38.9 (70)	33.3 (60)	5.0 (9)	14.4 (26)
862	111.6 (151)	136.3 (157)	43.8 (65)	51.8 (77)

Table 2: Coverage results for length-optimal planning. The abbreviations have the same meaning as in Table 1.

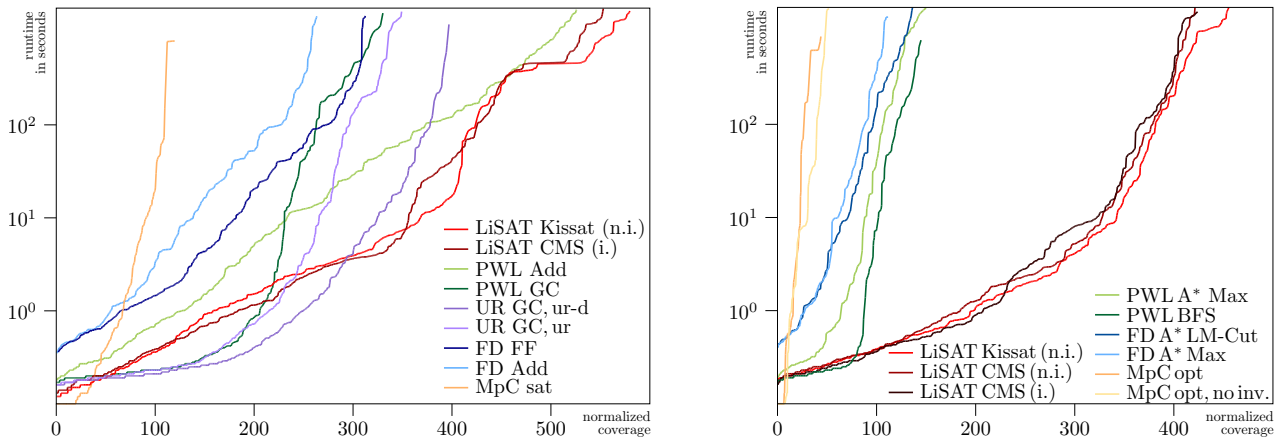

Figure 4: Normalized coverage against runtime (be aware the log scale) for the satisficing (left) and the optimal setting (right).

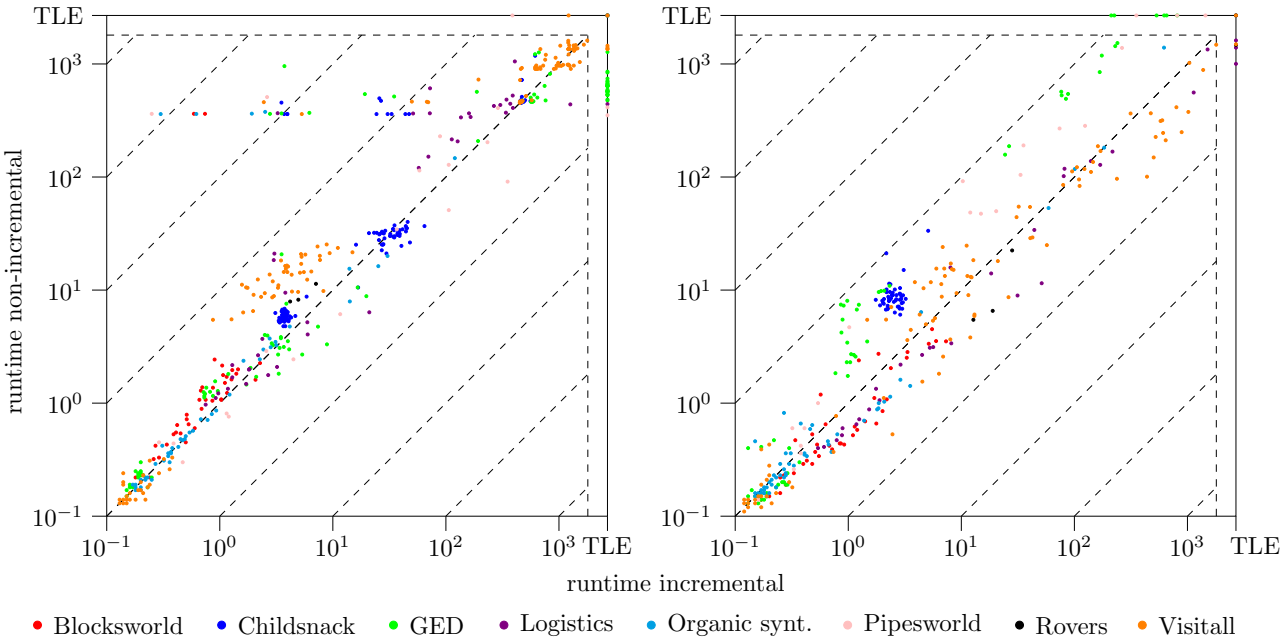

Figure 5: Ablation study on the runtime of CMS (be aware the log scale) for the satisficing (left) and the optimal setting (right). Each dot represents the runtime needed to solve an instance using incremental (x axis) and non-incremental (y axis) solving.

MiniSat (CMS) 5.8.0 (winner of the 2020 SAT Comp., incremental track) (Soos, Nohl, and Castelluccia 2009).

We evaluate our satisficing configuration (Alg. 2) against the PWL planner with (1) lazy best first search, helpful operators and the lifted h^{add} heuristic (Corrêa et al. 2021), and (2) eager best first search with Goal Counting (GC). Further we use GC with the unary relaxation heuristic as tiebreaker, with (ur-d) and without (ur) disambiguation of static predicates (Lauer et al. 2021). We also included Fast Downward (FD) (Helmert 2006) (which grounds the problems) with lazy best first search with h^{add} (Bonet and Geffner 2001) and h^{ff} (Hoffmann and Nebel 2001) and the SAT-based Madagascar system MpC (Rintanen, Heljanko, and Niemelä 2006; Rintanen 2014) in two configurations: once with and once without its pre-processing. We compare our length-optimal configuration (Alg. 1) with PWL using Breadth First Search (BFS) and an A^* search with the h^{max} heuristic used on the problems compiled to unit-costs. We included the FD system with A^* search with h^{max} (Bonet and Geffner 2001) and the LM-Cut heuristic (Helmert and Domshlak 2009), and the SAT-based Madagascar system MpC with the sequential encoding and a bound increase of 1.

The coverage results for the satisficing setting are given in Table 1. Since the domains come with a very different number of instances (between 40 and 312), we normalized the score in each domain to 100. The absolute number of solved instances is given afterwards in parenthesis.

From our configurations, the (non-incremental) Kissat solver reaches highest coverage, followed by the non-incremental and the incremental configurations using the

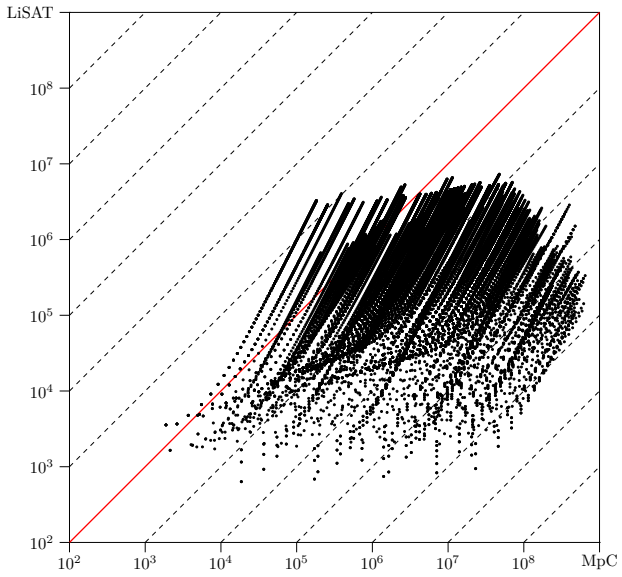

Figure 6: Comparison of formula sizes. Each dot represents the number of clauses generated by MpC (x axis) and LiSAT (y axis) for a certain instance and plan length bound.

CMS system. Compared to the competition, our system has the highest *normalized* coverage, while ur-d reaches the highest absolute coverage because it solves more instances in *Genome Edit Distance* (GED), which comes with 312 instances. In three domains, our system solves all instances, these are *Blocksworld*, *Childsnack*, and *Logistics*. In *Multi-dimensional Visitall* and *Organic Synthesis* we reach a coverage of over 97% and 92%. In *Pipesworld* and *GED*, we reach 42% and 35%. The worst performance of our system can be observed in *Rovers*, where we reach 10% coverage.

Compared with the lifted heuristic search-based systems, LiSAT performs better especially in two domains, *Blocksworld* and *Childsnack*, where its coverage is roughly 90.0 and 54.2 percent points higher. The PWL system has the highest advantage in *Rovers* (+67.5) and *GED* (+44.2).

The results for the length-optimal setting are given Table 2. Again, Kissat performed better than CMS on our encoding. When we compare the optimal to the satisficing configuration (Table 1), we see that the coverage dropped especially in two domains, *Childsnack* and *Visitall*. This is reasonable, since these are domains where the achiever/consumer distance should be low (resulting in a large gain of limiting their distance in the formula). Interestingly, there is also a domain with a (small) gain in coverage: in *Organic Synthesis*, LiSAT now has a coverage of 100%. In *Blocksworld*, the coverage is still 100%. In this setting, LiSAT outperforms all competitors reaching three times the normalized coverage of the other systems.

Figure 4 shows normalized coverage (on the x axis) against runtime (y axis). The left plot shows the satisficing setting. Between 1 and 10s, the GC heuristic and the heuristics by Lauer et al. reach highest (normalized) coverage, followed by LiSAT. Thereafter, LiSAT with the non-incremental solver is the fastest. The dent (around 400 points) is caused the following effect: in *Childsnack* and *Visitall*, the solver quickly shows unsolvability for small plan lengths, reaches the timeout for medium lengths, before solving the problems quickly when reaching a sufficient length. In the optimal setting (Figure 4, right) LiSAT outperforms the other systems. Since we do not use a time-slide strategy like before, we cannot observe a similar dent. For experimental results on classical IPC benchmarks, please see the appendix.

Next we compare an incremental and a non-incremental run of the CMS solver (Kissat does not support incremental solving). In the satisficing setting (Tab. 1), incremental solving does not increase the coverage in any domain. This is different in the optimal setting (Tab. 2). However, there is still no advantage in the total (normalized) coverage over all domains. Figure 5 gives an ablation study on incremental solving. Each dot represents the time needed by the two configurations to solve an instance. Over all, there are certain domains that benefit from incremental solving, but there is no systematic gain. For satisficing planning, the non-incremental version solves 36 problems not solved by the incremental, while only 2 are solved incrementally that are not solved without it. However, in 349 instances the incremental solver has a lower runtime, in 165 a higher one. The "horizontal" in the scatter plot are instances solved for length l by the incremental solver, but that time out non-incrementally and get solved quickly for the next length bound. For optimal planning, non-incremental solves 9 problems not solved by incremental and 5 vice versa. For 170 instances incremental solving is faster and for 163 non-incremental.

Figure 6 shows the number of clauses generated by MpC and by LiSAT in the optimal setting. For this experiment, we increased MpC's memory limit to 20GB. However, we only report a total of 488 instances from the domains *Blocksworld*, *Childsnack*, *GED*, *Pipesworld*, and *Visitall*. For the other problems, the formula generation of MpC ran out of memory. We give one point per instance and length bound from 1 to 100.

6 Conclusion

We presented a state-less encoding of lifted classical planning that is inspired by techniques from plan space planning. The resulting constraints can be realized in different ways, we presented an encoding in propositional logic to exploit the performance of modern SAT solvers. The encoding is quadratic in plan length. We presented a strategy to decrease the size by limiting the achiever/consumer distance, which proved to be very effective in practice. It further directs to a promising avenue for future work: to analyze the problem structure (e.g. using causal graph-like methods) and come up with different distances for each predicate. On a benchmark set dedicated to evaluate lifted planning systems, our approach reaches (slightly) higher normalized coverage than the predominating heuristic search-based systems. In a length-optimal setting, we reach three times the normalized coverage of these systems, including, e.g., a coverage of 100% in the *Organic Synthesis* domain.

Acknowledgments

We want to thank the reviewers for their help and effort to improve this work.

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 232722074 – SFB 1102 / Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 232722074 – SFB 1102.

References

Biere, A.; Fazekas, K.; Fleury, M.; and Heisinger, M. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In *Proceedings of the 2020 SAT Competition – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, 51–53. University of Helsinki.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33.

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Delete-Relaxation Heuristics for Lifted Classical Planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS)*, 94–102. AAAI Press.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation Using Query Optimization Techniques. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, 80–89. AAAI Press.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated planning – Theory and Practice*. Elsevier.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 162–169. AAAI Press.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)*, 14: 253–302.

Huang, R.; Chen, Y.; and Zhang, W. 2012. SAS+ Planning as Satisfiability. *Journal of Artificial Intelligence Research (JAIR)*, 43: 293–328.

Kautz, H.; and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI)*, 1194–1201. AAAI Press.

Kautz, H.; and Selman, B. 2006. SATPLAN04: Planning as Satisfiability. In *Proceedings of the 5th International Planning Competition (IPC)*, 45–46.

Kautz, H. A.; McAllester, D. A.; and Selman, B. 1996. Encoding Plans in Propositional Logic. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 374–384. Morgan Kaufmann.

Lauer, P.; Torralba, Á.; Fiser, D.; Höller, D.; Wichlacz, J.; and Hoffmann, J. 2021. Polynomial-Time in PDDL Input Size: Making the Delete Relaxation Feasible for Lifted Planning. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, 4119–4126. IJCAI organization.

Penberthy, J. S.; and Weld, D. S. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 103–114. Morgan Kaufmann.

Ridder, B.; and Fox, M. 2014. Heuristic Evaluation Based on Lifted Relaxed Planning Graphs. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*, 244–252. AAAI Press.

Rintanen, J. 2014. Madagascar: Scalable Planning with SAT. In *Proceedings of the 2014 International Planning Competition: Description of Participating Planners, Deterministic Track*, 66–70.

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

Robinson, N.; Gretton, C.; Pham, D. N.; and Sattar, A. 2009. SAT-Based Parallel Planning Using a Split Representation of Actions. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 281–288. AAAI Press.

Sinz, C. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP)*, 827–831. Springer.

Soos, M.; Nohl, K.; and Castelluccia, C. 2009. Extending SAT Solvers to Cryptographic Problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 244–257. Springer.

Wichlacz, J.; Höller, D.; and Hoffmann, J. 2021. Landmark Heuristics for Lifted Planning – Extended Abstract. In *Proceedings of the 14th International Symposium on Combinatorial Search (SoCS)*, 242–244. AAAI Press.

Younes, H. L. S.; and Simmons, R. G. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research (JAIR)*, 20: 405–430.