Symbolic Search for Optimal Total-Order HTN Planning

Gregor Behnke and David Speck
University of Freiburg
Freiburg im Breisgau, Germany
⟨behnkeg, speckd⟩@informatik.uni-freiburg.de

Abstract

Symbolic search has proven to be a useful approach to optimal classical planning. In Hierarchical Task Network (HTN) planning, however, there is little work on optimal planning. One reason for this is that in HTN planning, most algorithms are based on heuristic search, and admissible heuristics have to incorporate the structure of the task network in order to be informative. In this paper, we present a novel approach to optimal (totally-ordered) HTN planning, which is based on symbolic search. An empirical analysis shows that our symbolic approach outperforms the current state of the art for optimal totally-ordered HTN planning.

Introduction

Hierarchical Task Network (HTN) planning describes the problem to be solved in a twofold way: It provides a description of how the execution of actions changes the (propositional) state of the world and describes the ways and means by which a plan must be derived. This derivation takes the form of *decompositions* that are akin to derivation rules of formal grammars. While satisficing HTN planning has made significant progress, optimal HTN planning, i.e. the search for a guaranteed cheapest plan, is a still under-researched topic. This is witnessed by the small amount of planners that can actually produce optimal solutions to HTN planning problems (mainly Progression (Höller et al. 2020), Plan-Space-Search (Bercher et al. 2017), and SAT (Behnke, Höller, and Biundo 2019b)). We will not cover the full scope of HTN planning problems, but restrict ourselves to *Totally-Ordered* HTN (TOHTN) planning problems. This class has been the object of several independent investigations (e.g. (Nau et al. 1999; Schreiber et al. 2019)) and has an independent track at the 2020 International Planning Competition with twice as many competitors as the general track.

In this paper we propose a new way of solving HTN planning problems optimally – via symbolic search. In applying symbolic search, we draw from the success of symbolic search in classical planning (Cimatti et al. 1997; Edelkamp, Kissmann, and Torralba 2015). While explicit search techniques consider *individual* states as their search nodes, symbolic search techniques consider *sets* of states. Using suitable compact representations, symbolic search techniques can often outperform explicit search techniques. Symbolic representations have been used in HTN planning before. Kuter et al. (2005; 2009) used symbolic representations for solving non-deterministic planning problems. In this case, the purpose of a symbolic representation of sets of states was to represent all outcomes of *one* plan compactly. Combining states from multiple search nodes was not considered. This is in stark contrast to our work, which considers *multiple* states at the same time using symbolic representations.

Our work builds on research from the model-checking community (Esparza et al. 2000; Esparza and Schwoon 2001). We extend their work in three ways: (1) we show that their approach can be used to handle TOHTN planning problems, (2) we modify it to find cost-optimal plans, and (3) we show how such cost-optimal plans can be reconstructed.

We start our paper by first introducing the basic concepts of Totally-Ordered HTN planning and symbolic representations. We then present our approach in four steps: (1) we explain how it works for state-free domains without regard for costs, (2) we extend it to handle action costs, (3) we show how to handle the propositional state of the world, and (4) explain how plans can be reconstructed from a symbolic search structure. We conclude with an empirical evaluation.

Preliminaries

In this paper we consider TOHTN planning problems (Erol, Hendler, and Nau 1996). For an introduction into general HTN planning we refer to Erol, Hendler, and Nau (1996) and Geier and Bercher (2011). The difference between general and TOHTN planning is that in the latter methods contain *sequences* of tasks while for general HTNs they can contain arbitrary partially ordered sets. This strictly increases expressivity (Erol, Hendler, and Nau 1996; Höller et al. 2014) as the execution of parallel tasks can be interleaved.

A TOHTN planning problem $\Pi = \langle P, \mathfrak{A}, M, s_0, I\rangle$ describes a grammar-like structure from which plans must be derived. We distinguish two types of tasks: primitive actions P (terminals in a grammar) and abstract tasks $\mathfrak{A}$ (non-terminals in a grammar). The planning domain describes a set M of decomposition methods, which are rules of the form $A \mapsto t_1, \dots, t_n$, where $A \in \mathfrak{A}$ is an abstract task and $t_1, \dots, t_n$ is a possibly empty sequence of tasks (either primitive or abstract). With $M(t)$ we refer to all methods $t \mapsto t_1, \dots, t_n$. Applying a method to a sequence of tasks (a *task network*) $t_1^*, \dots, t_i^*, A, t_{i+1}^*, \dots, t_m^*$, yields the task network $t_1^*, \dots, t_i^*, t_1, \dots, t_n, t_{i+1}^*, \dots, t_m^*$. The TOHTN problem specifies an initial abstract task I. A primitive task network is a task network containing only actions, which is derivable from I by applying decomposition methods.

In addition to the grammar-like structure, the TOHTN problem also carries a state-transition semantics on its actions. Each action is associated with a set of preconditions $pre(a)$, add and delete effects $add(a)$ and $del(a)$, which are disjoint subsets of the set of state variables V. An action a is executable in a state $s \subseteq V$, iff $pre(a) \subseteq s$. The state resulting in executing a in s is $\gamma(s, a) = (s \setminus del(a)) \cup add(a)$. A sequence of actions $\pi = (a_1, \dots, a_n)$ is executable in s_0 if a_i is executable in s_{i-1}, where $s_i = \gamma(s_{i-1}, a_i)$. A primitive task network is a solution if it is executable in the initial state s_0. We are further given a cost function $c : P \to \mathbb{N}_0$ associating a cost with every action. The cost of a solution is the sum of the cost of all its actions. A solution is optimal when there is no other solution with strictly lower costs.

Example 1 (Running Example). *We will use the following TOHTN planning problem to exemplify the developed planning techniques. The problem has two abstract tasks: I, and A and three primitive actions: a, b, and c, all with cost* 1. *The initial abstract task is I. There are three decomposition methods:* $r_0 := I \mapsto A$, $r_1 := I \mapsto A, a$, *and* $r_2 := A \mapsto b, c$. *Consequently, there are two derivable primitive task networks: bca, bc. The set of state variables V is* $\{v\}$. *Further, b has the delete effect* $del(b) = \{v\}$ *and a has the precondition* $pre(a) = \{v\}$. *Thus executing b disables a. All other precondition, add, and delete sets are empty. The initial state* s_0 *is* $\{v\}$. *Thus, the primitive task network bc is a solution, while bca is not – as it is not executable.*

Regularisation. The right-hand side of a decomposition method can be any arbitrary sequence of actions and tasks. For the purposes of this paper, however, we only consider methods where the right-hand side consists of at most two actions and tasks. We call such a planning problem a 2-TOHTN problem. Given a general TOHTN planning problem, we can compile it into an equivalent 2-TOHTN problem by introducing new abstract tasks for methods containing more than two tasks on the right-hand side. For example, we can replace $A \mapsto a, b, c$ by $A \mapsto a, B$ and $B \mapsto b, c$. This compilation can be done in linear time and is performed by our planner prior to planning, but after grounding. Note that methods that decompose an abstract task into an empty sequence can be treated as single primitive actions with zero cost, which we assume for simplicity in this paper.

Progression Search. A common way to solve HTN planning problems is progression search (Nau et al. 1999). As its search nodes, it considers pairs $\langle s, \pi \rangle$ of states s and sequences of tasks and actions $\pi = (t_1, \dots, t_n)$. The intuitive meaning of such a search node is that we are currently in the state s and still have to perform the tasks and actions in π. We call this sequence of tasks the *stack*. If t_1 is primitive, the node has the successor $\langle \gamma(s, t_1), (t_2, \dots, t_n) \rangle$ iff t_1 is executable in s. If t_1 is abstract, the node has a successor for every method $t_1 \mapsto t_1', \dots, t_m'$. This successor is $\langle s, (t_1', \dots, t_m', t_2, \dots, t_n) \rangle$. Progression search starts with the node $\langle s_0, (I) \rangle$. A solution has been derived if we have reached the node $\langle s, \varepsilon \rangle$ for some state s (Alford et al. 2012).

Symbolic Representations

Symbolic search is a technique for exploring state spaces that uses efficient data structures to represent and manipulate sets of states (McMillan 1993). The underlying idea is to represent sets of states $S \subseteq 2^V$ via their characteristic function $\chi_S : 2^V \mapsto \{0, 1\}$, where $\chi_S(s) = 1$ if $s \in S$ and $\chi_S(s) = 0$ otherwise. In symbolic search, the most common data structure to represent such functions are Binary Decision Diagrams or BDDs (Bryant 1986). A primitive action $p \in P$ can be represented as a transition relation (TR) that is defined over sets of state pairs, namely predecessors and successors. A TR T_p representing a primitive action $p \in P$ is a function $\chi_{T_p} : 2^V \times 2^{V'} \mapsto \{0, 1\}$ which maps the pairs of states (s, s') to 1 iff successor s' is reachable from predecessor s by applying p. Given a set of states S and a TR T_p, the image operation computes all successors, referred to as the image set $\mathcal{I}$, of S with respect to p.

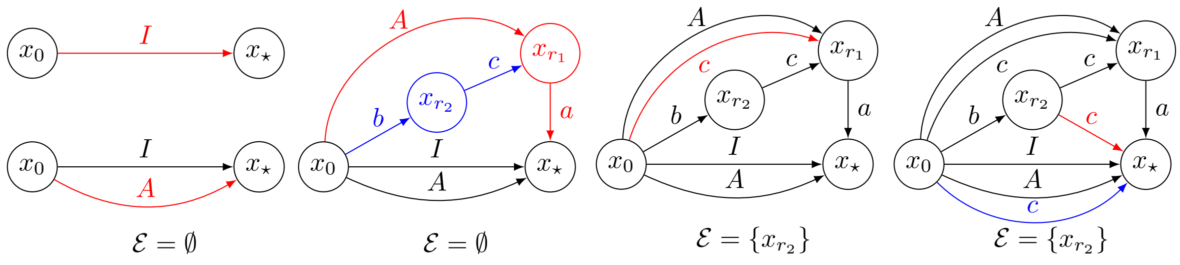
Symbolic Search for TOHTN Planning

We start by describing a restricted use-case of our symbolic HTN planning algorithm and subsequently extend it to cover full TOHTN problems. The underlying idea is to first create a foundation to understand the concepts and ideas of the algorithm, and then extend it to more sophisticated features.

State-free Reachability without Costs

In this section, we make two restrictive assumptions. First, we assume that there are no state variables, i.e. all actions are always applicable. Thus, every primitive task network is a solution to the planning problem. Second, we will not look for the cost-optimal solution, but only for *any satisficing* solution. Solving such a TOHTN planning problem boils down to finding any derivable primitive task network – or to formulate it in terms of a grammar G – any word in the language of G. These assumptions, in particular ignoring costs, make the model checking algorithm developed by Esparza et al. (2000) applicable to TOHTN planning problems.

Esparza et al. (2000) showed that the set of all reachable stacks $\mathfrak{S}$ forms a regular language and argued that any regular language can be represented by a finite automaton. Further, the automaton representing $\mathfrak{S}$ can be constructed incrementally using the problem's methods and actions. All intermediate steps in this contruction also only involve automata, i.e. regular sets of stacks. This makes searching for the empty stack possible. Intuitively, this follows the idea of the HTN progression search. The objective is to find a goal node, i.e. a search node with an empty stack $\langle s, \emptyset \rangle$. Since we assume for now that all actions are always applicable, the actual world state s of a search node $\langle s, \pi \rangle$ is ignored.



(a) Automata $\mathcal{A}_0$ and $\mathcal{A}_1$ (b) Automaton $\mathcal{A}_2$ (c) Automaton $\mathcal{A}_3$ (d) Automaton $\mathcal{A}_4$

Figure 1: Automata which are incrementally constructed by the algorithm of Esparza et al. (2000) for the running example (Examples 1) each explained in one of the Examples 2 to 5.

Definition 1. *An edge-labelled automaton $\mathcal{A}$ over the alphabet $\Sigma = P \cup A$ is a 4-tuple $(X, \delta, x_0, x_\star)$ where*
1. X is a set of nodes[1]*,*
2. $\delta \subseteq X \times \Sigma \times X$ is a transition relation,
3. $x_0 \in X$ is the initial node of the automaton, and
4. $x_\star \in X$ is the terminal node of the automaton
The set of stacks $\mathfrak{S}_\mathcal{A}$, i.e. the language, described by $\mathcal{A}$ is the set of accepting words which are induced by all paths from x_0 to $x_\star$. We thus use paths and stacks interchangeably.

The algorithm (Esparza et al. 2000) for determining whether $\mathfrak{S}$ contains the empty stack ε starts with the automaton $\mathcal{A}_0 = (\{x_0, x_\star\}, \{(x_0, I, x_\star)\}, x_0, x_\star)$ describing the set of stacks $\mathfrak{S}_{\mathcal{A}_0} = \{(I)\}$. The top automaton in Fig. 1a shows the initial automaton. The idea is to incrementally derive reachable stacks and add them to $\mathcal{A}_0$ until either no more stacks can be added (then the problem is unsolvable) or the empty stack is found (then the problem is solvable). Note that the following construction ensures that x_0 has only outgoing edges and $x_\star$ only incoming edges.

Let's consider an automaton $\mathcal{A} = (X, \delta, x_0, x_\star)$ representing stacks $\mathfrak{S}_\mathcal{A}$ and a queue τ of outgoing edges of x_0, i.e. $\tau = \langle (x_0, t, x), (x_0, t', x'), \dots \rangle$. We call these edges *heads*. Initially, τ consist of a single element $(x_0, I, x_\star)$. In every iteration of the algorithm, we pop the first element (x_0, t, x) of τ and process it. The head represents a subsets of stacks $\mathfrak{S}^h_\mathcal{A} \subseteq \mathfrak{S}_\mathcal{A}$ which start with the same task t. Not all stacks with the same starting task t must be necessarily represented by this head, as there can be multiple outgoing edges of x_0 with the same edge label but leading to different nodes. Overall, there are two cases, either the task t of the head (x_0, t, x) is a primitive action or t is an abstract task.

We consider a head (x_0, t, x) where $t \in \mathfrak{A}$ is an abstract task. For each decomposition method $t \mapsto t'$ which has a single task as its result, it is possible to replace the task t with the task t' as the first element of the set of stacks $\mathfrak{S}^h_\mathcal{A}$. Therefore, we add new edges $\{(x_0, t', x) \mid t \mapsto t' \in \mathfrak{A}\}$ to δ of automaton $\mathcal{A}$. All edges starting in x_0 which are new, i.e. which are not already included in δ, are added to τ.

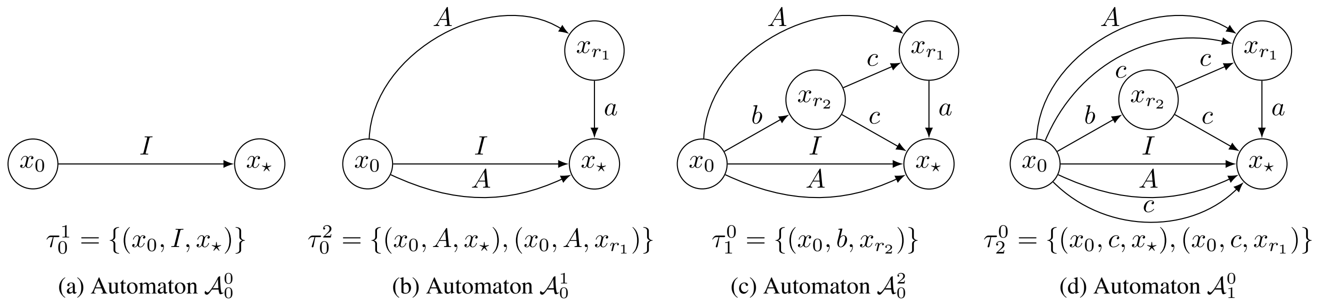
Example 2. *Consider Example 1, the initial automaton $\mathcal{A}_0$, which represent stack $\mathfrak{S}_{\mathcal{A}_0} = \{(I)\}$ and is shown in Fig. 1a (top), and the queue $\tau = \langle (x_0, I, x_\star) \rangle$. Fig. 1a (bottom) shows the automaton $\mathcal{A}_1$, which results from processing the head $(x_0, I, x_\star)$ according to the method $r_0 := I \mapsto A$. The automaton $\mathcal{A}_1$ represents the set of stacks $\mathfrak{S}_{\mathcal{A}_1} = \{(I), (A)\}$. We add the edge $(x_0, A, x_\star)$ to τ.*

Again, we consider a head (x_0, t, x) where $t \in \mathfrak{A}$ is an abstract task. For each decomposition method $r := t \mapsto t', t''$, we have to add a new node to the automaton $\mathcal{A}$. Following Esparza et al. (2000), we create one new node x_r and two new edges (x_0, t', x_r) and (x_r, t'', x) for each such method. With this procedure the set of stacks $\{(t', t'', t_1, \dots, t_n) \mid (t, t_1, \dots, t_n) \in \mathfrak{S}^h_\mathcal{A}\}$ is added to the set of stacks of the automaton. All new edges (x_0, t', x_r) are added to τ.

Example 3. *Consider Example 1 and the automaton $\mathcal{A}_1$ shown in Fig. 1a (bottom). We process the current head $(x_0, I, x_\star)$ according to the decomposition method $r_1 := I \mapsto A, a$, which introduces a new node x_{r_1} and two edges (x_0, A, x_{r1}) and $(x_{r1}, a, x_\star)$ shown in red in Fig. 1b. This intermediate automaton represents the set of stacks $\{(I), (A), (A, a)\}$. Additionally, we add the edge (x_0, A, x_{r1}) to τ, which already contains $(x_0, A, x_\star)$. For the purpose of the example, we pop (x_0, A, x_{r_1}) next. There is one method for A: $r_2 := A \mapsto b, c$. Processing the head results in the blue node and edges (Fig. 1b). The final automaton $\mathcal{A}_2$ represents the set of stacks $\mathfrak{S}_{\mathcal{A}_2} = \{(I), (A), (A, a), (b, c, a)\}$. The edge (x_0, b, x_{r_2}) is added to τ and we assume that this is the new first element of τ.*

Now we consider a head (x_0, p, x) where $p \in P$ is a primitive action. If the node x is the terminal node, i.e. $x = x_\star$, we have found the empty stack and thus a solution. Otherwise, we add new edges $\{(x_0, t', x') \mid (x, t', x') \in \delta\}$ to δ of the automaton $\mathcal{A}$. In other words, we add additional edges from x_0 to all nodes that are directly reachable from x. Intuitively, this describes the progression through the primitive action p and the automaton $\mathcal{A}$ then also represents the set of stacks $\{(t_1, \dots, t_n) \mid (p, t_1, \dots, t_n) \in \mathfrak{S}^h_\mathcal{A}\}$. Furthermore, we add each edge starting in x_0 which is new to τ. We also store an *epsilon set* $\mathcal{E}$, which is initially empty (Esparza et al. 2000). The epsilon set tracks the nodes $x \in X \setminus \{x_0, x_\star\}$ for which an incoming edge (x_0, p, x) with $p \in P$ has been processed. We explain this in more detail below.

[1]We call the states of an automaton nodes in this paper to avoid confusion with the state of the planning problem.



x_0
I
$x_\star$
$\tau_0^1 = \{(x_0, I, x_\star)\}$
(a) Automaton $\mathcal{A}_0^0$
A
x_{r_1}
a
x_0
I
A
$x_\star$
$\tau_0^2 = \{(x_0, A, x_\star), (x_0, A, x_{r_1})\}$
(b) Automaton $\mathcal{A}_0^1$
A
x_{r_1}
c
x_{r_2}
b
c
a
x_0
I
A
$x_\star$
$\tau_1^0 = \{(x_0, b, x_{r_2})\}$
(c) Automaton $\mathcal{A}_0^2$
A
x_{r_1}
c
c
x_{r_2}
b
c
a
x_0
I
A
c
$x_\star$
$\tau_2^0 = \{(x_0, c, x_\star), (x_0, c, x_{r_1})\}$
(d) Automaton $\mathcal{A}_1^0$

Figure 2: Automata of the cost layers constructed for the running example (Example 1) explained in Example 6. Note that we omit the epsilon sets as they are not relevant for the example.

Example 4. *Consider Example 1, the automaton $\mathcal{A}_2$ shown in Fig. 1b and the queue $\tau = \langle (x_0, b, x_{r_2}), (x_0, A, x_\star) \rangle$. The head (x_0, b, x_{r_2}) is popped from the queue and processed. The automaton $\mathcal{A}_3$ (Fig. 1c) is the result of processing (x_0, b, x_{r_2}), which represents the set of stacks $\mathfrak{S}_{\mathcal{A}_3} = \{(I), (A), (A, a), (b, c, a), (c, a)\}$. The new edge (x_0, c, x_{r_1}) is added to τ and x_{r_2} is added to the epsilon set, because the edge (x_0, b, x_{r_2}) was processed where $b \in P$.*

Lastly, we discuss the purpose of the epsilon set. Let us assume a head (x_0, t, x) is processed according to a decomposition method $r := t \mapsto t', t''$ and the node x_r and the edge (x_0, t', x_r) are already contained in the automaton. Further assume that $x_r \in \mathcal{E}$, implying that an edge (x_0, p, x_r) with $p \in P$ has already been processed. This edge stems from previously processing the edge (x_0, t', x_r) and possibly further edges which have been created and processed subsequently. If we just insert the edge (x_r, t'', x), we may never process any of the stacks that contain (x_r, t'', x) as all possible heads of these stacks have already been processed. The idea is to "reprocess" x_r at this point and add the edge (x_0, t'', x) to the automaton $\mathcal{A}$. This is necessary for completeness and is crucial for efficiency as it avoids redundant applications of operations.

Example 5. *Consider Example 1, the automaton $\mathcal{A}_3$ shown in Fig. 1c and the queue $\tau = \langle (x_0, A, x_\star), (x_0, c, x_{r_1}) \rangle$. The header $(x_0, A, x_\star)$ is popped from the queue τ and processed according to the decomposition rule $r_2 := A \mapsto b, c$, resulting in one new edge, which is shown in red in Fig. 1d. Note that x_{r_2} already exists and is part of the epsilon set. Since $x_{r_2} \in \mathcal{E}$, we add the edge $(x_0, c, x_\star)$ to the automaton shown in blue in Fig. 1d in order to consider the stack (b, c) which would be incomplete. The automaton $\mathcal{A}_4$ (Fig. 1d) is the result of processing $(x_0, A, x_\star)$, which represents the set of stacks $\mathfrak{S}_{\mathcal{A}_4} = \{(I), (A), (A, a), (b, c, a), (b, c), (c, a), (c)\}$. Finally, the new edge $(x_0, c, x_\star)$ is added to τ. After a possible progression of $(x_0, c, x_\star)$ the empty stack is derived.*

Finally, the algorithm presented is complete and correct, as it is a special case of the more general algorithm for stack-automata by Esparza et al. (2000).

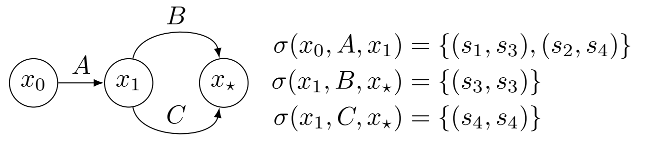
Adding Cost

Esparza et al. (2000) aimed at finding *some* solution. In optimal planning, we are interested in the *cheapest* plan. For this purpose, we consider the stacks in $\mathfrak{S}$ in ascending order of associated cost and thus construct a sequence of automata with increasing cost. To allow for a more efficient plan extraction, we will further introduce additional steps into the sequence of automata based on how may zero-cost operations were performed to get to them. In this, the generated layer or bucket structure is similar to the one used for symbolic classical planning (Edelkamp and Kissmann 2009; Torralba et al. 2017; Speck, Geißer, and Mattmüller 2018a).

We construct a sequence of automata $\mathcal{A}_c^i$ where c is a cost value and i is the layer-index inside of that cost value. $\mathfrak{S}_{\mathcal{A}_c^0}$ contains all stacks reachable with cost at most c. The cost of a stack π is the minimal cost through which we have to progress to reach π. $\mathfrak{S}_{\mathcal{A}_c^i}$ contains all stacks of $\mathfrak{S}_{\mathcal{A}_c^{i-1}}$ and those that can be created from any stack in $\mathfrak{S}_{\mathcal{A}_c^{i-1}}$ by one zero-cost progression or one method application. We set $\mathcal{A}_0^0 = (\{x_0, x_\star\}, \{(x_0, I, x_\star)\}, x_0, x_\star)$ (Fig. 2a).

Automata $\mathcal{A}_c^i$ are created in increasing order, first by layer and if a layer has been completed (no more stacks are reachable with additional cost of 0) by cost. The first time the empty stack is reached, a minimum cost plan is derived. As the basis for constructing an automaton $\mathcal{A}_c^i$, we take the previous automaton, i.e. $\mathcal{A}_c^{i-1}$ if $i \geq 1$ or else the last automaton of cost $c - 1$. For constructing the automata, we use one queue τ_c^i per automaton $\mathcal{A}_c^i$ and apply the operations described in the previous section to the heads in τ_c^i. Each queue contains all stack heads that have to be processed in order to construct the automaton $\mathcal{A}_c^i$. Separating the stack heads this way is possible as handling a head incurs a cost solely determined by the task t on the head (x_0, t, x') and not by the applied operation. If $t \in P$, progressing it will add cost $c(t)$, while applying any method does not add cost.

Let's consider constructing the automaton $\mathcal{A}_c^i$ and an edge (x_0, t, x') newly inserted into $\mathcal{A}_c^i$. The question is into which queue $\tau_{c'}^{i'}$ it is inserted. If t is a primitive $t \in P$ with non-zero cost $c(t) > 0$, we add it to $\tau_{c+c(t)}^0$ as progressing through this edge will incur additional cost. If either $c(t) = 0$ or $t \in \mathfrak{A}$, we add it to τ_c^{i+1}, as this does not incur costs, thus all resulting stacks still have cost c.



x_0
A
x_1
B
C
$x_\star$
$\sigma(x_0, A, x_1) = \{(s_1, s_3), (s_2, s_4)\}$
$\sigma(x_1, B, x_\star) = \{(s_3, s_3)\}$
$\sigma(x_1, C, x_\star) = \{(s_4, s_4)\}$

Figure 3: An automaton with edge labels.

Example 6. *Consider Example 1 and the initial automaton in Fig. 2a. We add its only edge to τ_0^1 as I is abstract. To construct $\mathcal{A}_0^1$ (Fig. 2b), we apply the two methods $I \mapsto A$ and $I \mapsto A, a$. The two new heads $(x_0, A, x_\star)$ and (x_0, A, x_{r_1}) are inserted into τ_0^2, as A is abstract. We then construct $\mathcal{A}_0^2$ (Fig. 2c) by applying the method $A \mapsto b, c$ to both heads. These applications both insert the edge (x_0, b, x_{r_2}) into τ_1^0, since $c(b) = 1$. $\mathcal{A}_1^0$ is then constructed by progressing through this edge (Fig. 2d). Thus, $(x_0, c, x_\star)$ and (x_0, c, x_{r_1}) are added to τ_2^0. When constructing the next automaton $\mathcal{A}_2^0$, we would progress through the first edge and thus reach the empty stack at cost 2. For the second edge, we would insert the edge $(x_0, a, x_\star)$ into the automaton and τ_3^0, yielding the second solution at cost 3.*

If we consider epsilon sets, we cannot use the same logic when we add an edge (x_0, t'', x') for a method $r := t \mapsto t', t''$ where $x_r \in \mathcal{E}$. This edge was inserted since the stack (t') induced by (x_0, t', x_r) can be processed into the empty stack ε. This separately performed processing is re-used when adding (x_0, t'', x'). While processing this stack into ε, we might have progressed through actions, whose costs we have to account for. We determine a cost value $\kappa(x_r)$ that provides for the cheapest way to process (x_0, t', x_r) into the empty stack if $x_r \in \mathcal{E}$. The edge (x_0, t'', x') is then inserted into a future automaton with $c^+ = c + \kappa(x_r)$, s.t. it is actually possible to reach the stacks starting with (x_0, t'', x_r) with the cost c^+. If $\kappa(x_r) = 0$, the edge is added to $\mathcal{A}_c^i$ and treated as an ordinary edge. Otherwise ($\kappa(x_r) > 0$), we add the edge to $\mathcal{A}^0_{c+\kappa(x_r)}$. Furthermore, in the latter case we consider the cost of t'': if $c(t'') > 0$, we add the edge to $\tau^0_{c+\kappa(x_r)+c(t'')}$ and else to $\tau^1_{c+\kappa(x_r)}$.

The value of $\kappa(x_r)$ can be determined when x_r is inserted into $\mathcal{E}$. At this time, we are constructing some automaton $\mathcal{A}_{c^*}^{i^*}$. Progressing towards x_r was started when the edge (x_0, t', x_r) was newly inserted into an automaton $\mathcal{A}_{c_0}^{i_0}$. The cost for processing (x_0, t', x_r) into the empty stack is $\kappa(x_r) = c^* - c_0$, since c^* is the cheapest cost where the edge (x_0, t', x_r) can be processed into the empty stack.

The addition of the edge (x_r, t'', x') can further lead to multiple new paths from x_0 via x_r to $x_\star$ because several already existing paths can lead from x_0 to x_r. However, the cost of such paths not starting with the edge (x_0, t', x_r) leading to x_r are not accounted for yet. In order to consider such costs, we add a cost label $\nu(x_r, t'', x')$ to the edge (x_r, t'', x'). When an edge (x_0, p, x_r) with $p \in P$ is processed for constructing $\mathcal{A}_z^j$, the resulting edge (x_0, t'', x') which is caused by the edge (x_r, t'', x') will incorporate the cost label. The edge is added to $\mathcal{A}^0_{z+\nu(x_r,t'',x')}$ if $\nu(x_r, t'', x') > 0$ and else to $\mathcal{A}_z^{j+1}$. Insertion into τ works similarly. In order to determine the additional cost associated with the edge (x_r, t'', x') we search for the automaton $\mathcal{A}_{c_0}^{i_0}$ in which the edge (x_0, t', x_r) was added. We assign the cost $\nu(x_r, t'', x') = c - c_0$ where c is the cost of the current automaton. Intuitively, this is the cost which is necessary to generate all paths leading from x_0 to x_r in automaton $\mathcal{A}_c^i$.

Thm. 1 proves that all stacks reachable with a given cost are actually contained in the respective automaton. Formal proofs can be found in a technical report (Behnke and Speck 2021).

Theorem 1. *The last automaton of every cost layer contains all stacks reachable with that cost.*

Thm. 1 does not guarantee that all costs are correctly computed, as stacks of cost c can occur in automata with lower cost. Thm. 2 shows that this is only possible for stacks that contain edges carrying additional costs ν. Since the empty stack can only be reached by progressing through an edge $(x_0, t, x_\star)$ – which cannot carry addition costs, the cost of this particular stack is correctly determined and hence also the cost of the empty stack and the the optimal plan's costs.

Theorem 2. *For every automaton $\mathcal{A}_c^i$, every stack in $\mathfrak{S}_{\mathcal{A}_c^i}$ that does not contain an edge (x, t, x') with $\nu(x, t, x') > 0$ can be reached with cost at most c.*

Adding State Variables

So far, we have only considered a relaxed version of the planning problem without state variables. The computational difficulty of HTN planning stems from the interaction between the state variables and the decomposition hierarchy. In this section, we describe how the previous construction can be extended to also handle state variables. Our work is based on Esparza and Schwoon's (2001) extension of the base algorithm without costs to cases with state variables. As in their work, we will use BDDs to handle state variables as they allow for compact representation and efficient manipulation of sets of states.

Up to now, an automaton $\mathcal{A}$ solely represented stacks. The idea of Esparza and Schwoon is to memorise a set of states associated which each stack in the automaton $\mathcal{A}$. More precisely, for each stack $\pi \in \mathfrak{S}_\mathcal{A}$ we maintain the set of states $S(\pi)$ that can be reached when we have the stack π still to be processed. The automaton will thus represent a set of progression search nodes given by $\{\langle s, \pi\rangle \mid \pi \in \mathfrak{S}_\mathcal{A}, s \in S(\pi)\}$. Intuitively, one might think that it suffices to memorise a set of states S for every outgoing edge (x_0, t, x') of x_0. This would imply that every stack starting with the edge (x_0, t, x') is combined with any state $s \in S$. This is problematic, as the representation by Esparza et al. (2000) forces a maximally compact representation of all stacks by only introducing a single state in the automaton per decomposition method. Thus this simple representation of states is not sufficient, since the required uniformity, i.e. every stack starting with an edge can be paired with every state in S, is not given. As an example consider the automaton depicted in Fig. 3 (left). It represents two stacks: (AB) and (AC). Now consider two progression states $\langle s_1, (AB)\rangle$ and $\langle s_2, (AC)\rangle$ for two propositional states s_1 and s_2. In order to represent them in the automaton, we would annotate the edge (x_0, A, x_1)

with $\{s_1, s_2\}$. This automaton also represents the progression states $\langle s_2, (AB)\rangle$ and $\langle s_1, (AC)\rangle$ – which is not intended. By introducing additional nodes to the automaton, it is still possible to represent only the two intended stacks, but this would lead to a less compact representation.

Esparza and Schwoon (2001) propose a different way of representing the possible current state associated with stacks. Each edge (x, t, x') of the automaton is annotated with a set of *pairs* (s_1, s_2) of states denoted $\sigma(x, t, x')$. For the outgoing edges of x_0 a state s_1 is one possible progression state. All other states are *tracking* states that are used to denote which progression search states are represented by an automaton and which are not.[2] When determining whether a specific progression search node is represented by the automaton, one does not simply traverse the states of automaton. Instead one traverses *pairs* (x, s) consisting of a state x of the automaton and a propositional state s. When traversing from (x, s) via the task t to (x', s') it is both required that the edge (x, t, x') is present in the automaton and that (s, s') is part of $\sigma(x, t, x')$. A formal definition of a path that is induced by such an automaton is given in Def. 2.

Definition 2. *Given an Σ-edge-labelled automaton $\mathcal{A} = (X, \delta, x_0, x_\star)$ and a function $\sigma : X \times \Sigma \times X \mapsto 2^{2^V \times 2^V}$.*
A path of $\mathcal{A}$ consisting of the edges $(x_0, t_0, x_1), (x_1, t_1, x_2), \ldots, (x_n, t_n, x_\star = x_{n+1})$ is induced by σ iff there is a sequence of states $s_0, s_1, \ldots, s_n$ s.t. $\forall i \in \{0, \ldots, n-1\} : (s_i, s_{i+1}) \in \sigma(x_i, t_i, x_{i+1})$.

If we again consider the example depicted in Fig. 3 together with σ, we can see that the two paths $\langle s_1, A, s_3, B, s_3\rangle$[3] and $\langle s_2, A, s_4, C, s_4\rangle$ are induced by σ. In contrast no path starting in s_1 and traversing (AC) is induced by σ, neither is any path starting in s_2 and traversing (AB). Thus this automaton represents exactly the intended progression search nodes: $\langle s_1, (AB)\rangle$ and $\langle s_2, (AC)\rangle$.

Conceptually, this construction is equivalent to considering a product automaton between the nodes of an original automaton and the states of a planning task. Edges in this automaton will take the form $((x, s_1), t, (x', s_2))$ and will be present if the edge (x, t, x') is present in the original automaton and $\sigma(x, t, x') = (s_1, s_2)$. Using σ sets introduced by Esparza and Schwoon (2001) makes it possible to store the associated sets of state pairs as BDDs.

Following Esparza and Schwoon (2001), the σ sets are computed as follows. Whenever an edge (x, t, x') is added to A^i_c, also a set of state pairs is added to the edge's BDD $\sigma(x, t, x')$. The same holds if the algorithm attempts to add an edge that is already present. τ only consists of edges and an edge is considered new if new pair of states are added to the edge's BDD. Consider processing a head edge (x_0, t, x').

Primitive actions. If we progress (x_0, p, x'), we apply p to $\sigma(x_0, p, x')$ using transition relation T_p resulting in the image set $\mathcal{I} = \{(\gamma(s_1, p), s_2) \mid pre(p) \subseteq s_1, (s_1, s_2) \in \sigma(x_0, p, x')\}$. For every outgoing edge (x', t, x'') we add $\{(s_1, s_3) \mid (s_1, s_2) \in \mathcal{I}, (s_2, s_3) \in \sigma(x', t, x'')\}$ to $\sigma(x_0, t, x'')$. The epsilon set stores sets of state pairs for every node x referred to with $\mathcal{E}(x)$. When progressing (x_0, p, x') the resulting image set $\mathcal{I}$ is added to $\mathcal{E}(x')$.

Methods with one subtask. Here the state does not change, but the head task in the stack is replaced with another. Therefore, $\sigma(x_0, t, x')$ is added to the new edge.

Methods with two subtasks. Consider method $r := t \mapsto t', t''$. Recall, that this adds two new edges (x_0, t', x_r) and (x_r, t'', x') to the automaton. We add $\{(s_1, s_1) \mid (s_1, s_2) \in \sigma(x_0, t, x')\}$ to $\sigma(x_0, t', x_r)$ and $\sigma(x_0, t, x')$ to $\sigma(x_r, t'', x')$. Further, $\{(s_1, s_3) \mid (s_1, s_2) \in \mathcal{E}(x_r), (s_2, s_3) \in \sigma(x_0, t, x)\}$ is added to $\sigma(x_0, t'', x')$.

Finally, action costs have to be taken into account – which was not considered by Esparza and Schwoon. The only element that needs modification is the computations of $\kappa(x_r)$ and $\nu(x_r, t'', x')$ – as they have to incorporate states. The costs $\kappa(x_r)$ must differentiate the state pairs with which we can reach the node x_r. It is not sufficient to store a single state, as the second state s' of the pair determines how the stack may be continued after x_r. Thus, instead of a function $\kappa(x_r)$ providing additional costs, we use one function $\kappa_z(x_r)$ per cost z, which returns state pairs represented as a BDD. If we progress (x_0, p, x_r) while constructing the automaton $\mathcal{A}^i_v$, we determine the state pairs in the image set $\mathcal{I}$ that are not contained in $\kappa_z(x_r)$ for all additional costs z. For each state pair $(s_1, s_2) \in \mathcal{I}$ we determine the automaton $\mathcal{A}^i_w$ in which (s_2, s_2) was first added to $\sigma(x_0, t, x_r)$. The state pair (s_1, s_2) is then added to $\kappa_{w-v}(x_r)$. We treat the function $\nu(x_r, t'', x')$ in the same way, i.e. we consider a family of function $\nu_z(x_r, t'', x')$ that provide for each edge (x_r, t'', x') and cost z those state pairs which are associated with an additional cost of z. The state pairs for $\nu_z(x_r, t'', x')$ are computed when new state pairs are added to that edge by applying a method with two subtasks. A formal proof of correctness and completeness can be found in a technical report (Behnke and Speck 2021).

Algorithm `autoSym`

The pseudo code of our BDD-based planning algorithm (`autoSym`) is given in Alg. 1. We assume that every function returns the empty set if not otherwise initialised. The algorithm proceeds by iterating over costs c and layers ℓ (line 6). Each iteration constructs the automaton $\mathcal{A}^\ell_c$ containing all stacks that can be reached by progression through actions with a total cost of at most c followed by ℓ layers of applications of methods and zero-cost actions. To compute $\mathcal{A}^\ell_c$ we first copy the previous automaton $\mathcal{A}^{ll}_{lc}$ – the **l**ast **c**ost and **l**ast **l**ayer (line 7). We then consider all head edges in τ^ℓ_c which must be processed to complete $\mathcal{A}^\ell_c$ (line 8) and call functions for handling the tree possible cases: actions and empty methods (line 10), methods with one subtask (line 13), and methods with two subtasks (line 15).

Alg. 2 adds new state pairs to the head edge (x_0, t, x') and adds the edge to the correct τ queue if necessary. Note that the edge is added to a future queue as the newly inserted head edge (x_0, t, x') may incur costs if processed and the state pairs added are those prior to performing t.

[2]These states are not chosen arbitrarily, but they are solely used to track possible stacks in the automaton. As a general idea, the tracking state leading to a state x_r in $\mathcal{A}$ is the state in which the method r was applied to the first task of the stack. For methods of the form $t \mapsto t'$ these states are actually inherited to other nodes.

[3]We add the intermediate states for easy readability.

Algorithm 1: Overall BDD-based Algorithm

```
1  Function autoSym($\Pi = \langle P, \mathfrak{A}, M, s_0, I\rangle$):
2      $\sigma_0^0(x_0, I, x_\star) = \{(s_0, s_0)\}$; $\tau_0^1 = \{(x_0, I, x_\star)\}$
3      $c = 0$; $\ell = 1$                      // current automaton
4      $lc = 0$; $ll = 0$          // last/previous automaton
5      $layers_0 = 1$                  // #layers for cost 0
6      while $\exists c' \geq c \wedge \ell' \geq \ell : \tau_{c'}^{\ell'} \neq \emptyset$ do
7          $\sigma_c^\ell = \sigma_{lc}^{ll}$
8          foreach $(x_0, t, x') \in \tau_c^\ell$ do
9              if $t \in P$ and $(t \mapsto \varepsilon) \in M(t)$ then
10                 $done$ = handlePrimEmpty($c, \ell, lc, ll, t, x'$)
11                 if $done$ then extractPlan
12             foreach $(t \mapsto t') \in M(t)$ do
13                 addToAutomaton($c, \ell, t', x', \sigma_{lc}^{ll}(x_0, t, x')$)
14             foreach $(r = t \mapsto t't'') \in M(t)$ do
15                 handle2Tasks($c, \ell, lc, ll, x', r$)
16         $lc = c$; $ll = \ell$
17         // Next cost/layer to handle
18         if $\tau_c^{\ell+1} = \emptyset$ then
19             $c = c + 1$ and $\ell = 0$ and $layers_c = 0$
20         else $\ell = \ell + 1$ and $layers_c = \ell$
21     return unsolvable
```

Algorithm 2: Adding state pairs

```
1  // add set of state pairs statePairs to edge
       $(x_0, t, x')$ at cost $c$ and layer $\ell$
2  Function addToAutomaton($c, \ell, t, x', statePairs$):
3      $newPairs = \sigma_c^\ell(x_0, t, x') \setminus statePairs$
4      if $newPairs \neq \emptyset$ then
5          $\sigma_c^\ell(x_0, t, x') = \sigma_c^\ell(x_0, t, x') \cup newPairs$
6          if $t \in \mathfrak{A}$ or ($t \in P$ and $c(t) = 0$) then
7              $\tau_c^{\ell+1} = \tau_c^{\ell+1} \cup \{(x_0, t, x')\}$
8          else $\tau_{c+c(t)}^0 = \tau_{c+c(t)}^0 \cup \{(x_0, t, x')\}$
```

Alg. 3 handles both primitive actions and tasks with an empty method. We start by applying the actions transition relation (or do nothing for abstract tasks; lines 2-4). If we progressed through an edge leading to the final state $x_\star$ of the automaton, we signal to start plan extraction (called in Alg 1, line 11) by returning true. Plan extraction is discussed in the next section. Thereafter, we compute the values for κ (lines 7-12) and add new edges to $\mathcal{A}_c^\ell$ via calling the addToQueue function, while taking the additional costs of edges in ν into account (lines 13-21).

Alg. 4 handles methods with two subtasks and compute the table ν used by handlePrimEmpty. Initially, we insert the two main edges into the automaton (lines 3-4). For state pairs that are already in the automaton (line 6), we take extra costs κ into account (lines 8-12) and compute additional costs ν associated with (x_t, t'', x') (lines 13-16).

Plan Extraction

Esparza and Schwoon (2001) were only interested in determining whether – formulated for planning – the instance is solvable. In planning, we are also interested in the resulting plan. We follow the typical approach used by classical symbolic planners (Torralba 2015; Speck, Mattmüller, and Nebel 2020): backwards search from the goal while using the constructed data structures as a heuristic. In classical planning, we have constructed the perfect heuristic h^* making this search efficient. Our experiments show that in practice the time it takes to extract the plan is low compared to the time needed to construct the automata.

Algorithm 3: Primitive actions and empty methods

```
1  Function handlePrimEmpty($c, \ell, lc, ll, t, x'$):
2      if $t \in P$ then
3          $\mathcal{I} = \{(\gamma(s_1, t), s_2) \mid pre(t) \subseteq s_1, (s_1, s_2) \in \sigma_{lc}^{ll}(x_0, t, x')\}$
4      else $\mathcal{I} = \sigma_{lc}^{ll}(x_0, t, x')$          // $t$ has empty method
5      if $x' = x_\star$ and $\mathcal{I} \neq \emptyset$ then return true
6      $new\mathcal{E} = \mathcal{I} \setminus \mathcal{E}(x')$; $\mathcal{E}(x') = \mathcal{E}(x') \cup \mathcal{I}$
7      if $x' \neq x_\star$ then
8          Let $r = t' \mapsto t''t''' \in M(t')$ s.t. $x_r \equiv x'$
9          foreach $c_0 \in \{0, 1, \ldots, c\}$ do
10             $started = \{(s_1, s_2) \in new\mathcal{E} \mid$
                   $(s_2, s_2) \in \sigma_{c_0}^{layers_{c_0}}(x_0, t'', x')\}$
11             $new\mathcal{E} = new\mathcal{E} \setminus started$
12             $\kappa_{c-c_0}(x') = \kappa_{c-c_0}(x') \cup started$
13         foreach $x''$ with $\sigma_{lc}^{ll}(x_r, t''', x'') \neq \emptyset$ do
14             $new = \{(s_1, s_2, s_3) \mid (s_1, s_2) \in \mathcal{I},$
                   $(s_2, s_3) \in \sigma_{lc}^{ll}(x_r, t''', x'')\}$
15             foreach $c^+ \in \{i \mid \nu_i(x_r, t''', x'') \neq \emptyset\}$ in ascending
                   order do
16                 $add = \{(s_1, s_3) \mid (s_1, s_2, s_3) \in new,$
                       $(s_2, s_3) \in \nu_{c^+}(x_r, t''', x'')\}$
17                 $new = new \setminus \{(s_1, s_2, s_3) \in new \mid (s_1, s_3) \in add\}$
18                 if $c^+ = 0$ then
19                     addToAutomaton($c, \ell, t''', x'', add$)
20                 else addToAutomaton($c + c^+, 0, t''', x'', add$)
21             addToAutomaton($c, \ell, t''', x''$,
                   $\{(s_1, s_3) \mid (s_1, s_2, s_3) \in new\}$)
22     return false
```

The plan extraction is started when the empty stack is reached, i.e. when progressing through an edge $(x_0, p, x_\star)$, where $p \in P$ (Alg. 3, line 5). We then perform a backwards search over the automata $\mathcal{A}_c^i$ starting from a terminal state s_t. The terminal state s_t to start reconstruction with, can be chosen freely from $\{s_1 \mid (s_1, s_2) \in \mathcal{I}\}$ where $\mathcal{I}$ is the image set we have computed for the edge $(x_0, p, x_\star)$. During search, we maintain a current progression search state $\langle s, \pi\rangle$ from which the empty stack is reachable and which is contained in the current automaton $\mathcal{A}_c^i$. We not only memorise the stack (i.e. the tasks) π, but also the intermediate automata states. We thus start extraction with the pair $\langle s_g, x_\star\rangle$ at the last constructed automaton. As an invariant, we maintain that $\langle s, \pi\rangle$ was newly inserted at layer i of cost c.

If we are given a current state s and a current stack (i.e. path) π with the head (x_0, t, x') (or the empty stack in the beginning), we have to construct a predecessor pair $\langle s^*, \pi^*\rangle$ of $\langle s, \pi\rangle$ that was newly added to a previous automaton. For this purpose, we keep track of which heads (x_0^*, t^*, x'^*) in the queue τ_c^i caused the insertion of (x_0, t, x') into $\mathcal{A}_c^i$. Further we memorise why the head (x_0^*, t^*, x'^*) was inserted

Algorithm 4: Handling methods with two subtasks

```
1  Function handle2Tasks($c, \ell, lc, ll, x', r = t \mapsto t't''$):
2      $add = \{(s_1, s_1) \mid (s_1, s_2) \in \sigma^{ll}_{lc}(x_0, t, x')\}$
3      addToAutomaton($c, \ell, t', x_r, add$)
4      $\sigma^{\ell}_{c}(x_r, t'', x') = \sigma^{\ell}_{c}(x_r, t'', x') \cup \sigma^{ll}_{lc}(x_0, t, x')$
5      // Extra costs (nodes/edges)
6      $known = add \cap \sigma^{ll}_{lc}(x_0, t', x_r)$
7      if $known \neq \emptyset$ then
8          foreach $c^+ \in \{i \mid \kappa_i(x_r) \neq \emptyset\}$ in ascending order do
9              $processed = \{(s_3, s_1) \in \kappa_{c^+}(x_r) \mid (s_1, s_1) \in known\}$
10             if $c^+ = 0$ then
11                 addToAutomaton($c, \ell, t'', x', processed$)
12             else addToAutomaton($c + c^+, 0, t'', x', processed$)
13         foreach $c_0 \in \{0, 1, \ldots, c\}$ do
14             $started = known \cap \sigma^{layers_{c_0}}_{c_0}(x_0, t', x_r)$
15             $known = known \setminus started$
16             $\nu_{c-c_0}(x_r, t'', x') = \nu_{c-c_0}(x_r, t'', x') \cup$
                   $\{(s_1, s_2) \in \sigma^{ll}_{lc}(x_0, t, x') \mid (s_1, s_1) \in started\}$
```

into τ^i_c. These reasons are pairs of cost c^- and layer i^- s.t. processing some head in $\tau^{i^-}_{c^-}$ has lead to (x^*_0, t^*, x'^*) being inserted into τ^i_c. To find the predecessor pair, we iterate over the causes (x^*_0, t^*, x'^*) for the head (x_0, t, x'). Given the cause, the performed operation is reversely applied to $\langle s, \pi \rangle$ – regressing t (if $t \in P$) or un-applying a method for t (if $t \in \mathfrak{A}$) – yielding $\langle s^*, \pi^* \rangle$. If they are present in $\mathcal{A}^{i^-}_{c^-}$ we continue with reconstructing that pair, if not, we try the next cause. We repeat this process until the extraction finishes by reaching $\langle s_0, (x_0, I, x_\star) \rangle$ in $\mathcal{A}^0_0$. To find the actual sequence of actions solving the planing problems, we iterate over the backwards steps we have performed during search and add an action to the plan whenever we regressed through it.

Example 7. *Consider the automaton in Fig. 2d. We reach the empty stack while constructing $\mathcal{A}^0_2$ when progressing through $(x_0, c, x_\star)$. We thus start extraction at $\mathcal{A}^0_2$ with $\langle \emptyset, (x_\star) \rangle$. The cause for reaching the empty stack is that the edge $(x_0, c, x_\star)$ was inserted into $\mathcal{A}^0_1$, i.e. $c^- = 1$ and $i^- = 0$. We thus apply the progression backwards and obtain $\langle \emptyset, (x_0, c, x_\star) \rangle$. The cause for inserting the head edge $(x_0, c, x_\star)$ was progressing through the edge (x_0, b, x_{r_2}) in $\mathcal{A}^2_0$, i.e. $c^- = 0$ and $i^- = 2$. We again apply progression backwards and obtain $\langle \{v\}, (x_0, b, x_{r_2}, c, x_\star)$, which was new for $\mathcal{A}^2_0$. Next, we consider the cause for inserting the head edge (x_0, b, x_{r_2}) – which was applying the methods r_2 to the task A on the edge $(x_0, A, x_\star)$. Here $c^- = 0$ and $i^- = 1$. At this point, we have to apply the method $r_2 = A \mapsto bc$ backwards by removing b and c from the stack and adding A. This results in $\langle \{v\}, (x_0, A, x_\star) \rangle$, which was new for $\mathcal{A}^1_0$. The cause for inserting $(x_0, A, x_\star)$ was processing the edge $(x_0, I, x_\star)$ of $\mathcal{A}^0_0$. We apply the method r_0 backwards and obtain $\langle \{v\}, (x_0, I, s_\star) \rangle$. Here we can extract the solution plan bc, since we reached the initial state $s_0 = \{v\}$ and the initial stack (I).*

This extraction idea only works if the cause of adding an edge was *regularly* processing an other edge. If the edge was added due to additional costs (i.e. using the κ and ν tables), we have to extract a plan corresponding to them, too. Consider that we have added an edge due to a node $x_r \in \mathcal{E}$, which happens if we apply a method $r := t \mapsto t', t''$ to an edge (x_0, t, x'). We cannot "un-apply" the method r as the task t' is not currently on the stack. We "pause" the extraction of the main plan. We then extract a plan that decomposes the stack (x_0, t', x_r) into the empty stack (if the method is $r := t \mapsto t', t''$). As a result, we add (x_0, t', x_r) to the stack, apply r backwards, and continue as normal. Finding a plan decomposing (x_0, t', x_r) works as the main extraction – while treating x_r as $x_\star$ and t' as I and must have cost $\kappa(x_r)$. Such sub-extractions can be nested inside of each other. If we regress through an edge with additional costs ν, we have to split the extraction into a sub-extraction that extracts the correct plan to reach that edge and then continue the main extraction that caused the edge to be inserted.

Empirical Evaluation

We implemented the presented approach `autoSym` and empirically compared it with other optimal (TO)HTN planners.[4] `autoSym` is based on the PANDA planning framework in its C++ version pandaPI (Höller et al. 2021). Since `autoSym` operates on a grounded model, we use parser and grounder of pandaPI (Behnke et al. 2020).[5] `autoSym` uses the default pandaPI variable ordering for the BDDs. Modern classical planners (Torralba et al. 2014; Kissmann, Edelkamp, and Hoffmann 2014; Speck, Geißer, and Mattmüller 2018b) use sophisticated methods to predict good variable ordering, which can play a crucial role in symbolic planning based on decision diagrams (Kissmann and Hoffmann 2014). We leave this to future work. To represent and manipulate BDDs we use CUDD 3.0.0 (Somenzi 2015).

Algorithm	**Sym. Search**	SAT		A*	
Domain (# Inst.)	**`autoSym`**	bin	dec	LM-cut	TDG-c
Barman (20)	0	**12**	7	0	0
Blocks-GTOHP (20)	12	1	1	**18**	4
Blocks-HPDDL (20)	**3**	**3**	**3**	1	0
Childsnack (20)	2	**6**	**6**	0	0
Depots (20)	**20**	10	10	15	3
Entertainment (12)	**12**	10	**12**	5	5
Gripper (20)	**20**	**20**	**20**	17	4
Hiking (20)	**7**	0	0	0	0
Minecraft-Area (30)	**15**	0	0	0	0
Minecraft-Normal (30)	**19**	1	0	0	0
Multiarm-Blocks (74)	**10**	0	0	4	0
Robot (10)	**1**	**1**	**1**	0	0
Rover-GTOHP (20)	**7**	4	4	5	4
Rover-PANDA (20)	13	**18**	15	10	4
Satellite-GTOHP (20)	3	3	3	**5**	3
Satellite-PANDA (25)	**25**	**25**	**25**	**25**	22
SmartPhone (7)	6	**7**	**7**	5	4
Towers (14)	**10**	6	6	9	2
Transport (30)	**24**	14	12	5	2
UM-Translog (22)	**22**	**22**	**22**	**22**	**22**
Woodworking (11)	**11**	**11**	**11**	**11**	8
Overall (465)	**242**	174	165	157	87

Table 1: Coverage Table.

[4]https://github.com/galvusdamor/pandaPIengineSymbolic

[5]https://github.com/panda-planner-dev/pandaPIparser and https://github.com/panda-planner-dev/pandaPIgrounder

Domains. In contrast to the previous work by Behnke, Höller, and Biundo (2019b), we have used a far more extensive domain set. We have gathered as many domains as possible from former evaluations of (satisficing) TOHTN planners. This amounts to four different sources: (1) (Behnke, Höller, and Biundo 2018) with 7 domains, (2) (Alford et al. 2016) with 4 domains, (3) (Schreiber et al. 2019) with 8 domains, and (4) (Wichlacz, Torralba, and Hoffmann 2019) with 2 domains – with a total of 465 instances.

Planners. As stated in the introduction, *optimal* HTN planning is a relatively new and so-far under-researched topic. Thus, only few planners exist for comparison. We compare autoSym against four other planners: (1&2) the SAT-based planners SAT(bin) and SAT(dec) (Behnke, Höller, and Biundo 2019b), (3) Progression search with the LM-cut heuristics (Höller et al. 2020; Helmert and Domshlak 2009), and (4) PANDA using the TDG-c heuristics (Bercher et al. 2017). HTN2STRIPS (Alford et al. 2016) is excluded as it was significantly less efficient in previous evaluations. Tree-REX (Schreiber et al. 2019) does not produce optimal solutions, but only locally optimises satisficing plans. Every planner was given 4 GB of RAM and 30 minutes of runtime on a compute cluster with nodes equipped with two Intel Xeon Gold 6242 32-core CPUs, 20 MB cache and 188 GB shared RAM running Ubuntu 18.04 LTS 64 bit.

Results. Tab. 1 shows the overall coverage of all five planners on the benchmark instances. autoSym outperforms its next best competitor SAT(bin) by 68 instances. In 15 out of 21 domains autoSym solves more or as much as the other planners. In four other domains (Blocks-GTOHP, Rover-PANDA, SmartPhone, Satellite-GTOHP) it is comparable in performance with the other planners. In the two remaining domains (Barman, Childsnack) its performance is significantly worse than that of the SAT-based planner. This is due to the high number of methods in these instances, which lead to a quite large automaton compared to the other instances. In these two domains, autoSym cannot profit from the compacted representation using BDDs. Improving this is interesting for future research.

Figure 4 shows the coverage over time. The progression-based planner (A* LM-cut) has an early lead, but autoSym overtakes it after less than a second and dominates it afterwards. Figure 5 shows a per-instance runtime comparison between the two best planners overall, autoSym and SAT(bin). In most instances, which are solved by both planners, autoSym is at least one order of magnitude faster. However, there are instances where autoSym is significantly slower or which cannot be solved by autoSym within the given memory and time limits. Overall, autoSym is a valuable addition to the state of the art, as it is a technique that is orthogonal to the existing ones.

Finally, we would like to mention the time it takes for the presented symbolic approach autoSym to reconstruct a found solution. Plan extraction can take up to 64.88% of the planner's time, but this is usually only the case with instances that are solved quickly. For instances solved in more than one second, this percentage decreases to 22.66%, and for one minute, it decreases further to 6.94%. The slowest extraction took 38.66 sec. (Minecraft-Normal).

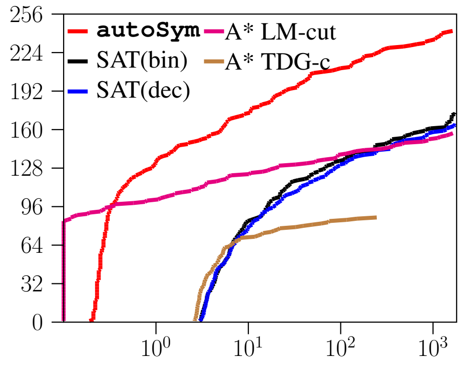


Figure 4: Inverted Cactus Plot showing the runtime necessary to solve a given amount of instances. Runtime in seconds is on the x-axis (log scale). Number of solved instances is on the y-axis. Planners are coded by colour.

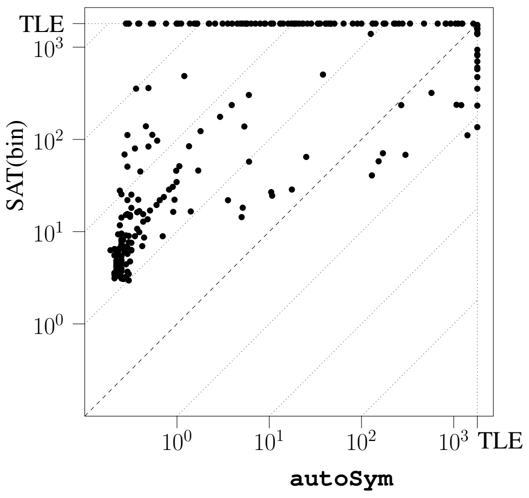


Figure 5: Runtime scatter plot (log scale) of autoSym and SAT(bin) in sec. per instance. TLE=Time Limit Exceeded.

Conclusion

We have presented a new means to solve TOHTN planning problems: symbolic search. We have build upon techniques from the model-checking community. This technique has the advantage of being able to produce guaranteed optimal solutions. The resulting planner was shown to significantly outperform other state-of-the-art optimal TOHTN planners. In the future, we may address the weaknesses of the algorithm shown by the evaluation on some domains. This could be possible through a more sophisticated method of variable ordering or through the use of heuristics. While it has been shown that the use of heuristics in the form of distance estimates to the goal often does not pay off in symbolic search (Speck, Geißer, and Mattmüller 2020), it might be possible here to use heuristics to prioritise the edges of the constructed automaton to be processed. Another interesting direction is to extend the algorithm to partially-ordered HTN planning (Behnke, Höller, and Biundo 2019a).

Acknowledgments

David Speck was supported by the German Research Foundation (DFG) as part of the project EPSDAC (MA 7790/1-1). We thank the anonymous reviewers for their comments and suggestions.

References

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016. Bound to Plan: Exploiting Classical Heuristics via Automatic Translations of Tail-Recursive HTN Problems. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 20–28. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 2–9. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-Ordered Hierarchical Planning Through SAT. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 6110–6118. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing Order to Chaos - A Compact Representation of Partial Order in SAT-Based HTN Planning. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7520–7529. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding Optimal Solutions in HTN Planning – A SAT-based Approach. In Kraus, S., ed., *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 5500–5508. IJCAI.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On Succinct Groundings of HTN Planning Problems. In Conitzer, V.; and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*, 9775–9784. AAAI Press.

Behnke, G.; and Speck, D. 2021. Symbolic Search for Total-Order HTN Planning: Technical Report. Technical Report 295, University of Freiburg, Department of Computer Science.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 480–488. IJCAI.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* 35(8): 677–691.

Cimatti, A.; Giunchiglia, F.; Giunchiglia, E.; and Traverso, P. 1997. Planning via Model Checking: A Decision Procedure for *AR*. In Steel, S.; and Alami, R., eds., *Recent Advances in AI Planning. 4th European Conference on Planning (ECP 1997)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, 130–142. Springer-Verlag.

Edelkamp, S.; and Kissmann, P. 2009. Optimal Symbolic Planning with Action Costs and Preferences. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1690–1695. AAAI Press.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2015. BDDs Strike Back (in AI Planning). In Bonet, B.; and Koenig, S., eds., *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 4320–4321. AAAI Press.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence (AMAI)* 18(1): 69–93.

Esparza, J.; Hansel, D.; Rossmanith, P.; and Schwoon, S. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In Emerson, E. A.; and Sistla, A. P., eds., *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, 232–247. Springer.

Esparza, J.; and Schwoon, S. 2001. A BDD-Based Model Checker for Recursive Programs. In Berry, G.; Comon, H.; and Finkel, A., eds., *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, 324–336. Springer.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 1955–1961. AAAI Press.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2021. The PANDA Framework for Hierarchical Planning. *KI – Künstliche Intelligenz* .

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN Planning as Heuristic Progression Search. *Journal of Artificial Intelligence Research* 67: 835–880.

Kissmann, P.; Edelkamp, S.; and Hoffmann, J. 2014. Gamer and Dynamic-Gamer – Symbolic Search at IPC 2014. In *Eighth International Planning Competition (IPC-8): planner abstracts*, 77–84.

Kissmann, P.; and Hoffmann, J. 2014. BDD Ordering Heuristics for Classical Planning. *Journal of Artificial Intelligence Research* 51: 779–804.

Kuter, U.; Nau, D. S.; Pistore, M.; and Traverso, P. 2005. A Hierarchical Task-Network Planner based on Symbolic Model Checking. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference* *on Automated Planning and Scheduling (ICAPS 2005)*, 300–309. AAAI Press.

Kuter, U.; Nau, D. S.; Pistore, M.; and Traverso, P. 2009. Task decomposition on abstract states, for planning under nondeterminism. *Artificial Intelligence* 173(5): 669–695.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In Dean, T., ed., *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 1999)*, 968–975. Morgan Kaufmann.

Schreiber, D.; Pellier, D.; Fiorino, H.; and Balyo, T. 2019. Tree-REX: SAT-Based Tree Exploration for Efficient and High-Quality HTN Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 382–390. AAAI Press.

Somenzi, F. 2015. CUDD: CU decision diagram package - release 3.0.0. https://github.com/ivmai/cudd. Unofficial git mirror of http://vlsi.colorado.edu/~fabio/, Accessed: 2020-02-20, commit f54f533.

Speck, D.; Geißer, F.; and Mattmüller, R. 2018a. Symbolic Planning with Edge-Valued Multi-Valued Decision Diagrams. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 250–258. AAAI Press.

Speck, D.; Geißer, F.; and Mattmüller, R. 2018b. SYMPLE: Symbolic Planning based on EVMDDs. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 91–94.

Speck, D.; Geißer, F.; and Mattmüller, R. 2020. When Perfect Is Not Good Enough: On the Search Behaviour of Symbolic Heuristic Search. In Beck, J. C.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 263–271. AAAI Press.

Speck, D.; Mattmüller, R.; and Nebel, B. 2020. Symbolic Top-k Planning. In Conitzer, V.; and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*, 9967–9974. AAAI Press.

Torralba, Á. 2015. *Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning*. Ph.D. thesis, Universidad Carlos III de Madrid.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A Symbolic Bidirectional A* Planner. In *Eighth International Planning Competition (IPC-8): planner abstracts*, 105–109.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient Symbolic Search for Cost-optimal Planning. *Artificial Intelligence* 242: 52–79.

Wichlacz, J.; Torralba, Á.; and Hoffmann, J. 2019. Construction-Planning Models in Minecraft. In *ICAPS 2019 Workshop on Hierarchical Planning*, 1–5.