Applying Monte-Carlo Tree Search in HTN Planning

Julia Wichlacz, Daniel Höller, Álvaro Torralba, Jörg Hoffmann

Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

wichlacz@cs.uni-saarland.de, hoeller@cs.uni-saarland.de, torralba@cs.uni-saarland.de, hoffmann@cs.uni-saarland.de

Abstract

Search methods are useful in hierarchical task network (HTN) planning to make performance less dependent on the domain knowledge provided, and to minimize plan costs. Here we investigate Monte-Carlo tree search (MCTS) as a new algorithmic alternative in HTN planning. We implement combinations of MCTS with heuristic search in Panda. We furthermore investigate MCTS in JSHOP, to address lifted (non-grounded) planning, leveraging the fact that, in contrast to other search methods, MCTS does not require a grounded task representation. Our new methods yield coverage performance on par with the state of the art, but in addition can effectively minimize plan cost over time.

1 Introduction

Hierarchical Task Network (HTN) planning complements actions causing state transition like in classical planning with a hierarchy on the things to do, the *tasks*. *Abstract* tasks are not applicable directly and are decomposed in a process similar to the derivation of words from a formal grammar. Solutions need to satisfy constraints induced by decomposition structure and state transition system. The motivation behind the hierarchy is manifold: it allows the description of complex behavior (Erol, Nau, and Hendler 1994; Höller et al. 2014), enables the communication with users on different levels of abstraction (see e.g. Behnke et al., 2019) or to add advice, i. e., to guide the search using human domain knowledge (as e.g. exploited by Nau et al., 2003).

Traditionally, HTN planning systems relied mostly on the informativeness of the domain knowledge provided, making do with simple search algorithms like depth-first search (Nau et al. 2003). Recent work lines have established more informed search mechanisms, namely sophisticated reachability analysis (Bit-Monnot, Smith, and Do 2016; Behnke et al. 2020), heuristic search (Bercher, Keen, and Biundo 2014; Bercher et al. 2017; Höller et al. 2018), compilations to classical planning (Alford, Kuter, and Nau 2009; Alford et al. 2016), and SAT (Behnke, Höller, and Biundo 2018; 2019a; Schreiber et al. 2019). These methods improve HTN planning performance, make it less dependent on the domain knowledge, and provide it with the ability to search for low-

cost plans (sometimes with optimality guarantees depending on the technique used, see e.g. Behnke, Höller, and Biundo, 2019b). Here we investigate Monte-Carlo Tree Search (MCTS) as a new algorithmic alternative in HTN planning.

MCTS is a well-known search paradigm, originating in probabilistic planning problems (Kocsis and Szepesvári 2006; Keller and Helmert 2013), but successful also in games (Silver et al. 2016) and in deterministic planning settings (Trunda and Barták 2013; Schulte and Keller 2014). MCTS is essentially based on sampling the state space, feeding back rewards obtained at terminal states to inform the action decisions on earlier states. This converges to optimal decisions in the limit, but its main prowess in practice is to take good action decisions in limited time, with an anytime behavior where decisions improve over time. MCTS can also be combined with heuristic search, as systematized by Keller and Helmert (2013) in their trial-based heuristic tree search (THTS) framework. Here, the focus is to balance exploration and exploitation during search.

Our contribution here is the implementation and evaluation of MCTS in HTN planning. We do so in two HTN planning systems, namely Panda (Bercher, Keen, and Biundo 2014) and SHOP (Nau et al. 1999).

Panda is the state-of-the-art system incorporating heuristic search, thus facilitating a direct empirical comparison, and allowing to leverage prior work on heuristic functions (Höller et al. 2018). We implement and evaluate THTS.

SHOP (Nau et al. 1999; 2003) is the state-of-the-art system for lifted HTN planning, not requiring a grounded task representation. This serves to leverage the milder representation requirements of MCTS: in difference to heuristic search and SAT, MCTS does not require a grounded task representation. This is a key advantage because grounded representations quickly become prohibitively large when the object universe is large, and/or when predicate/action arity is high. This happens, for example, in natural language generation (Koller and Stone 2007; Koller and Hoffmann 2010) and in Minecraft planning (Roberts et al. 2017; Wichlacz, Torralba, and Hoffmann 2019). Implementing MCTS in JSHOP, a version of SHOP, with a number of adaptations for increased efficiency, we obtain the first non-blind HTN planner not requiring a grounded task representation. [1]

---

[1]Attempts have been made to compute heuristic functions with-

Evaluating our implementations on standard HTN benchmarks, we find that our new methods are generally on par with previous methods in terms of coverage, thus adding new technology options into the HTN arsenal. More specifically, in Panda, it turns out that the exploration term can yield benefits beyond purely relying on heuristics to guide search. In JSHOP, a naïve implementation of MCTS lags behind in coverage, but our adaptations – in particular, using backtracking on MCTS samples to escape shallow dead-end paths – remove that disadvantage.

To evaluate plan-cost minimization over time without guidance of a heuristic function, we create a version of JSHOP that continues search after the first plan is found. Comparing MCTS against this system, we find that MCTS is more effective at improving plan cost over time. This is especially the case on benchmarks aimed at featuring variance in action costs, including a number of Minecraft benchmark studies that we design for lifted planning.

The paper is organized as follows. Section 2 gives the required background in HTN planning, MCTS, and THTS. Section 3 describes our THTS implementation in Panda, which we empirically evaluate in Section 4. Section 5 describes our adaptation of MCTS in JSHOP, Section 6 gives its evaluation. Section 7 concludes the paper.

2 Background

In this section we give a brief introduction to HTN planning and Monte-Carlo Tree Search search.

HTN Planning

In Hierarchical Task Network (HTN) planning (see e.g. Ghallab, Nau, and Traverso, 2004, Chapter 11, or Bercher, Alford, and Höller, 2019), there are two types of tasks, *abstract* tasks and *primitive* tasks (also called *actions*). Like in classical planning, actions are directly applicable to the environment and cause state transition. In its most basic definition, the environment is described using a finite set of state features, but there are also systems supporting numbers (see e.g. Nau et al., 2003) and time (e.g. Bit-Monnot, Smith, and Do, 2016). Tasks are maintained in *task networks*, which are partially ordered multisets of tasks.

Abstract tasks are not directly executable, but are decomposed into other tasks (that may be primitive or abstract) using *decomposition methods*. A method definition includes the task it is applicable to and a task network that defines its *subtasks*. Usually, there is more than one method for a given abstract task, resulting in the combinatorial problem of choosing the right method to apply to a given task. When a method is applied to a task c in a task network tn, c is deleted, the method's subtasks are inserted and inherit the same ordering constraints to other tasks in the network tn that have been present for c. The process of decomposition is similar to the derivation of words in formal grammars (though the tasks are partially ordered).

out relying on a grounded task representation (Ridder and Fox 2014; Röger, Sievers, and Katz 2018). But this research is still in its infancy, and none of it is directly applicable to HTN planning.

The problem definition contains an initial task network tn_I and an initial state s_0. Some task network is a solution if and only if it can be derived from tn_I via the decomposition process as described before, it contains only primitive tasks, and there is a (totally ordered) sequence of the tasks in line with its ordering definition that is executable in s_0. There are two points that we want to emphasize:

- A planner is not allowed to insert additional actions apart from those included due to decomposition.

- Usually, there is no state-based goal definition given, the objective of the problem is defined in terms of the tasks contained in tn_I.

There are several restrictions that define subclasses of the given HTN definition (Bercher, Alford, and Höller 2019), mostly on the ordering definitions in methods or on recursion allowed in decomposition. The two systems we build on differ in terms of the HTN variants they deal with. Panda allows for recursive, partially ordered models. The version of SHOP we build on allows for recursion but is restricted to totally ordered models, i. e. models where all task networks in method definitions and the initial task network are totally ordered.

Monte-Carlo Tree Search (MCTS)

Monte-Carlo tree search (MCTS) is a family of algorithms (Browne et al. 2012) that have been very successful in Games and probabilistic planning on MDPs settings. Many variants of MCTS, or algorithms containing elements thereof, have been proposed. Generally, MCTS considers a search tree. This is initialized with a root node, then iteratively expanded, performing four operations at each step:

- Selection: Traverse the tree from the root to a leaf, selecting actions according to a strategy that balances exploration (visit nodes that have been explored less often) and exploitation (visit more promising nodes more often). The UCT algorithm defines a popular selection strategy taking inspiration from multi-armed bandit problems (Kocsis and Szepesvári 2006).

- Expansion: Generate the successors of the selected leaf node.

- Evaluation: Estimate the solution cost (reward/quality) of a newly generated node. This is done by roll-outs, simulations following a random action policy, until a terminal state is reached. (But a link to heuristic search can be made here, see below.)

- Backpropagation (backup): Update the cost estimates of the nodes traversed by the selection strategy, backwards from the leaf to the root. The most common strategy is to define a node's cost via the average cost of its children.

Trial-based Heuristic Tree Search (THTS)

As indicated, node evaluation can be linked to heuristic search. Instead of roll-outs, one can use a heuristic function – if available – to estimate node cost. This is an important feature of, e. g., the AlphaGo/Zero system series (Silver et

al. 2016; 2018). The relation to heuristic search was systematized by Keller and Helmert (2013) in trial-based heuristic tree search (THTS), an algorithm framework that merges MCTS with heuristic search.

THTS was originally proposed in the context of finite-horizon MDPs (Keller and Helmert 2013), and later applied in classical planning as well (Schulte and Keller 2014). Apart from replacing roll-outs with calls to a heuristic function, THTS defines a space of algorithm configurations, which includes traditional heuristic search algorithms. This pertains, foremost, to the choice of backup function, including in particular minimization which corresponds to the choice of open nodes in algorithms like A*. THTS furthermore distinguishes parameters governing exploration and greediness; we will give details in the next section when discussing our Panda implementation. Finally, drawing inspiration from earlier work (Bonet and Geffner 2003), THTS also features solved-labeling, avoiding exploration beneath nodes whose cost value is known to have converged.

3 MCTS in Panda

To enable the direct comparison of MCTS-based HTN planning with heuristic search, we implemented the THTS framework in Panda. Specifically, we build on THTS for classical planning as presented by Schulte and Keller (2014).

To adapt THTS to the HTN setting, nodes in our search tree have the current task network as additional element. That is, they are defined as $n = \langle s, tn, N, f, v, l \rangle$, containing the current state s, the current task network tn, the set of successor nodes N, a value estimate f, the number of visits v, and a label l storing whether the node is solved (should not be searched anymore).

Given this data structure, THTS applies mostly straightforwardly, as we shall spell out below. A key change though pertains to node initialization, due to the different nature of forward search in classical planning vs. HTN: whereas in the former there is a single successor for each (applicable) action, in the latter the successor function is more complex due to the more complex structure of task networks.

---

1 **procedure** INITIALIZENODE (*node n*)
2 $SN \leftarrow$ SUCCESSORS (n)
3   **for** $(s', tn') \in SN$ **do**
4 $n' \leftarrow \langle s', tn', \emptyset, w \cdot h(s', tn'), 1, goal(s', tn') \rangle$

5   **if** n' *is no dead end* **then**
6 $N(n) \leftarrow N(n) \cup \{n'\}$
7   **if** n' *is goal node* **then**
8   terminate // for agile setting
Algorithm 1: Initialization function

Consider Algorithm 1. Our SUCCESSORS (n) function (line 2) generates the successor nodes that the HTN progression search of Panda would generate: as usual in HTN progression search, only tasks without predecessor in the ordering definition of the task network are processed (so-called *unconstrained* tasks). If there is at least a single unconstrained *abstract* task left, one is picked (without branch-

ing) that is further processed. Since there will (usually) be more than one method applicable to that abstract task, processing a single abstract task might result in more than one successor node in the tree, one for each method applicable to it. The resulting combinations of state and task network are inserted in the set of successor nodes SN. When no unconstrained abstract tasks are left, the algorithm branches over all unconstrained (applicable) primitive tasks, i. e. one successor is generated and added to SN for each such task.

As heuristic function (line 4), we use the *Relaxed Composition* (RC) heuristic, which relaxes the HTN model to a classical model capturing the state of the HTN as well as relaxed hierarchy information. This relaxed model is used for heuristic calculation (Höller et al. 2018; 2019), and can be combined with several heuristics from classical planning. Here we use h^{FF} (Hoffmann and Nebel 2001), which performed best in the experiments of Höller et al. (2019). The parameter w (line 4) sets the weight of the heuristic relative to path cost, as in Weighted A*. Since we are primarily interested in an agile setting (finding any solution as quickly as possible) we move the goal test to node generation.

Being in the setting of classical planning, Schulte and Keller proposed to use loop detection. In our partially ordered HTN setting, this is not easily possible (see Behnke, Höller, and Biundo, 2015), so we have not integrated it.

To balance exploration and exploitation, THTS combines traditional heuristic search with UCT. First, the action selection function minimizes costs, instead of averaging. Let $N(n)$ be the successors of a node in the tree and $c(n, n')$ the cost of the operator applied to transform the search node n into n'. Then action selection is defined by

$$\underset{n' \in N(n): \neg l(n')}{\arg \min} \quad \bar{f}(n') - C \cdot \sqrt{\frac{log\, v(n)}{v(n')}} \qquad (1)$$

Here, $\bar{f}(n')$ is the sum of the child's f value and the path costs $k \times c(n, n')$, where k is a parameter weighing the path costs. The sum is normalized to a value in $[0, 1]$. C is a parameter trading off exploration vs. exploitation.

Second, while MCTS traditionally uses a backup operator defining the value of a node as the average over its children, THTS for deterministic planning allows to define a node's value via the minimum-value child instead. Specifically, the traditional backup function uses

$$\frac{\sum_{n' \in N(n)} (v(n') \cdot (f(n') + k \cdot c(n, n')))}{\sum_{n' \in N(n)} v(n')} \qquad (2)$$

while the alternative backup uses

$$\min_{n' \in N(n)} f(n') + k \cdot c(n, n') \qquad (3)$$

Following Schulte and Keller, we instantiate the framework parameters with the following value combinations, yielding the algorithm configurations used in our evaluation:

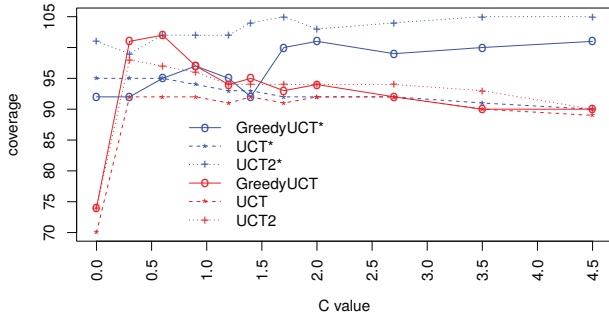| Name	Backup	k	w
UCT*	Eq. 3 (min.)	1	1
UCT2*	Eq. 3 (min.)	1	2
GreedyUCT*	Eq. 3 (min.)	0	1
UCT	Eq. 2 (avg.)	1	1
UCT2	Eq. 2 (avg.)	1	2
GreedyUCT	Eq. 2 (avg.)	0	1

Figure 1: Coverage for different C values.

4 Experiments with Panda

We evaluate our system on a benchmark set used in several recent papers on HTN planning (e.g. in (Höller et al. 2018; 2019; Behnke, Höller, and Biundo 2019b)). It includes 144 problem instances in 8 domains. Experiments ran on Xeon E5-2660 CPUs with 2.60 GHz base frequency and 10 min. time limit.

Figure 1 shows the coverage of our configurations. The number of solved instances (on the y-axis) is given for different weights of the exploration term (the C value, given on the x-axis). Configurations using the classical UCT backup (calculating the average of the children) are given in red, those using the minimum in blue. Like in the experiments of Schulte and Keller, the latter perform slightly better. The configuration weighting the heuristic with 2 and incorporating path costs performs best. It is similar to a Weighted A* search, so the result is in line with results for heuristic search-based HTN planning (see e. g. Höller et al., 2018).

When setting the C value to 0, the system behaves similar to a priority queue-based search. However, the tree, i. e. the data structure storing the search nodes, is not optimized to this special case (there is e.g. some data left in the tree when a node has already been processed; the tree is not balanced in any way; etc.). When we compare this configuration to the original Panda, we lose 5 instances of coverage. Interestingly, the system benefits from a higher C value, having its highest coverage for a C value of at least 1.7. This value is close to $\sqrt{2}$, the weight proposed in the original paper (however there are even configurations with higher C value and the same coverage). To see whether a similar behavior can be reached by adapting the weight in the Weighted A* search of the original Panda system we tested it with different weights. The results are given in Figure 2. Like in previous experiments with the Panda system, a weight of 2 reaches the highest coverage. So the exploration term adds a novel means to control the search.

Over all configurations using Eq. 3 (min.) as backup function, the results in the ENTERTAINMENT domain suffer from increasing the exploration factor, while for the transport domain the UCT* and GreedyUCT* configurations solve more instances (leading to a drop in the middle of the curves).

Schulte and Keller did their evaluation of C values for Greedy Best First search, because this is the configuration performing best in their evaluation. Here, their peek was be-
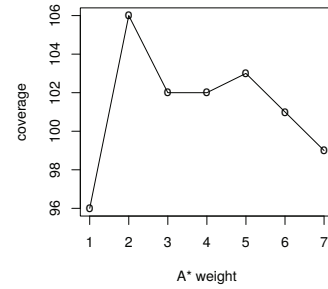

Figure 2: Coverage of the original Panda system with the RC heuristic combined with FF and Weighted A* search.

	#instances	$UCT2^\star$ with $C = 1.7$	$UCT2$ with $C = 0.6$	$Panda_{pro}, RC^{FF}$	$Panda, SAT$	$Panda_{ps}, TDG_M$	$Panda_{ps}, TDG_C$	$2ADL$	$JSHOP2$
ENTERTAINMENT	12	9	8	11	**12**	9	11	5	5
PCP	17	**14**	**14**	**14**	12	8	8	3	0
SATELLITE	25	**25**	**25**	**25**	**25**	24	21	23	22
SMARTPHONE	7	5	5	5	**7**	6	5	6	4
UM-TRANSLOG	22	**22**	**22**	**22**	**22**	**22**	**22**	19	**22**
WOODWORKING	11	10	10	10	**11**	8	10	5	8
ROVER	20	5	4	4	**10**	4	6	5	3
TRANSPORT	30	15	10	15	**22**	1	1	19	0
total	144	105	98	106	**121**	82	84	85	64

Table 1: Coverage of different HTN planning systems.

tween 0.6 and 0.9, i. e. lower than ours.

When using the UCT backup function, Greedy search performs best. This is uncommon in HTN planning (though quite common in classical planning). The peek is somewhere between 0.3 and 0.6, i. e. close to the one of Schulte and Keller. However, the difference between the best performing configurations of the two update functions is mainly caused by a single domain (TRANSPORT, see Table 1).

When we compare our search to other HTN planning systems (Table 1), we see that it is competitive with other search-based systems in terms of coverage, but (like in other evaluations) the translation to propositional logic (Behnke, Höller, and Biundo 2019a) performs best. The best search-based system is the original Panda progression search, followed by ours, the translation to classical planning by Alford et al. (2016) in combination with JASPER (Xie, Müller, and Holte 2014) and the plan space-based search of Panda with its TDG heuristics (Bercher et al. 2017). We further included the JSHOP2 (Nau et al. 2003) system. Note that this is not the same system we based our system in Section 5 on, but the one that can deal with partially ordered planning tasks. However, JSHOP2 can only be seen as a baseline, since it has been made to be used with hand-tailored planning problems and is used on problems intended for domain-independent systems.

Figure 3 shows how the number of solved instances evolves over time. It can be seen that our new search behaves similar to the Weighted A* search of the Panda system.
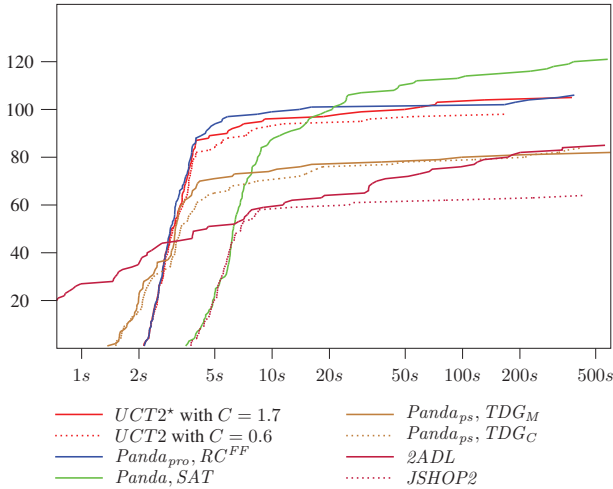
Figure 3: Accumulated number of solved instances (on the y-axis) over time (on the x-axis, be aware of the log scale).

It is known from previous evaluations that the cost structure of this benchmark set does not lead to a high variation in solution costs (see Behnke, Höller, and Biundo, 2019b). Therefore we do not present plots on solution costs here.

5   MCTS in JSHOP

We now consider SHOP (Nau et al. 1999; 2003), a lifted planner which allows to address planning tasks whose grounded representations would be infeasible to build. There are several variants and implementations of SHOP. We build on the Java JSHOP version for implementation reasons, and specifically on the first version of JSHOP, as the second version focuses on generation of instance-specific planners (Ilghami and Nau 2003) and it is harder to build on.

We use traditional MCTS, denoted by $\mathcal{M}$, evaluating nodes through roll-outs, because the state-of-the-art heuristic functions in planning rely on grounded encodings. A roll-out in our context means to perform a random sequence of task decomposition steps, until terminating either in a goal node, or in a dead-end node where no more decomposition steps are possible.

A few words are in order regarding what to expect from roll-outs in HTN planning. In general, random roll-outs are not well suited for scenarios where meaningful experiences are sparse and hard to find. Classical planning is a prime example for this, as MCTS will not learn anything about plan cost until it finds a plan, which with random action choices is typically exceedingly unlikely (Trunda and Barták 2013).

This problem, however, is arguably less daunting in HTN planning, where the task hierarchy can effectively be used to introduce domain control knowledge guiding the search towards valid plans – as evidenced by the fact that SHOP is a highly successful planner based on blind depth-first search. In this sense, MCTS and HTN planning seem to be made for each other. We see this as a highly promising combination to address challenging planning problems too large to ground, where the hierarchy provides enough guidance to find solutions easily and MCTS is used as a tool for plan-cost minimization.

Our primary target are thus HTN planning scenarios where the user can provide good-quality guidance towards solutions in the model, but where finding low-cost plans is a challenge. Minecraft instruction planning is one such scenario (Wichlacz, Torralba, and Hoffmann 2019), and we will run experiments with several benchmarks from that scenario in the next section.

That said, even within such scenarios, some algorithmic optimizations are required to obtain good performance in lifted MCTS HTN Planning. These are explained in the following.

Selection Strategy   As selection strategy, we use the standard UCT formula:

$$\arg\max_{n' \in N(n)} \frac{R^*}{r(n')} + C \cdot \sqrt{\frac{log\,v(n)}{v(n')}} \qquad (4)$$

The reward of each node is inversely proportional to the average cost of the plans found by roll-outs underneath, $r(n')$, normalized between 0 and 1 according to the cost of the incumbent solution, R^*. This normalization ensures that the constant C for balancing exploration and exploitation can be chosen in a domain-independent way, since it does not depend on the cost of the plans. Nodes for which no plan has been found are assigned a reward of 0 (i.e., $r(n') = \infty$), so if no incumbent plan has been found yet, all nodes have the same reward. Note that this ignores any roll-out that ends in a dead-end. Other strategies could be tried, e.g., by assigning a large penalty for roll-outs that failed to find any solution. However, this was not a problem in our experiments where, if a solution is found at all, then the hierarchy guidance towards solutions is good enough so that most of the roll-outs succeed in finding a solution. Moreover, this cannot occur if depth-first search roll-outs are used (see below).

Eliminating nodes with a single child   Nodes with a single successor are very common in totally ordered HTN benchmarks, since there is no choice of which task to perform next and tasks can sometimes be decomposed with only a single method in the current state. These are not problematic for other search algorithms like depth-first search or best-first search where each node is expanded only once, but MCTS has to traverse the tree at every iteration and update the statistics of every visited node, so nodes without any branching unnecessarily slow down the algorithm. To avoid this, we eliminate nodes without multiple siblings during search. Whenever a parent node n has a single child n', we proceed to generate the children of n' and so forth, until a node with several children is found. These children are then set as the children of n, and the intermediate nodes are dismissed. We denote this configuration as $\mathcal{M}_e$.

Node caching   On top of eliminating nodes with a single child, we also store all nodes explored during the roll-outs, and mark them as fully explored if all their children have

been fully explored, as done in the THTS framework. This helps to avoid redundant work, making it impossible to repeat the same simulation, and the memory overhead was not a limiting factor in our experiments. This combined configuration is named $\mathcal{M}_{e,c}$.

Depth-first search roll-outs In many HTN benchmark domains, there are "shallow" dead-ends, due to method decompositions that incorporate a task which cannot be decomposed into a fully-ground sequence of actions. Such methods are commonly pruned during grounding, but they can pose a problem for lifted planners. In MCTS they become a problem whenever a roll-out fails by choosing one such decomposition. To avoid this, we allow the roll-outs to backtrack when they reach a dead-end. That is, each roll-out performs a depth-first search for a solution, guaranteeing to terminate with either a plan, or a proof that the current MCTS leaf node is a dead-end and can be pruned. Note that such a strategy would be completely hopeless in classical planning or other similar search problems. Yet it turns out to be quite useful in HTN planning, cf. our discussion above.

Upper-bound pruning Once a first solution has been found, we have an incumbent solution that can be used for pruning nodes whose path cost is greater than that of the best solution found so far. We also forbid the roll-outs to exceed this cost, backtracking immediately in this case.

6   Experiments with JSHOP

As baseline we run the depth-first search of JSHOP ($\mathcal{J}$) and its enhanced variant with upper-bound pruning ($\mathcal{J}_P$). JSHOP supports only totally-ordered HTN problems, so we use the benchmark sets by Behnke, Höller, and Biundo (2018) and Schreiber et al. (2019). We also add several variants of the Minecraft domain (Wichlacz, Torralba, and Hoffmann 2019), where the task is to instruct a human user how to construct a high-level object in Minecraft. This Minecraft domain features two important characteristics: (1) the world has a huge size, so grounded approaches cannot deal with it well; and (2) while finding a solution is easy, finding one with good quality is particularly hard because it has a complex state-dependent action cost function where the cost of each instruction approximates how hard it is to be expressed in natural language and understood by the user and this may depend on previous instructions.

We consider three variants of the domain that differ on the high-level object that must be built: a house, a flat bridge (beam bridge), or an arch bridge. In each scenario, we consider two variants depending on the instructions that are available to our system: Construction plans may only use "place block" actions that would result in instructions similar to "Place a block at coordinates (3, 5, 4).", whereas instruction plans can additionally use other high-level actions like "build wall" that would lead to an instruction like "Build a wall of height 4 and length 3 southwards starting at the coordinates (3, 5, 4)".

Experiments ran on Xeon E5-2650 CPUs with 2.30 GHz and a time limit of 30 minutes and a memory limit of 4 GB.

Domain	$\mathcal{J}$	$\mathcal{M}$	$\mathcal{M}_e$	$\mathcal{M}_{e,c}$	$\mathcal{M}_{e,c,dfs}$
Blocksworld (20)	20	9	10	10	20
Childsnack (20)	18	0	0	0	17
Depots (20)	20	9	9	9	20
Entertainment (12)	3	1	1	1	3
Gripper(20)	20	20	20	20	20
Rover (40)	35	16	16	16	36
Satellite (36)	26	5	5	5	23
SmartPhone (7)	7	6	6	6	7
Transport (30)	0	0	0	0	0
UM-Translog (22)	21	21	21	21	21
Woodworking (11)	6	5	5	5	6
Total (238)	182	109	112	126	180
M-Bridge-const (20)	20	20	20	20	20
M-Bridge-inst (20)	20	20	20	20	20
M-Bridge-stairs-const (20)	19	17	18	20	20
M-Bridge-stairs-inst (20)	20	18	17	20	20
M-House-const (20)	13	13	13	20	20
M-House-inst (20)	12	12	12	20	20
Total Minecraft (120)	104	100	100	120	120
Total (358)	286	209	212	246	300

Table 2: Coverage on standard HTN benchmarks (above) and the Minecraft domain (below).

Coverage Analysis

Table 2 shows coverage results of the different algorithms. A vanilla MCTS implementation is not competitive with JSHOP depth-first search in all domains except Gripper and UM-Translog. The reason is that, in those domains, random roll-outs are not very likely to find any solution due to shallow dead-ends that can easily be avoided by the backtracking that depth-first search performs. Eliminating states with only a single child improves the coverage slightly, since this speeds up the traversal of the MCTS tree, but grants no additional information to guide the search. By also saving the states generated during the random roll-outs, and thereby pruning the fully explored parts of the state space much earlier, improves our coverage more significantly. The best results with regards to coverage are achieved when incorporating the backtracking also performed by depth-first search into the roll-outs, which is important to avoid the shallow dead-ends. The results of upper-bound pruning do not affect the coverage, as it only takes effect after the first solution is found. In the Minecraft variants, all method decompositions and primitive actions that can be generated lead to a valid plan, so it is trivially solved by depth-first search. The difference in coverage is due to an implementation detail that keeps all the relevant information about the states and task networks on the stack for JSHOP, $\mathcal{M}$ and $\mathcal{M}_e$ while the other configurations keep this information on the heap. For the larger instances of the Minecraft domains, the depth of the search tree grows so large that this leads to a stack overflow.
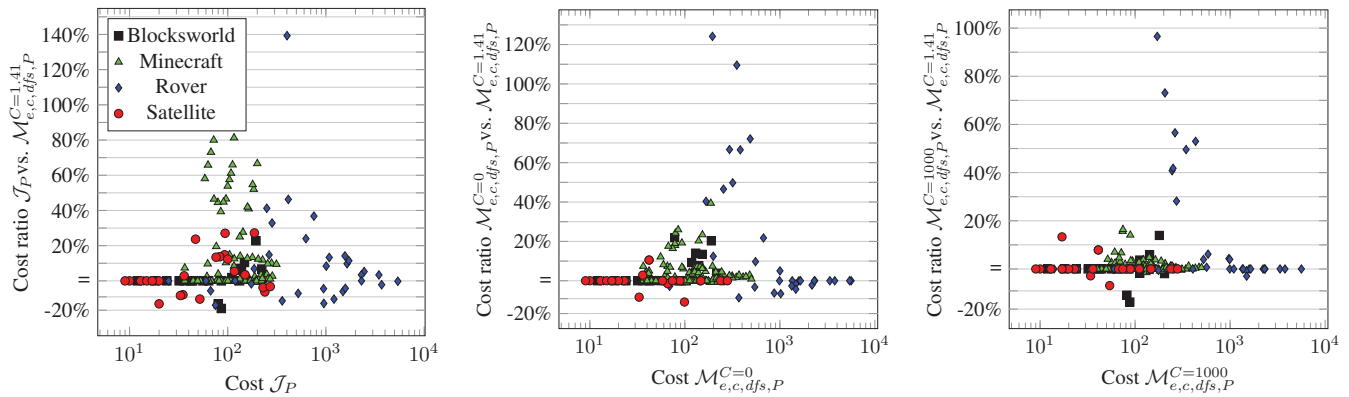
Figure 4: Comparison of the best plan found after 30 minutes by MCTS ($\mathcal{M}_{e,c,dfs,P}^{C=1.41}$) against depth-first search ($\mathcal{J}_P$), and two variants of MCTS only doing exploitation ($\mathcal{M}_{e,c,dfs,P}^{C=0}$), and only doing exploration ($\mathcal{M}_{e,c,dfs,P}^{C=1000}$). The x-axis shows the cost of the plan found by the baseline, which is representative of the problem size. The y-axis shows the ratio of improvement in plan cost. Points above the "=" line represent cases where the solution found by $\mathcal{M}_{e,c,dfs,P}^{C=1.41}$ has better quality than the one found by $\mathcal{J}_P$, $\mathcal{M}_{e,c,dfs,P}^{C=0}$, and $\mathcal{M}_{e,c,dfs,P}^{C=1000}$, respectively.

Cost Minimization

Next, we analyze the cost of the solutions found by JSHOP and MCTS. We report results on Blocksworld, Rovers, Satellite, and the different variants of Minecraft. We omit other domains where there are no solutions of different quality and all algorithms always find plans of the same cost. Figure 4 shows a high-level comparison of the algorithms in terms of the best solution found after 30 minutes.

The comparison of $\mathcal{J}_P$ and $\mathcal{M}_{e,c,dfs,P}^{C=1.41}$ shows that MCTS is indeed able to find better quality plans in general. The reason is that depth-first search strategies commit to decisions made at the top of the search tree, so any early decision that leads to plans of higher cost cannot be easily corrected until the entire sub-tree is explored. Nevertheless, depth-first search finds better solutions in a few instances. MCTS has a larger advantage in instances of middle-size, large enough so that depth-first search is not able to cover big portions of the state space or even exhaust it completely; and small enough so that MCTS can perform a sufficient number of roll-outs to allow for informed exploration.

Since arguably the main advantage of MCTS over depth-first search is that the latter cannot correct early mistakes in the search tree, one might consider whether MCTS is better than depth-first search with random restarts. To test this, we evaluate MCTS under different exploration factors, by choosing different values of C in the UCT formula. As default, we chose $C = \sqrt{2} = 1.41\ldots$, which is often considered a good default value for UCT. The configuration with $C = 0$ performs no exploration, exploring each child once and then always choosing the child who obtained a best plan in the first iteration. The configuration with $C = 1000$ performs no exploitation, always choosing to explore one child among the ones that have been selected the least amount of times. As shown in Figure 4, MCTS obtains significantly better solutions when balancing exploration and exploitation.

One advantage of our algorithms is that they have an anytime behavior, finding an incumbent solution quickly and improving its quality over time. Figure 5 compares different algorithms based on how the solution quality improves over time. We compare depth-first search with and without pruning ($\mathcal{J}_P/\mathcal{J}$), MCTS without pruning, and MCTS with pruning and several values of C: 0, 0.6 1, 1.41, 2, and 1000.

The plots show the average cost over all instances of the domain, reducing the noise caused by randomness of the algorithms. To have meaningful aggregated statistics, we only report data after all instances have been solved, excluding those instances whose first incumbent solution is not found in less than 30 seconds. Plots of individual instances show that the cost of the first solution is similar for all algorithms, but MCTS improves the quality of the plans much faster than the depth-first search configuration of JSHOP.

Pruning based on the cost of the incumbent solution is always helpful for depth-first search, and it is generally helpful for MCTS except on Blocksworld. The plots show that balancing exploration and exploitation is often necessary to obtain the best results. The configuration that does not perform enough exploration ($C = 0$) does not do a lot better than depth-first search, since it also has trouble recovering from bad decisions near the root of the tree. The configuration performing only exploration ($C = 1000$) behaves better thanks to re-starting the random roll-outs from different nodes in the tree. However, it does not make use of the information collected by previous MCTS runs so it generally takes more time to find solutions of good quality than configurations with smaller values of C that successfully balance exploration and exploitation. The best choice of C depends on the domain, e.g., slightly larger values of 1.41 and 2 seem to do well on Blocksworld and Rovers, where as smaller values (0.6) are preferred in Satellite and Minecraft. However, the difference is not too large compared against using depth-first search, or the extreme choices of C (0, 1000), suggesting that the algorithms' performance is robust with respect
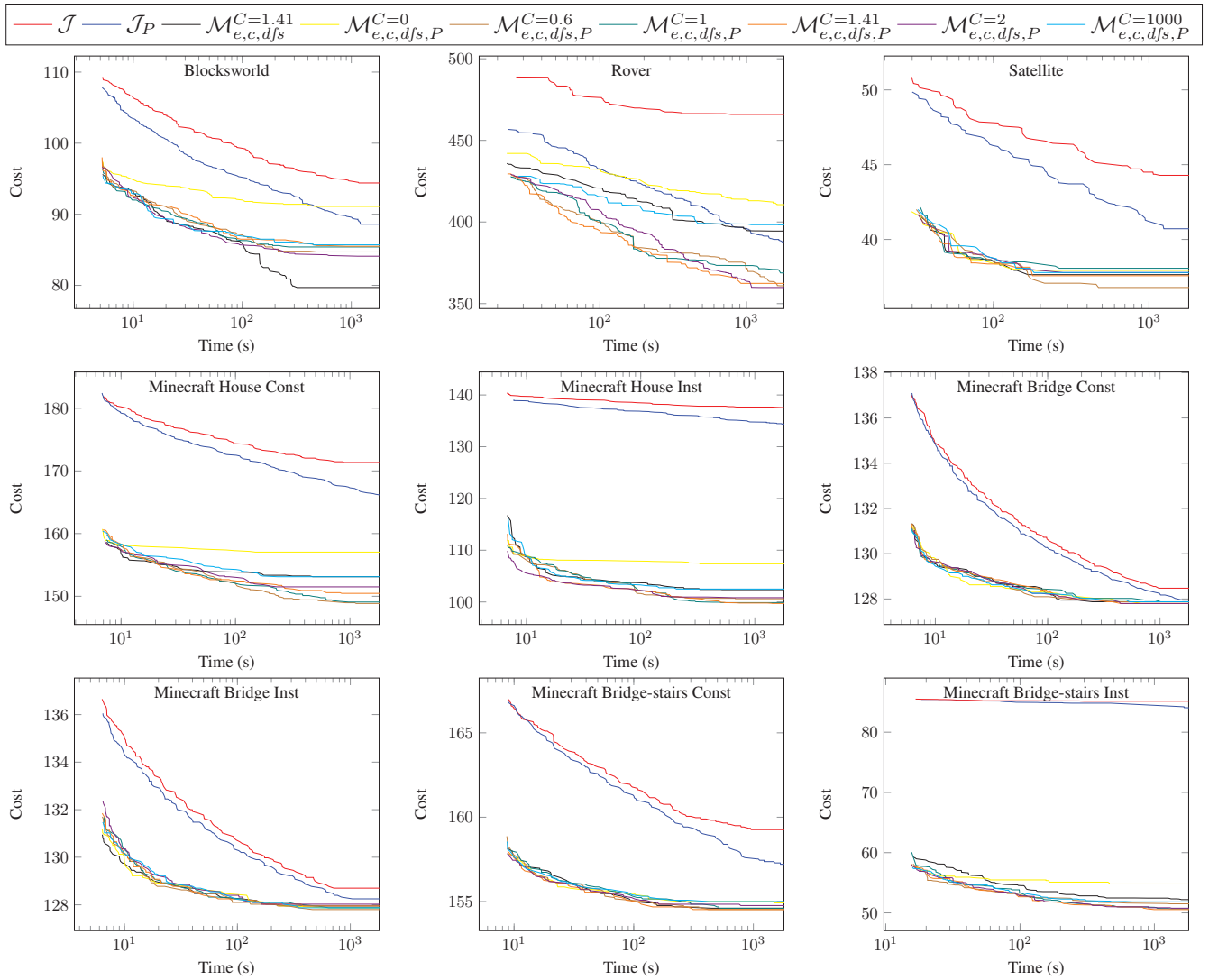
Figure 5: Improvement of cost over time of JSHOP and MCTS with/without upper bound pruning (above) and MCTS with pruning with different values for the exploration factor. Cost is the average cost on all instances where all configurations found an incumbent solution in less than 30 seconds.

to reasonable choices of the C parameter.

7 Conclusion

We have evaluated the performance of MCTS in two different HTN planning settings. In the THTS setting with focus on balancing exploration and exploitation, the approach is competitive with the state of the art in heuristic search planning on a standard benchmark set. However, like other search-based systems it is outperformed by SAT-based solvers. Most interestingly, the overall performance benefits from including a certain amount of exploration (i. e. increasing the C-value) instead of always following the advice provided by the heuristic function.

In the lifted HTN setting, we can see that an unmodified version of MCTS is not competitive, but once we include simple modifications common to HTN planning our cover-

age is on par with the JSHOP planning system. The real advantage of MCTS can be seen in the cost minimization once a first plan was found without depending on heuristics (that would require a grounded representation). Here MCTS in general outperforms JSHOP in terms of solution cost and is especially effective in mid-sized problems where many rollouts can be performed within the time limit. The choice of C-value only has a mild impact on the performance of our algorithm, as long as it is chosen in a reasonable fashion.

All this suggests that MCTS is a reasonable alternative to other HTN planning methods, especially in lifted HTN planning, where it can help to find better solutions faster.

Acknowledgments

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 *Proc. ICAPS*, 20–28.

Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proc. IJCAI*, 1629–1634.

Behnke, G.; Schiller, M. R. G.; Kraus, M.; Bercher, P.; Schmautz, M.; Dorna, M.; Dambier, M.; Minker, W.; Glimm, B.; and Biundo, S. 2019. Alice in DIY wonderland or: Instructing novice users on how to use tools in DIY projects. *AI Communications* 32(1):31–57.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On succinct groundings of HTN planning problems. In *Proc. AAAI*.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *Proc. ICAPS*, 25–33.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – totally-ordered hierarchical planning through SAT. In *Proc. AAAI*, 6110–6118.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning. In *Proc. AAAI*, 7520–7529.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding optimal solutions in HTN planning – A SAT-based approach. In *Proc. IJCAI*, 5500–5508.

Bercher, P.; Alford, R.; and Höller, D. 2019. A survey on hierarchical planning – One abstract idea, many concrete realizations. In *Proc. IJCAI*, 6267–6275.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In *Proc. IJCAI*, 480–488.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. SOCS*, 35–43.

Bit-Monnot, A.; Smith, D. E.; and Do, M. 2016. Delete-free reachability analysis for temporal and hierarchical planning. In *Proc. ECAI*, 1698–1699.

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. ICAPS*, 12–21.

Browne, C.; Powley, E. J.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Liebana, D. P.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games* 4(1):1–43.

Erol, K.; Nau, D. S.; and Hendler, J. A. 1994. UMCP: A sound and complete planning procedure for hierarchical task-network planning. In *Proc. AIPS*, 249–254.

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

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

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. ECAI*, 447–452.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proc. ICAPS*, 114–122.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019. On guiding search in HTN planning with classical planning heuristics. In *Proc. IJCAI*, 6171–6175.

Ilghami, O., and Nau, D. S. 2003. A general approach to synthesize problem-specific planners. TR, Maryland University.

Keller, T., and Helmert, M. 2013. Trial-based heuristic tree search for finite horizon MDPs. In *Proc. ICAPS*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *Proc. ECML*, 282–293.

Koller, A., and Hoffmann, J. 2010. Waking up a sleeping rabbit: On natural-language sentence generation with FF. In *Proc. ICAPS*, 238–241.

Koller, A., and Stone, M. 2007. Sentence generation as planning. In *Proc. ACL*.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: simple hierarchical ordered planner. In *Proc. IJCAI*, 968–975.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Ridder, B., and Fox, M. 2014. Heuristic evaluation based on lifted relaxed planning graphs. In *Proc. ICAPS*, 244–252.

Roberts, M.; Piotrowski, W.; Bevan, P.; Aha, D.; Fox, M.; Long, D.; and Magazzeni, D. 2017. Automated planning with goal reasoning in Minecraft. In *Proc. ICAPS workshop on Integrated Execution of Planning and Acting*.

Röger, G.; Sievers, S.; and Katz, M. 2018. Symmetry-based task reduction for relaxed reachability analysis. In *Proc. ICAPS*, 208–217.

Schreiber, D.; Pellier, D.; Fiorino, H.; and Balyo, T. 2019. Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proc. ICAPS*, 382–390.

Schulte, T., and Keller, T. 2014. Balancing exploration and exploitation in classical planning. In *Proc. SOCS*, 139–147.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529:484–503.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science* 362(6419):1140–1144.

Trunda, O., and Barták, R. 2013. Using monte carlo tree search to solve planning problems in transportation domains. In *Proc. of MICAI*, 435–449.

Wichlacz, J.; Torralba, A.; and Hoffmann, J. 2019. Construction-planning models in Minecraft. In *Proc. ICAPS Workshop on Hierarchical Planning*, 1–5.

Xie, F.; Müller, M.; and Holte, R. 2014. Jasper: The art of exploration in Greedy Best First Search. In *Proc. IPC*, 39–42.