Karlsruher Institut für Technologie

Master's Thesis

Malleable Distributed Hierarchical Planning

Niko Wilhelm

Date of submission: October 31, 2022

Betreuer: Prof. Dr. Peter Sanders
M.Sc. Dominik Schreiber

Institut für Theoretische Informatik, Algorithmik
Fakultät für Informatik
Karlsruher Institut für Technologie

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den October 31, 2022

Zusammenfassung

Hier die deutsche Zusammenfassung.

Ich bin Blindtext. Von Geburt an. Es hat lange gedauert, bis ich begriffen habe, was es bedeutet, ein blinder Text zu sein: Man macht keinen Sinn. Man wirkt hier und da aus dem Zusammenhang gerissen. Oft wird man gar nicht erst gelesen. Aber bin ich deshalb ein schlechter Text? Ich weiß, dass ich nie die Chance haben werde im Stern zu erscheinen. Aber bin ich darum weniger wichtig? Ich bin blind! Aber ich bin gerne Text. Und sollten Sie mich jetzt tatsächlich zu Ende lesen, dann habe ich etwas geschafft, was den meisten „normalen“ Texten nicht gelingt.

Ich bin Blindtext. Von Geburt an. Es hat lange gedauert, bis ich begriffen habe, was es bedeutet, ein blinder Text zu sein: Man macht keinen Sinn. Man wirkt hier und da aus dem Zusammenhang gerissen. Oft wird man gar nicht erst gelesen. Aber bin ich deshalb ein schlechter Text? Ich weiß, dass ich nie die Chance haben werde im Stern zu erscheinen. Aber bin ich darum weniger wichtig? Ich bin blind! Aber ich bin gerne Text.

Abstract

And here an English translation of the German abstract.

I'm blind text. From birth. It took a long time until I realized what it means to be random text: You make no sense. You stand here and there out of context. Frequently, they do not even read. But I have a bad copy? I know that I will never have the chance of appearing in the. But I'm any less important? I'm blind! But I like to text. And you should see me now actually over, then I have accomplished something that is not possible in most “normal” copies.

I'm blind text. From birth. It took a long time until I realized what it means to be random text: You make no sense. You stand here and there out of context. Frequently, they do not even read. But I have a bad copy? I know that I will never have the chance of appearing in the. But I'm any less important? I'm blind! But I like to text.

Danksagungen

Thanks to i10pc135 which suffered much to make the experimental evaluation possible.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

e the
ability
r on
great
eature

1.2 Research Goal

- Provide a performant parallel TOHTN planner by improving upon the Crowd planner - improve the performance of CrowdHTN - provide an efficient malleable TOHTN planner - first it needs to be correct and complete - Provide integration of TOHTN into the Mallob malleable load balancer - Compare performance of parallel to malleable TOHTN planning

1.3 Thesis Overview

- preliminaries

- algorithmic improvements to CrowdHTN - explore

Terminology - PE: parallel unit in general - worker: Mallob worker specifically - job: job within Mallob - Mallob assigns a number of PEs to a job running a worker on each one

2 Preliminaries

2.1 Planner Properties

- soundness: all solutions we find are correct - completeness: we always find a plan if it exists - correctness: kinda obvious - systematicity ([22]): for search we explore each search node at most once (Kambhampati, Knoblock, & Yang, 1995) -> not a problem for TOHTN? check p.18 in holler2020 again - optimality: shortest plan (few planners, Lilotane maybe?)

2.2 TOHTN Formalism

In this section we first define what HTN and TOHTN problems are from a formal perspective 2.2.1. Afterwards we take a short look at the algorithmic worst case complexity of HTN and TOHTN planning 2.2.2.

2.2.1 Defining TOHTN Planning Problems

e more
alisms

Both HTN and TOHTN planning are based on the idea of decomposing a list of initial tasks down into smaller subtasks until those subtasks can be achieved by simple actions.

Multiple definitions for HTN planning exist. In this work we build on the definition introduced in [18].

Definition 1. *A **predicate** consists of two parts. Firstly a predicate symbol $p \in \mathcal{P}$ where $\mathcal{P}$ is the finite set of predicate symbols. Secondly of a list of terms $\tau_1, \ldots, \tau_k$ where each term τ_i is either a constant symbol $c \in \mathcal{C}$, with $\mathcal{C}$ being the finite set of constant symbols, or a variable symbol $v \in \mathcal{V}$, where $\mathcal{V}$ is the infinite set of variable symbols.*
The set of all predicates is called $\mathcal{Q}$.

With the definition of a predicate in place, we can then define a grounding as well as our world state.

Definition 2. *A **ground predicate** is a predicate where the terms contain no variable symbols or, in other words, a predicate that contains only constant symbols.*

Definition 3. *A **state** $s \in 2^{\mathcal{Q}}$ is a set of ground predicates for which we make the closed-world-assumption. Under the closed-world-assumption, only positive predicates are explicitly represented in s. All predicates not in s are implicitly negative.*

Definition 4. *With T_p the set of primitive task symbols, a **primitive task** t_p is defined as a triple $t_p(\tilde{t}_p(a_1, \ldots, a_k), pre(t_p), eff(t_p))$. $\tilde{p} \in T_p$ is the task symbol, $a_1, \ldots, a_k \in \mathcal{C} \cup \mathcal{V}$ are the task arguments, $pre(t_p) \in 2^{\mathcal{P}}$ the preconditions and $eff(t_p) \in 2^{\mathcal{P}}$ the effects of the primitive task t_p. We further define the positive and negative preconditions of t_p as $pre^+(t_p) := \{p \in pre(t_p) : p \text{ is positive}\}$ and $pre^-(t_p) := \{p \in pre(t_p) : p \text{ is negative}\}$. We define $eff^+(t_p)$ and $eff^-(t_p)$ analogously.*
*We call a fully ground primitive task an **action**.*

As preconditions and effects may not be concerned with the whole world state the closed-world assumption does not apply to them. To any HTN instance we could create an equivalent one where each precondition and effect cares about the whole world state. This would be achieved by instantiating all the "don't care" terms in preconditions and effects with all possible combinations of predicates. Doing this would, however, come at the price of a huge blowup of our planning problem.

Definition 5. *An action t_p is **applicable** in state s if $pre^+(t_p) \subseteq s$ and $pre^-(t_p) \cap s = \emptyset$. The **application** of t_p in state s results in the new state $s' = (s \setminus eff^-(t_p)) \cup eff^+(t_p)$.*

Definition 6. *We define a **compound task** as $t_c = \tilde{t}_c(a_1, \ldots, a_k)$, where $\tilde{t}_c \in T_c$ is the task symbol from the finite set of compound task symbols T_c and $a_1, \ldots, a_k$ are the task arguments.*

Primitive and compound tasks together form task networks. In places where both can be used, we will refer to them simply as tasks $t \in T$.

Definition 7. *Let $T = T_p \bigcup T_c$ be a set of primitive and compound tasks. A task network is a tuple $\tau = (T, \psi)$ consisting of tasks T and constraints ψ between those tasks.*

Definition 8. *Let M be a finite set of method symbols and $T = T_p \bigcup T_c$ a set of primitive and compound tasks. A **method** $m = (\tilde{m}(a_1, \ldots, a_k), t_c, pre(m), subtasks(m), constraints(m))$ is a tuple consisting of the method symbol $\tilde{m}$, the method arguments $a_1, \ldots, a_k$, the associated compound task $t_c \in T_c$ the method refers to, a set of preconditions $pre(m) \in 2^{\mathcal{P}}$, a set of tasks $subtasks(m) = \{t_1, \ldots, t_l\}, t_i \in T$ and a set of ordering constraints $c_1, \ldots, c_m$ defining relationships between the subtasks. Any arguments appearing in $t_c, pre(m), subtasks(m)$ must also appear in $a_1, \ldots, a_k$.*
In TOHTN planning, $constraints(m)$ is implicitly set s.t. the subtasks $t_1, \ldots, t_l$ are totally ordered.
*We call a fully ground method a **reduction**.*

Each method m has exactly one associated compound task t_c. However, multiple methods $m_1, \ldots, m_k$ may be associated with a single compound task t_c. Additionally, while any arguments of t_c must be present in m, the contrary is not true and m may have arguments not

link dire
to resol
of a tas
network
resolvin
defined
fore me

present in t_c, i.e., m is not fully determined by t_c. As a result methods present choice points both in the choice of method itself as well as through the argument instantiation.

Definition 9. *Let $\tau = (T, \psi)$ be a task network, s a state, $m = (\tilde{(}m)(a_1, \ldots, a_k), t_c, pre(m), subtasks(m), co$ be a method. m **resolves** τ iff $t_c \in T$, the constraints in ψ allow for t_c to be resolved, $pre^+(m) \in s$ and $pre^-(m) \cap s = \emptyset$.*
Resolving a compound task $t \in T$ results in a new task network $\tau' = ((T \setminus t) \cup \{t : t \in subtasks(m)\}, \psi \cup constraints(m))$ and state s.
Applying a primitive task results in a new task network $\tau' = (T \setminus t, \psi)$ in state s' where the effects of t have been applied to s.

Definition 10. *An **HTN domain** is a tuple $D = (V, C, P, T, M)$ consisting of finite sets variables V, constants C, predicates P, tasks T and methods M. An **HTN problem** $\Pi = (D, s_0, \tau_0)$ consists of a domain D, an initial state s_0 and an initial task network τ_0.*
*If $subtasks(m)$ has a total order for all $m \in M$ and the tasks in τ_0 are totally ordered, we speak of a **TOHTN domain** and **TOHTN problem**.*

It is possible to translate any HTN problem with initial task network τ_0 into an equivalent HTN problem with initial task network τ_0' s.t. τ_0' consists of only a single task.
It is possible to simplify the model s.t. τ_0 always consists of only a single task with no constraints. We do this by inserting a new initial task t_0 and method m_0 with no arguments s.t. resolving t_0 via m_0 results in τ_0.

2.2.2 Complexity of (TO)HTN planning

The complexity of HTN and TOHTN planning has been studied in many papers. Here the problem PLANEXIST describes, whether for any given (TO)HTN instance a plan exists at all. It is not concerned with optimality.
Early on it was shown by [13] and [14] that the complexity of hierarchical planning formalisms depends on things such as the existence and ordering of non-primitive tasks, whether a total order between tasks is imposed and whether variables are allowed. The combination of arbitrary non-primitive tasks, no total order imposed and allowing variables is what we talk about with HTN planning, the same combination but with a total order is what we mean with TOHTN planning. They showed that HTN planning is semi-decidable whereas TOHTN planning is decidable in D-EXPTIME while being EXPSPACE-hard.
We can see what D-EXPTIME means in practise when we consider the maximum size of a task network we need to consider. From [4] we know that if a solution to an HTN instance exists, it can be found within a maximum depth of $|T_c| \cdot (2^{|\mathcal{Q}|})^2$. Similarly, we see that a task network can have exponential width in it's depth. Consider for this an instance constructed such that each compound task has exactly two children and where primitive tasks are only occuring at the bottom most layer. Now each layer will be twice as wide as the one before, giving us exponential width.
Regarding the general relationship of hierarchical planning to complexity theory, [13] and [14] showed early on that HTN instances can be used to simulate context-free languages. This was extended by [20] who showed that TOHTN instances correspond exactly to context-free grammars.
In addition to planning itself, the problem of plan verification was studied. Here, [3] showed that plan verification is NP-complete, even under the assumption that not only the plan but also the decompositions leading to it are provided.

Figure 1: An example TOHTN domain

2.2.3 Differences from other Kinds of Planning

[27] creates a classification of planners into domain-specific, domain-independent and domain-configurable planners. They argue that HTN planning falls under domain-configurable with the decompositions providing advice to the planner to gain efficiency. [22] argue that HTN-planning is not simply a domain-configurable version of classical planning and argue on the basis that [13, 14] showed that HTN-planning is strictly more powerful compared to classical planning which is PSPACE-complete.
While we agree with [22], one can still use HTN planning without using the full complexity of the model, using it instead to provide more efficient and guided versions of classical planning problems.

2.2.4 Graphically Representing TOHTN Problems

In the rest of this work we will sometimes represent the structure of TOHTN domains graphically. We will always use the same scheme which we explain here. In our visualization, we present the structure of a domain as a series of methods. To the left we show the compound task which can be resolved by the method, followed by the method itself on its right and followed again by the method subtasks in their fixed order. Compound tasks are always represented by green rounded squares and their name, methods by an arrow with the method name above and actions by blue rounded squares and their name.
A short example is shown in figure 1. It consists of one method, m_1 which resolves task t_1. The subtasks of m_1 are t_2, a_1 and a_2. t_1 and t_2 are compound tasks, a_1 and a_2 are actions.

2.3 Techniques to solve TOHTN planning problems

In this section, we will give an overview over the different techniques with which HTN problems can be solved. The HTN planners produced by researchers can be classified along two main axes:

- the planning algorithm
- lifted vs grounded approaches

For the algorithms, the two main variations are translation-based algorithms that take an HTN instance and translate it into a problem in a simpler complexity class such as classical planning or SAT and search-based algorithms, that utilize techniques such as plan-space search and progression search. We will focus on progression search here, as it is the technique employed in our own planner, CrowdHTN.
After that we will have a short discussion on lifted versus grounded approaches which is largely independent of the search algorithm.

2.4 Translation-based

Title ins
as "SAT
based"?

One of the main techniques employed in HTN planning is to find an efficient encoding into a simpler problem. Two such problems are classical planning ([1]) and propositional logic (SAT). While translation to SAT was already proposed in 1998 ([25]), the first complete encodings without assumptions about the instance were publicized in 2018 ([4]). In recent years, SAT seems to be the most popular problem to translate an HTN instance into, utilized by planners such as totSAT ([4], [5]), Tree-REX ([31]) and Lilotane ([30]).
As we have seen in the previous section 2.2.2 on (TO)HTN complexity, (TO)HTN problems are in D-EXPTIME and undecidable respectively. Both classical planning and SAT are less powerful. As a result, HTN problems cannot be encoded and even for TOHTN problems we would suffer a blowup in the size of the instance. Instead, as noted in [31], SAT-based planners tend to explore the set of potential hierarchies layer by layer, increasing the encoding size as they go. As a result, those SAT-based planners tend to have a BFS-like characteristic to their search.

2.4.1 Search-based

The second main category of techniques to solve HTN planning problems are search-based algorithms, such as plan space search and progression search. Both of these are described in [22]. Plan space search, as the name says, searches the space of partial plans, aiming to fix flaws - take definitions from [22] unless noted otherwise - explain where we differ from HTN progression search (holler2020htn also does this)

Progression search on the other hand generates plans in a forward way. What is meant by this is that it always chooses an open task that is currently unconstrained, i.e. has no unresolved predecessors under the ordering constraints, and resolves it. This allows the planner to update the world state as it goes along, as the sequence of actions from the start to the current point is known at each step of the search. In case of TOHTN planning, the choice of the next unconstrained task becomes trivial, as there is always exactly one such task. Knowing the full world state gives progression search two main advantages over plan space search. First, it allows the planner to prune parts of the search space by immediately validating action and reduction preconditions against this world state. Second, it gives us maximum information to be used in heuristics that guide our search.
The progression search algorithm is given in pseudocode in algorithm 1. As mentioned, line 6 becomes trivial for TOHTN planning and the loop from lines 7 to 16 is no loop, as there is always exactly one unconstrained task. Additionally, the location of our goal test can be moved around, depending on need. Performing the goal test upon popping a node is useful if we want to find optimal plans and our fringe data structure - and thus popping order - have a notion of node cost. Performing the goal test upon node creation allows us to terminate earlier.
Notable search-based planners are SHOP ([26]), HyperTensioN ([24]), PANDA ([22]) and our own planner CrowdHTN which will be presented in a later section 2.6.

2.4.2 Lifted and Ground HTN Planning

As mentioned in section 2.2.1, (TO)HTN instances are normally given in a lifted representation and can be ground, i.e. all variables are filled with all possible parameter combinations. Specifying the instance in a lifted fashion is done for ease of use, as it is a more compact representation and allows domains to be reused for different problems [6].
The efficient grounding and pruning of HTN instances is an active field of research ([28], [6]). While it is an easier problem than (TO)HTN planning itself, the ground instance may still be exponential in size compared to the lifted instance [6].

Algorithm 1: Classical Progression Search for HTN as introduced in [22]

```
1  fringe ← {(s_0, tn_I, ϵ)}
2  while fringe ≠ ∅ do
3      n ← fringe.pop()
4      if n.isgoal then
5          return n
6      U ← n.unconstrainedNodes
7      for t ∈ U do
8          if isPrimitive(t) then
9              if isApplicable(t) then
10                 n' ← n.apply(t)
11                 fringe.add(n')
12         else
13             for m ∈ t.methods do
14                 n' ← n.decompose(t, m)
15                 fringe.add(n')
```

Planners may choose to operate on either lifted or ground instances. A discussion on the trade-offs involved is found in [30]. We will reiterate the main advantage of each approach here. Grounded representations have more information available for pruning. As an example, while some parameter combinations in reductions may lead to a contradiction later on and can thus be pruned, not all such combinations may be invalid and thus the corresponding lifted method may not be prunable. Lifted representations on the other hand may be a lot more compact in practice. For example, our (TO)HTN instance may want us to choose any of N trucks to transport a package from A to B where in practice the choice might not matter. Whereas a grounded representation will have to instantiate all operators concerning a truck N times, a lifted operation will avoid this and only choose a truck ad-hoc.
The choice of grounded vs lifted representation is independent of the choice of planning algorithm. We have examples of grounded translation-based planners (totSat [4], Tree-REX [31]), lifted translation-based planners (Lilotane [30]) and also search-based planners that work on both lifted (HyperTensioN [24]) and ground representations (PANDA [22]).
Our own planner, CrowdHTN, walks a middle ground . It performs its search on a ground representation to allow detailed pruning according to the world state. However, it does not front-load the cost of a grounding procedure and instead grounds tasks and methods as needed.

pun intended

2.5 Malleability

In this work, we follow the classification of [17] regarding parallel jobs. A job is **rigid** if it has a fixed number of required processing elements (PEs) which is hard-coded in the application. This number stays the same between runs. We call a job **moldable** if the number of PEs is variable and can be set at application start but remains fixed within any one run. An **evolving** job is one where the required number of PEs changes during execution and where these changes are initiated by the user. If the number of PEs changes during execution with the changes initiated externally, we call a job **malleable**. Malleability can be defined more generally as the ability to deal with changing resources, not only PEs [33]. In practise, we see that most jobs run on supercomputers follow the moldable model [12]. The moldable model is also supported

by programming environments such as MPI [23].
Systems that utilize malleable jobs have been shown to be highly efficient [17]. They achieve this in multiple ways. First, they allow for efficient scheduling, as the scheduler can reevaluate and change previously made decisions [33]. This allows to resolve the conflict in scheduling between throughput and response latency, where low latencies come with the need to keep spare resources on hand instead of fully utilizing them [17], [23]. Second, they allow applications to utilize additional resources as they become available, leading to improved performance [23]. Lastly, [11] make the case that malleable applications are more fault-tolerant which is of increasing importance as applications become more parallel.
While malleable jobs are desirable from a scheduling and administration perspective, they are not popular with the user side, as they impose additional complications [17]. The effort required to make any one application malleable varies depending on the problem. In case the problem at hand is easily split into independent small subtasks, we can use a central work queue from which other PEs can receive new tasks as needed [17], [34]. This approach allows us to redistribute PEs to other jobs in between tasks. It is however limited by the central work queue which tends to be a bottleneck. Alternatively, in data driven applications, we may have distributed data structures that are redistributed as the number of available PEs changes [17]. This is more complicated, though, and is an expensive operation which should not be performed too often. To sum it up, making an application malleable is highly dependent on the specific problem and only easy in cases that are trivial to parallelize.

2.6 The CrowdHTN Planner

The CrowdHTN (Cooperative randomized work stealing for Distributed HTN) planner is implemented as a parallel state machine that uses work stealing for work balancing purposes. According to the definition of [17] we introduced in section 2.5, CrowdHTN is a moldable task, as the number of workers is arbitrary but fixed during execution.
Each local worker of CrowdHTN owns it's own queue of search nodes, tuples of *open tasks* and *world state* and performs progression search on these nodes as explained in section 2.4.1. The basic CrowdHTN parallel search algorithm is shown in algorithm 2.
In the initial state, only the root worker has any search nodes. All other workers start empty. To perform load balancing, randomized work stealing is used . The work package exchange is implemented as a three step protocol

(i) work request
(ii) response
(iii) ack (if response was positive)

Upon sending a positive work response, a worker increments it's local tracker of outgoing work packages. When receiving an ack, the local tracker of outgoing work packages is decremented. This ensures that there is always at least one node that acknowledges the existence of each search node. This is used to enable CrowdHTN to determine a global UNPLAN. To do this each worker reports whether it has any work left. A worker reports true if it has a non-empty fringe or at least one outgoing work package.
This capability is especially helpful for small instances where it is plausible to explore the whole search space. As we saw in the earlier section on complexity (2.2.2), TOHTN planning is in D-EXPTIME making it infeasible to explore the whole search space for big instances.
The specifics of which node to explore for a work step and which node to split off to send a positive work response depend on the fringe implementation. In general, we perform work at the back end of the fringe, in the case of depth-first-search these are the last nodes to have been inserted. To split off work, we take from the front end of the fringe. This is done as a heuristic

to split off a node which is still far from a plan and thus forms a relatively larger work package, leading to less communication. In case of depth-first search, this reduction of communication volume is doubly true. Nodes which are close to the beginning of our search path generally have fewer open tasks, reducing the size of the message when sending it off.

Algorithm 2: The parallel CrowdHTN algorithm

```
1  while true do
2      work_step()
3      if fringe.empty and not has_active_work_request then
4          r ← random worker id
5          send work request(r)
6          has_active_work_request ← true
7      for (message, source) ∈ incoming messages do
8          if message is work request then
9              if fringe.has_work() then
10                 send positive work response(fringe.get_work(), source)
11                 outgoing work messages += 1
12             else
13                 send negative work response(source)
14         if message is work response then
15             if response is positive then
16                 fringe.add(work response)
17                 send work ack(source)
18             has_active_work_request ← false
19         if message is work ack then
20             outgoing work messages -= 1
```

2.7 Mallob

Mallob is both the **Mal**leable **Lo**ad **B**alancer and the **M**ulti-tasking **A**gil**e** **Lo**gic **B**lackbox [32]. Mallob provides a malleable scheduler which focuses on (NP-)hard jobs with unknown processing times, where the jobs themselves can be small while still being hard [29].

In this it focuses on the problem of propositional logic (SAT). - won multiple prices in the international SAT competition - Mallob has been referred to as “by a wide margin, the most powerful SAT solver on the planet”

The following overview of Mallob is taken from [32]. Mallob is a decentralized, distributed malleable job scheduler and load balancer. Being distributed, it does not assume any shared memory and instead has the different workers communicate via message passing. As a scheduler it is able to solve multiple jobs in parallel and adjusts the resources available per job on a dynamic basis. New jobs j can be introduced to Mallob at any times and are described by a number of attributes. Among those, each job has a fixed *priority* $p_j \in (0, 1)$. Additionally, each job has a variable resource *demand* $d_j \in \mathbb{N}$, describing the maximum number of PEs that job j is able to utilize efficiently. In the trivial case, assuming jobs only happen one after the other, each job can simply set d_j to the total number of available PEs. When it comes to the total number of active jobs at any one time, Mallob assumes that their number is lower than

the total number of PEs. This allows Mallob to assign each PE to only a single job at a time while still making progress on all active jobs. The total number of PEs assigned to a job is also called the job's *volume* v_j. The volume of each job is set proportional to $d_j p_j / \sum_{j'} d_{j'} p_{j'}$, i.e., proportional to the product of a job's demand and priority.
The v_j workers currently assigned to a job are internally organized as a binary tree, s.t. all levels except the last one of the tree are always full and the last level is filled from left to right. If a PE is taken away from a job, the associated data is not immediately deleted. Instead, a small and constant number of previous jobs is kept around. When the volume v_j grows again, PEs containing a suspended worker of the same job are preferred to increase efficiency.

related work: - distributed search: hash distributed a star

- paracooba as malleable SAT solver - mallob as malleable SAT solver

- parallel planners: SHIP, Crowd, Mallotane (Lilotane + Mallob, limited as only the SAT solving is malleable) - in the wider sense: applications that internally rely on SAT - few malleable applications exist as it puts a strain on developers

- not aware of any other work in malleable (TO)HTN planning

3 Theoretical Improvements of the CrowdHTN Planner

In this section we will discuss improvements to the algorithms underlying CrowdHTN. We start by introducing different search algorithms in 3.1 and offer a discussion regarding their completeness. In 3.2 we give an overview of loop detection mechanisms in other hierarchical planners, discuss loop detection as it was already present in CrowdHTN and present our design of a distributed loop detection mechanism based on bloom filters.

3.1 Search Algorithms Used in CrowdHTN

As part of the re-engineering of CrowdHTN, we changed the implementation of the search algorithms to be based on a fringe. As mentioned in [22] we can simply switch out the underlying fringe data structure to emulate different search algorithms without making any changes to our core planner. Enabled by this change, we have implemented four search algorithms and will discuss them in the following section:

- Random depth-first search (DFS)
- Random breadth-first search (BFS)
- Heuristic depth-first search
- A-star like search

3.1.1 Random Depth-First Search

Random DFS is the only search algorithm that was already present in the previous implementation of CrowdHTN. It is implemented using a Last-In-First-Out queue as our fringe. Whenever new search nodes are inserted into the fringe we randomize their order. This is done to avoid any pathological cases a fixed order may induce, such as getting stuck in an infinite loop if we always resolve an open task t to itself. We do not expect any differences in behavior or performance compared to the previous implementation.

3.1.2 Random Breadth-First Search

Random BFS is the first new search algorithm that we implemented. It is done by using a First-In-First-Out queue, allowing us to explore all the potential task hierarchies layer by layer. The insertion order of new nodes is randomized as in the DFS. We do this as the number of search nodes per layer can be exponential in the depth. As such, the last layer may dominate the overall work and the order in which we explore it can have a large impact on performance. In general, we expect a higher memory footprint compared to DFS and assume that the planner will struggle with domains where plans are only found in deep layers or where the branching factor is very high as both will lead to a blowup in the size of our fringe. At the same time, we expect the performance of BFS to be a lot more consistent than for DFS, as the layer at which we find a plan stays fixed for any single instance.
Overall, we do not expect high performance of our BFS. It may however prove useful on some domains and help us understand and validate assumptions about the behavior of TOHTN problems.

3.1.3 Heuristic Search

Both random DFS and BFS are unguided and do not adapt the order of search node exploration to information contained in those nodes. Other planners, such as PANDA, use heuristics to

guide their search. We will describe search heuristics and their use in PANDA according to [22] and then go over how we try and adapt the use of heuristics with the added constraints of malleability.

Heuristics in hierarchical planning in general and PANDA specifically The general idea of heuristic search is to explore our search space more intelligently. Heuristics achieve this by guiding the search to the most promising search nodes first. In TOHTN planning, our choices during search are restricted by both the hierarchy of tasks as well as the world state. As a result, the best heuristics should make use of both pieces of information for the best results.
One avenue to deriving heuristics for HTN planning is to adapt classical planning heuristics. This proves difficult as these heuristics do not know about the hierarchy and may assume a state-based goal which HTN planning often does not have. To avoid these issues, PANDA instead adapts the hierarchical problem to match the heuristics. PANDA computes a classical planning problem which is a relaxation of the HTN problem at hand. Then a solution to this relaxed model is approximated with the help of classical planning heuristics and the result is used to guide the initial HTN planning procedure. The computation of the classical model is possible in polynomial time and only done fully in the beginning, afterwards the model is only updated for the current state of planning. As a result, the heuristic takes into account both hierarchy and world state with little overhead during search.

Problems with the PANDA heuristic for malleable HTN planning While PANDA has managed to make great use of heuristics in HTN planning, we cannot simply adopt the same heuristics for malleable CrowdHTN. The reason for this lies in the assumption of PANDA that a ground problem instance is already available. Grounding is an expensive operation, though, as discussed in [6]. A full grounding may be exponential in size compared to the input and run times of grounding procedures are accordingly high.
While a grounding is already available in PANDA, CrowdHTN does not perform explicit grounding before planning. In a malleable environment without shared memory we can expect this grounding to take place every time a PE is added to a job, adding a high startup cost. A short-lived worker may be interrupted while still grounding, never starting the actual search. This would interfere with the efficient usage of available resources. For this reason we have decided against using the PANDA heuristics in CrowdHTN and instead tried to design a simpler heuristic to be used in malleable TOHTN planning.

A heuristic for malleable HTN planning To counter the startup cost of the PANDA heuristic, we have devised a simpler heuristic which is cheap to precompute and can easy to use in malleable planning. The goals are to have little startup overhead and to retain the efficient evaluation at each search step. As discussed in the previous paragraphs, this limits any precomputation to the lifted instance. We hope to still find performance gains on at least some problem instances.
As heuristic value, we use a lower bound on the number of reductions we still need to perform to fully resolve our list of open tasks. When computing this lower bound we ignore preconditions and effects, searching the shortest possible way through the hierarchy. We precompute this value for each task as described in algorithm 3. The heuristic value for each action is initialized to zero. The heuristic value for each abstract task is initially unknown. In each step we loop over all tasks t. For each method m of task t we check the heuristic value of all subtasks. The heuristic value of a method is set to the sum of the heuristic over all subtasks plus one. For each task we choose the minimum value over all corresponding methods.

Once there are no more changes in the mapping of tasks to heuristic values, we stop. Any task which at this point does not have an assigned value is not resolvable at all and can be pruned. To visualize the computation of our heuristic, we provide an example TOHTN domain in figure 2 and table 1 shows how the heuristic is computed on this domain.
Computing the final heuristic will take at most as many iterations as there are compound tasks. To show this we look at our hierarchical planning problem as graph. The tasks and methods form the vertices. We get edges from abstract tasks to all applicable methods and from methods to all their subtasks. Actions do not have any outgoing edges. During computation, the heuristic values are initialized at the actions and propagated and update throughout the graph. As tasks and methods alternate, any cycle contains at least one method and as such propagating the heuristic through a cycle would strictly increase it.
improve this?
To evaluate our heuristic while planning, we now need to look at the whole sequence of open tasks and calculate the sum of our heuristic over those tasks. While the naive approach gives us linear run time in the number of open tasks, we can stretch the computation over task instantiation and reuse parts of it to perform heuristic evaluation in

$$\mathcal{O}\left(\max\left\{\#\text{subtasks of } m | m \in \mathit{methods}\right\}\right)$$

at run time. Details can be found in 6.5 on efficient hashing of the open tasks, the technique also applies to the heuristic computation. Additionally, while we only use this heuristic to guide TOHTN planning, it ignores any orderings between open tasks. As such, it can be applied to HTN planning as well.

Algorithm 3: GBFS heuristic calculation

```
1 task depths ← {(t_c, 0)|t_c ∈ concrete tasks}
2 while task depths changed do
3     for t_c ∈ compound tasks do
4         reduction depths = ∅
5         for r ∈ reductions for t_c do
6             if depths of all subtasks are known then
7                 reduction depths ∪= 1 + ∑{d|d is depth of a subtask of r}
8         if reduction depths ≠ ∅ then
9             task depths ∪= {(t_c, min(reduction depths))}
```

Using the new heuristic in TOHTN planning With the new heuristic presented in the previous paragraph, we implemented two new search algorithms for CrowdHTN, those being a heuristic DFS and an A-star like search.
The implementation of DFS used so far performs the search in a uniformly random order. Without knowing anything about the domain, this is a reasonable choice. While it is far from optimal search, it also avoids any pathological cases that may arise from a fixed order of exploration. As an alternative to this random order, we used the heuristic to guide our DFS. We have to note that this is not necessarily fully deterministic. Randomness comes into play both when two reductions lead to search nodes with the same heuristic score - which happens e.g. for different instantiations of the same method - and when performing work stealing in a parallel setting.
In addition to the guided DFS, we also implemented an A-star like search where a node's value

Figure 2: Example TOHTN domain to demonstrate our heuristic

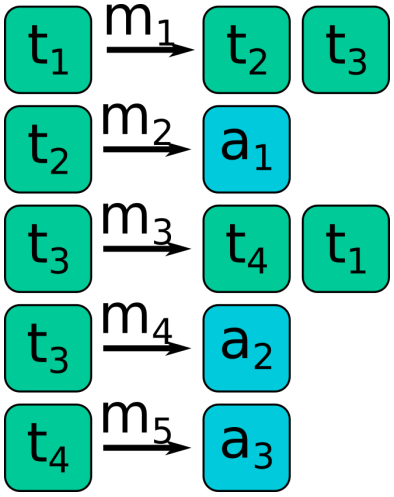


Table 1: Example computation of our TOHTN heuristic for the domain in figure 2. Changing values are bold.

	Iteration			
task	0	1	2	3
a_1	**0**	0	0	0
a_2	**0**	0	0	0
a_3	**0**	0	0	0
t_1			**3**	3
t_2		**1**	1	1
t_3		**1**	1	1
t_4		**1**	1	1

is the sum of the heuristic value and the number reductions applied to reach the node. We differ from A-star in that we terminate the search as soon as a plan has been found instead of continuing on the search for an optimal plan. Our goal is for the heuristic to guide us towards a plan while giving weight to the number of applied reductions forces us to turn back and explore other parts of the search space without getting lost in endless loops due to pathological cases in our heuristic.

3.1.4 Completeness of different Search Algorithms

In the previous paragraphs we discussed a number of different search algorithms that we implemented for CrowdHTN and how we expect them to affect the performance characteristics of our planner. The search algorithm has a more fundamental impact than that, however, and may affect the completeness as well. In this section we want to give a short overview over the completeness of each of the algorithms. A summary is found in table 2. Note that this discussion is only about the algorithms without any modifications. In section 3.2 we discuss both loop detection and a restart mechanism and in section 4 we have a more detailed discussion about the completeness of different planners as well as the completeness of progression search with these additions.
DFS is not complete as it may enter an endless loop and, even if it still explores side-tracks from this loop, will never be able to backtrack out of the loop, cutting of parts of the search space. There is however always a chance to find a plan if it exists. I.e., there is a non-zero chance that the random choices all happen to be done correctly, the loop is never entered and a plan is found.
BFS on the other hand is trivially complete. While the exploration order within each layer is random we can provide an upper bound on the number of search steps required to explore a given search node n on layer i. One such bound is the sum of all layer sizes from 0 up to and including i.
Next in our list is heuristic DFS. Similar to random DFS it may run into an endless loop. Compared to DFS, however, heuristic DFS may do so in a deterministic fashion if the domain triggers a pathological case in the heuristic. One example domain which would trigger such a case in our heuristic is visualized in figure 3. In this instance our heuristic assigns the value 1 to m_1, 2 to m_2 and 3 to m_3. If the preconditions of m_1 are not fulfilled, heuristic DFS will first try to resolve t_1 via m_1, fail, then use m_2 with the goal to try m_1 again afterwards. This will fail again and the loop is repeated indefinitely. While m_3, m_4, m_5 may provide a path out, they will never be used.
Lastly, we implemented A-star like search. While it does reuse the same heuristic, this algorithm achieves completeness by also valuing the number of applied methods so far. Let n be a node with heuristic value h and r previously applied reductions to reach n. Then n is guaranteed to be explored once all nodes with at most $h+r$ previously applied reductions have been explored.
All of this discussion so far has assumed sequential planners. The implications do hold for parallel search, too. For a planner using at most n PEs we may always provide an instance with n ways to enter an infinite recursion.

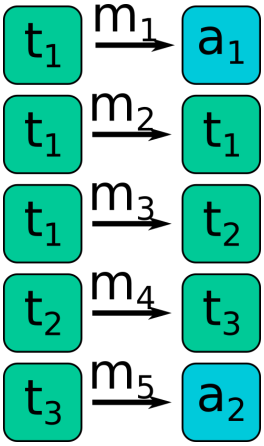
3.2 Loop Detection

In recent years it has become clear that the recursive nature of HTN instances poses its own set of challenges to planners. As a result, mechanisms to perform loop detection have become an active area of research with both HyperTensioN [24] and PANDA [19] exploring it. In this

Table 2: Completeness of the different search algorithms in CrowdHTN

Algorithm	Completeness
Random DFS	Not complete. Positive probability to find a plan if it exists
Random BFS	Complete
Heuristic DFS	Incomplete
A-star like	Complete

Figure 3: A pathological case in our new HTN heuristic



section we will discuss loop detection specifically in the context of parallel and distributed hierarchical planning. We start out with a discussion of loop detection techniques in other planners such as PANDA and HyperTensioN in section 3.2.1. This is followed by a short overview of how CrowdHTN specifically differs and how this changes our base assumptions in section 3.2.2. Afterwards we first explore loop detection based on hash sets in section 3.2.3. We conclude by exploring how approximate-membership-query (AMQ) data structures can be used in loop detection, how this affects completeness of the progression search algorithm and present our design for a distributed and global loop detection mechanism in section 3.2.4.

3.2.1 Loop Detection in Other HTN Planners

Loop detection in HTN planning is a recent phenomenon and was introduced in 2020 by the HyperTensioN planner with the so-called 'Dejavu' technique [24]. Dejavu works by extending the planning problem, introducing primitive tasks and predicates that track and identify when a particular recursive compound task is decomposed. These new primitive tasks are invisible to the user. Information about recursive tasks is stored externally to the search as to not loose it during backtracking. Dejavu comes with performance advantages and protects against infinite loops. However, as Dejavu only concerns itself with information about the task network but ignores the world state it may have false positives. This was also noted by [19] and means that HyperTensioN is not complete. [19] further nodes that the loop detection is limited in that it only finds loops in a single search path but cannot detect if multiple paths lead to equivalent states.
In response to HyperTensioN, the PANDA planner introduced its own loop detection in [19]. Similar to HyperTensioN, PANDA keeps the loop detection information in a separate list of visited states, $\mathcal{V}$. Search nodes (s, tn) of world state s and task network tn, are only added to the fringe if they are not contained in $\mathcal{V}$. To reduce the number of comparisons required to determine whether $(tn, s) \in \mathcal{V}$, $\mathcal{V}$ is separated into buckets according to a hash of s. In

the sub-case of TOHTN planning, both an exact comparison of the sequence of open tasks as well as an order-independent hash of the open tasks called *taskhash* are used. Similar to HyperTensioN, using a hash to identify equal task networks can lead to false positives and an incomplete planner. The loop detection in PANDA improves upon the one in HyperTensioN insofar as it is not just able to detect loops but also recognizes when equivalent search nodes are reached on independent paths.

3.2.2 Assumptions in Loop Detection for CrowdHTN

To design the loop detection in CrowdHTN, we have both simplifying and complicating assumptions that we will discuss here.
While both PANDA and HyperTensioN are HTN planners, CrowdHTN concerns itself only with TOHTN planning. As a result, the remaining task network can be represented as a sequence of open tasks with the ordering constraints implicit in how the sequence is stored.
As tasks of equivalent task networks are always in the same order, we can incorporate that order into our hash of tn to reduce the number of collisions compared to PANDA's *taskhash*. This will increase performance where we fall back to comparisons in case of collisions and reduce our false positive rate in case we forgo comparisons for performance reasons.
Both PANDA and HyperTensioN are sequential planners whereas CrowdHTN is highly parallel. This adds an additional design constraint to our loop detection. If we want to efficiently share information about visited states, directly sharing search nodes would be infeasible due to their size. If we perform loop detection only locally, we expect to suffer from decreased performance as the degree of parallelism increases. I.e., if a search node exists multiple times we may encounter it on different PEs, not realizing that it is a duplicate.

3.2.3 Hash Set Based Loop Detection

One simple way to perform loop detection which is also used in PANDA ([19]) is to use a hash set of visited states. The implementation in Crowd is slightly different from PANDA in that we use one combined hash for both world state and open tasks. Other than PANDA, CrowdHTN does incorporate the order of tasks into the hash, which should reduce collisions and makes the two levels of hashing less needed.
Using hashes combined with a full comparison provides us a perfect loop detection, i.e., neither false positives nor false negatives exist. This makes it a useful technique to benchmark other loop detection methods. However, both in the sequential and in the distributed case this technique suffers from performance problems.
In case of hash collisions, we have to fall back to a full comparison of world state s and open tasks tn. While s is bound in size by the total number of predicates, the size of tn is effectively unbound, making this an expensive operation. Additionally, we have to keep both s and tn around for all nodes ever encountered, increasing the memory footprint of our planner.
The hash function we do use is a combination of the hashes for the task network and the world state. The sequence of open tasks can be hashed as-is in a deterministic fashion by iterating over it from beginning to end. For the world state as a set of predicates we do not have a fixed order. We solve this by combining hash values of predicates by adding their squares, a commutative operation. Other options would have included first sorting the elements to impose a fixed order but would have negatively impacted hashing performance.

3.2.4 Approximate Loop Detection

In the preceding sections we have always made the assumption that our loop detection mechanism needs to be perfect, i.e., it needs to avoid both false positives and false negatives. Some hierarchical planners do not share this assumption and both HyperTensioN and PANDA have configurations that allow for false positives. In the following paragraphs, we will also permit false positives to occur and explore the implications.
We start out by introducing the concept of AMQ data structures with a specific focus on bloom filters and how to use the scalable bloom filter in loop detection. Once this is done, we turn to the problem of false positives and introduce a restart mechanism that guarantees that, given enough time, we are able to reach any search node. The section is concluded by us using the special properties of bloom filters to design a distributed loop detection mechanism which allows for efficient information sharing between PEs.

Approximate membership queries AMQ data structures are used as a memory efficient representation of sets that allow for a false positive rate during membership queries to be able to gain memory efficiency [7]. They were introduced with the bloom filter in 1970 [8]. Since then both variations of bloom filters, such as counting bloom filters [16], and other AMQ data structures have been introduced, among them quotient filters [7] and cuckoo filters [15].
As the guarantees of the bloom filter are sufficient for us, we will now focus on this specific data structure using the definition of [8]. A bloom filter is defined by three numbers, the number of bits in the filter m, initially all set to zero, the number of hash functions k, each producing hashes in the range $0, \dots, m-1$, and the number of elements already present in the filter n. To insert a new element, we use the hash functions to compute k hashes and use them as indices into our bit vector, setting the corresponding bits to 1. Similarly, to query for membership we check whether the corresponding k locations all contain a 1. This leads to highly efficient insertion and membership queries in time $\mathcal{O}(k \cdot h)$ where h is the time required to hash an element.
One limitation of bloom filters is the fact that they do not support element deletion. We cannot simply set a bit to zero, as there may be more than one element requiring it to be 1. To deal with this limitation the concept of a counting bloom filter was proposed [16]. Instead of a single bit per element, multiple bits are used per index to store a counter tracking the number of elements belonging to the index.
Given m, n and k we can compute the probability of encountering a false positive. A detailed discussion of this can be found in [10]. The main result is that the probability for any bit to contain a 1 is

$$p' = 1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}}$$

This gives us an overall probability of false positives of

$$p = p'^k = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

We see that the probability of encountering false positives steadily rises as the number of elements in the filter grows, reaching 1 as all bits are set to 1. As a result, bloom filters in their original form are best suited for static sets of known size and may necessitate setting m conservatively high to guarantee a low rate of false positives. To fix this and guarantee an upper bound on p even in dynamic sets, the concept of scalable bloom filters was introduced [35]. A scalable bloom filter builds a hierarchy of bloom filters, each new bloom filter being twice the size of the previously largest one. To insert an element into a scalable bloom filter, we first check the false positive probability of its largest sub-filter. If it is still under our decided bound,

we simply insert the element into the largest filter. If inserting the new element would raise the false positive rate beyond the limit, we add a new filter, twice as large as the previously largest, and insert the element there. Membership queries now have to check all levels of the filter. While there is a performance overhead, only a linear number levels is required to store an exponential number of elements.
Given bloom filters and their variations we have decided on the use of a scalable bloom filter for loop detection in CrowdHTN. While our set of search nodes is theoretically limited in size, in practice the search space is prohibitively large and unlikely to be fully explored. Using a single bloom filter of fixed size would lead to a high size requirement to ensure low false positive rates. At the same time, once inserted we want to forever keep an element in our set of known nodes and do not require deletion, allowing us to forgo mechanisms like the counting bloom filter. To get k hash functions, we reuse the hash function we already use in the hash set based loop detection, varying the seed to generate different hashes.

Completeness in the face of false positives While using approximate membership queries as described in the previous paragraph gives us a number of advantages, it also introduces a new set of challenges. Among these is are false positives. As a result, we can expect to loose parts of our search space. More specifically, if we perform progression search using bloom filters and $n = n_1, \ldots, n_l$ is a path of length l from our initial search node to a goal search node, then for any n_i in n, the search nodes $n_1, \ldots, n_{i-1}$ may collectively set the k hashes associated with n_i, filtering it out. This may end up turning a TOHTN problem unsolvable for us even though a plan exists. As a result, our planner, if otherwise unchanged, will no longer be complete.
We will now take a closer look at the probabilities involved. Assuming we use a scalable bloom filter with maximum false positive rate $p < 1$ and a hash function which maps search nodes to uniformly independent values and n is a shortest path. Then

$$q = 1 - (1 - p)^l < 1$$

is an upper bound on the probability that we are unable to solve the problem even though a solution exists.
If hashes between runs are independent of each other we get

$$\lim_{t \to \infty} q^t \to 0$$

That is, if keep re-running our search with new, independent hash functions we regain completeness. It is critical to use independent hash functions between runs to ensure that false positives in different runs are not correlated with each other. In the next step we need to determine when to re-run our planner. There are three main constraints.

(i) As the TOHTN instance may be recursive, the search space may be infinite
(ii) The number of restarts needs to be unbounded as run time goes to infinity
(iii) As a plan may be arbitrarily long, the number of runs with run time at least u needs to be infinite

Constraint one implies that we cannot simply wait until we explore the whole search space before restarting. Instead we will base our restarts on total run time so far. To fulfill constraints two and three, we perform a check every second where after t seconds we perform a restart with probability $\frac{1}{t}$.
For the expected number of restarts as we get $\sum_{t=1}^{\infty} \frac{1}{t}$. This is the harmonic series and diverges, giving us the required unbounded number of restarts. As t grows, the probability of restarting decreases, allowing for increasingly long runs, fulfilling the third constraint.

This mechanism allows us to restore completeness to our planner while utilizing AMQs. Restarts may prove to have additional benefits to planner performance, as DFS-based planners tend to be hit-or-miss where restarts increase the number of opportunities for a hit. We do note that approximate loop detection does come at the cost of no longer being able to detect UNPLAN, as we can never guarantee that we explored the full search space. In practice, we do not expect this to matter as the search space of (TO)HTN problems tends to be too big to feasibly fully explore.

Global Loop Detection In the section 3.2.2 we already mentioned that loop detection in distributed hierarchical planning comes with unique problems. Specifically, current loop detection techniques do not include ways to efficiently share the visited states between PEs. As a result, a search node is only fully filtered out once each PE has encountered it at least once. This is less of a problem for recursive tasks which are quickly re-encountered by our local search and subsequently filtered out. It does however loose us the ability to discover when different paths lead to equivalent nodes in which case we want to avoid duplicate work. To address this issue, we will start with a short discussion on how previous loop detection mechanisms are hard to adapt for the distributed case and then show how we implement distributed loop detection on the basis of bloom filters.
As mentioned in section 3.2.3 on hash set based loop detection, it suffers from a high memory footprint as we keep whole search nodes around. This problem extends to the distributed case, as we would now have to communicate whole search nodes leading to a large overhead for encoding, sending and decoding. Even if we assume the communication overhead to be low enough, we run into additional problems. Inserting n elements into a hashset takes $\mathcal{O}(n)$ time even without hash collisions. The higher the number of PEs, the more time would be spent on inserting search nodes received from other PEs which would either block us from performing search for large amounts of time or introduce synchronization problems. As a result, we have decided that it is infeasible to extend hash set based loop detection to the distributed case.
Compared to hash sets, bloom filters offer a number of advantages for distributed loop detection. First, bloom filters offer a more compact representation of the already encountered nodes which leads to a lower communication overhead. Second, as the filter itself is stored as a simple bit vector, we have negligible overhead regarding encoding and decoding for communication. Thirdly, we can efficiently merge two bloom filters by performing a simple bitwise or operation of the bit vectors. In a combined filter, we get a conservative upper bound for the total number of contained elements by summing the number of elements of both filters. This guarantees that our maximum rate of false positives is not exceeded.
To integrate bloom filters into our distributed loop detection, we also need to address the question of what to do in case the global filter gets filled up, i.e. it's false positive rate reaches our set limit. For local loop detection we introduced scalable bloom filters. This is a problem as we now communicate whole sets of search nodes whereas locally we introduce new search nodes into the filter one by one, increasing the size at exactly the right moment. As a result, we face the choice of increasing the size early, throwing away some information or loosing our guarantees regarding false positives. Similarly, we face the problem where different PEs may disagree about the current maximum size of the scalable bloom filter, putting more information in a smaller filter that will be thrown away by other PEs.
To deal with this problem, we induce a restart in our search once the global filter is full. As the restart mechanism is already present due to the need to deal with false positives this is an easy adaption. To limit the number of needed restarts and once again allow arbitrarily long runs, we double the size of our global filter with each restart. Additionally, we limit the amount of information present in our filter to further reduce the number of restarts we need. We do

this by only putting 'important' search nodes into our filter. Our heuristic to determine search node importance is to put a node into our global filter if it is present in our local filter and encountered again. Other heuristics are possible but beyond the scope of this thesis.

4 Completeness of HTN Planning Algorithms

In section 3.1 on different search algorithms we already did a short discussion on the completeness of different algorithms. The current section will start with a short recap of our findings, expand them to other planners and will then do an expanded discussion that takes factors like loop detection and restarts into account.
Before we dive into the more detailed discussion we want to note that we have seen in section 2.2.2 that there is an upper bound to task network depth where, if a plan exists at all, it can be found before that depth. By limiting our planning to task network expansions with lower depth, we can trivially achieve completeness. This is however of little practical use as this depth bound is exponential in the problem size. As a result, we can expect to run out of memory before hitting this bound. For this reason we do not make use of this bound and as far as we know no other planner does. We will now resume a more practical discussion of planner completeness.
As previously noted, we can split our search algorithms into three main groups:

- Algorithms that are complete (BFS, A-star like search)
- Algorithms with a chance but no guarantee to find a plan (DFS)
- Algorithms which for some domains will never find a plan (heuristic DFS with pathological cases)

Completeness in other planners So far we have only classified the different search algorithms present in CrowdHTN. For now we will take a look at other planners, starting with translation-based planners totSAT ([4]), Tree-REX ([31]) and Lilotane ([30]). As we have noted in the discussion on planning algorithms in section 2.4, all three of these planners are based on SAT. Additionally, they all explore the set of potential expansions of the task hierarchy in a layer-by-layer fashion, leading to a BFS-like characteristic in their behavior. As a result, these planners are complete.
In contrast to this, we have the space of search-based planners, starting with HyperTensioN [24]. For HyperTensioN, the authors themselves note that their inbuilt loop detection mechanism suffers from false positives with no mechanism to mitigate them [24]. It follows that their planner is not complete. If we disabled the loop detection in HyperTensioN we would be left with a planner performing DFS, which would put it in the category of planners which are not complete but still have a chance to solve any instance.
PANDA on the other hand is a planner based on heuristic progression search that offers a number of configuration options for both search and loop detection. Regarding search, PANDA offers both a pure heuristic DFS and a weighted A* search taking into account the previous path [22]. For loop detection PANDA offers hashing based mechanisms both with and without a fallback to full search node comparison [19]. Completeness of the planner varies depending on the chosen configuration. If loop detection is configured to allow for false positives, we expect PANDA to not be complete regardless of search algorithm, as there is no mechanism in place to mitigate their effect. If a loop detection mechanism is chosen which does not have false positives, we expect PANDA to be complete if weighted A* with a weight $w > 0$ is used, as this introduces a BFS-like behavior into the search. This leaves pure heuristic DFS combined with loop detection without false positives. Due to the complex nature of the PANDA heuristic we were unable to construct any pathological case which leads PANDA into an infinite recursion. As heuristics are inherently limited we do assume that such cases exist. We will explore this case and the similar case in CrowdHTN in the following paragraph.

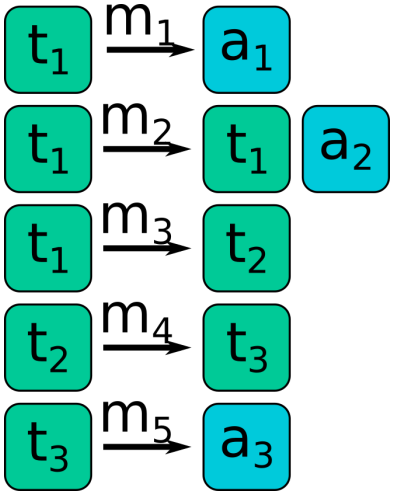
Loop detection and completeness As we have seen, heuristic search on its own may increase planner performance but comes at the cost of completeness. Random DFS is able to find any plan but may still get stuck in endless loops. We will now explore the implications of combining heuristic search with loop detection but without restarts to see how this changes the overall situation. In this paragraph we are only interested in loop detection mechanisms that do not suffer from false positives as, without restarts, this automatically disqualifies a planner from being complete.
In general, loops are only a problem in hierarchical planning if there exists at least one recursive task. If no such task exists, there exist only a finite and usually small number of possible task network expansions such that we can easily search the full search space. If we do have a recursive task, we can further classify our instances according to how hard it is to deal with. We identify three categories:

- Tasks which recurse into themselves with no change in the world state
- Tasks which recurse into themselves with changes in the world state
- Tasks which recurse into themselves while adding more tasks afterwards

For general HTN planning, a task recursing into itself also implies that the ordering constraints of the new open tasks are a superset of the old ordering constraints.
The first case is the easiest to detect and fix. If a task recurses into only itself we do not get any changes to the open tasks. As a result, search nodes before and after this recursion are equivalent. They will be detected by loop detection as it is used in PANDA and CrowdHTN and the search will be guided into another direction.
In the second case we have to perform additional work before a loop can be detected. As the set of open tasks stays the same and the world state changes, we do not immediately get equivalent search nodes. However, in an HTN instance with predicates $\mathcal{Q}$, there are only $2^{|\mathcal{Q}|}$ possible world states. We can easily see that we will recurse at most $2^{|\mathcal{Q}|}$ times before our loop detection activates and we backtrack. In practice we can often obtain a smaller upper bound on the possible number of recursions by looking only at the predicates which occur as effects in the resolution of any tasks present in the recursion. We see that, while less efficient, our known loop detection mechanisms correctly deal with this case.
This leaves us with the third case, where a task does not directly recurse into itself but where the resolution of task t gives rise to a new instance of t as well as additional tasks $t_1, \ldots, t_k$ which are restricted to be resolved after t. As a result, once we re-encounter t our open task set has changed. While we were able to limit the number of possible world states in previous case, this is not possible here, as the number of open tasks is unbounded. More specifically, loop detection as used in PANDA and CrowdHTN is unable to handle this case.
Figure 4 provides an example of an instance which would guide our proposed heuristic into a recursion while not creating loops. Our heuristic would assign the values 1 to m_1, 2 to m_2 and 3 to m_3. If the preconditions for m_1 are not fulfilled we would then apply m_2, creating a unique set of open tasks and then repeat application of m_2 indefinitely.

Restarts and completeness Together with AMQ based loop detection, we introduced a restart mechanism in 4. We will now go over the use of restarts to achieve completeness for random DFS.
To show that restarts help us to achieve completeness for random DFS we can use a similar argument as we did for the loop detection. For any path in our search graph, random DFS gives us a probability $p > 0$ to take this path. As the number of restarts we perform goes to infinity, the probability to take any fixed path at least once goes to 1. We are under the same constraints as previously, needing both an unbounded number of restarts and an unbounded

Figure 4: Pathological instance for our proposed heuristic that is not caught by loop detection



number of runs of at least length u for any u. The second constraint is needed so that we do have the time to fully explore a path once we take it. Our restart mechanism fulfills both constraints, restarting with probability $\frac{1}{t}$ at second t. It follows that random DFS with restarts is complete.

Conclusion In this section we have taken a look at completeness in hierarchical planners and how loop detection and restarts can help us to achieve it. We have shown that completeness is highly dependent on the specific search behavior with BFS-like behavior being trivially complete. In addition we show that loop detection, while helpful, is not able to solve the problem for some instances. Introducing our restart mechanism, we show that it can turn random DFS into a complete algorithm. This does not extend to heuristic DFS if the heuristic exhibits pathological behavior as the heuristic will always guide the search back into the same recursion.

5 A Malleable TOHTN Planner

The goal of this section is to describe how we adapt it to be a malleable TOHTN planner by integrating it with Mallob, preserving both the completeness and scalability of CrowdHTN in the process. Before we get into the details, let's recall that CrowdHTN is already a moldable program according to the definition introduced in section 2.5, i.e., it may utilize any number of PEs as long as that number stays fixed during the run. We will now introduce a design that extends the parallel capabilities to achieve malleability. For this we need to address three main concerns.

- Distributing the job information
- Integrating new PEs into a running job
- Dealing with PEs leaving the job while it runs

In the following sections we will address these problems in this order. Both distributing the job information and integrating new workers do not pose significant problems. Most time will be spent on the handling of disappearing PEs. Due to the fact that we specifically integrate CrowdHTN with Mallob, in some parts we will have to refer to implementation details regarding how Mallob organizes the PEs assigned to a job as well as general message delivery.

5.1 Distributing Jobs

When a PE is assigned to a job, it needs to obtain a description of this job. In case of TOHTN planning, the choice is mostly between a lifted or ground TOHTN instance. Depending on pruning, a ground instance may be up to exponential in size [6]. Encoding and communicating such a ground instance would take up much time, which is why we decided to communicate our problem as a lifted instance.
With the lifted instance, we choose to simply take the textual hddl input ([21]) and send it as-is. While this does occur the overhead of locally parsing the instance on each PE, communicating the parsed instance would involve re-encoding and effectively re-parsing it locally, too.
In malleable (TO)HTN planning there is a more general trade-off involved when it comes to precomputation. While parsing the instance is unavoidable, we can choose whether we want to spend time grounding and pruning our instance. It has been shown that grounding and pruning improve the planning performance ([6]) and allow for the computation of complex and good heuristics ([22]), but grounding, pruning and other precomputations are expensive operations themselves. As a result, a PE which is only assigned to our job for a short time may never perform any actual planning work before it is reassigned to the next job. For this reason, CrowdHTN takes an alternative path. The TOHTN instance is kept in lifted form. Instantiation is only performed as needed to explore the current search node. This allows CrowdHTN to start working immediately to utilize even short-lived PEs.

5.2 Integrating New PEs Into Malleable CrowdHTN

To integrate a new PE into a running TOHTN job, it needs both the general job description and part of the actual work to handle. In the previous section we explained how the job description is obtained, now we will focus on the work itself.
The efficient integration of new PEs into a running job is where work stealing shows it's strength. For work stealing, there is no functional difference between a PE which has locally run out of work and a new PE which has the job description but no work yet. Both will message other PEs at random to receive a new work package with no special handling required. As a result, a

new PE can perform at full efficiency almost immediately, allowing our job to utilize resources as they become available.

5.3 Handling PEs Leaving at Run Time

The last challenge in designing a malleable CrowdHTN is the fact that PEs may disappear at any time. This represents a potential loss of information. The information loss presents itself in two ways. First, the loss of the local search fringe, if we do not communicate it to another PE and second, messages which may be lost in transit as their receiver no longer belongs to the same job. To deal with this, Mallob does allow us to detect locally when a PE is taken away from a job and additionally provides a message return mechanism. We will present our solutions to both cases with a focus on preserving the completeness property of CrowdHTN.

5.3.1 Handling the Local Fringe

When a local PE is unassigned from a job, we will loose the local search fringe. As Mallob signals a PE when it is unassigned from a job, we are however free to encode parts or all of the fringe and communicate them to another PE. This leaves us with a number of choices where we may trade-off data loss versus efficiency and communication. On this axis we discuss three choices

- Encode and redistribute the whole local fringe
- Communicate the root of the local search space
- Communicate nothing, loose the local fringe

Encoding and redistributing the whole fringe Encoding and sending off the local fringe to another PE is, in a way, the easiest operation. No information is lost, preserving completeness in our planner. It does, however, come with a number of disadvantages. First, the local fringe may be arbitrarily large, especially considering that TOHTN planning is EXPSPACE-hard as seen in section 2.2.2. Encoding and communicating a large fringe is a very expensive operation which would increase the time from Mallob telling a PE to suspend itself until the PE actually is free for the next job. Second, receiving a large fringe would strain the memory of the receiving PE which may lead to deleting parts of it anyways to avoid crashes. Third, to avoid duplication of work as Mallob may reassign the PE to the old job, the local fringe would have to be cleared out. Doing so would weaken the effect of Mallob reassigning previously used PEs to the same job.

Communicating the root of the local search space Instead of communicating the whole local fringe, we can simply encode the root node the local fringe emerged from. In a way, this search node represents a very efficient encoding of the local search space. As we would only communicate a single search node, this would be more efficient and could reuse the facilities we already have in place for work stealing. Similar to encoding the whole fringe, communicating only the root search node would lead to no information loss, preserving completeness.
While this approach is very efficient and avoids loss of information, it does suffer from duplicate work. As we loose the local fringe, we will have to re-explore it again. Additionally, other nodes may have received parts of the local search space via work stealing. These nodes will be re-encountered leading to further duplication. In this way, we would trade-off local performance for encoding and communication against global performance through duplicating parts of our search. Similar to the first case, we would either have to clear out the local fringe upon

suspension, decreasing the gains of PE reuse, or accept potential further duplication of work. Implementing global loop detection as we propose in section 3.2 would further complicate matters. Upon suspension of a PE, we could leave the global loop detection unaffected. This might lead to some losses in our search space as nodes will not be reexplored but may also help the search on our other workers which can still profit from being aware of common loops and prominent search nodes. Alternatively, we could remove the global loop detection data of our suspended PE from all other PEs. As more than one PE may have committed the same search node to global loop detection and due to the way bloom filters work, this brings its own problems. Namely, it would degrade global loop detection performance as we may delete more search nodes from our filter than strictly necessary.

Communicate nothing Our third option in dealing with disappearing workers is to accept the loss of information and communicate nothing to the remaining PEs. This comes at the cost of loosing information while being easy to implement and allowing for immediate reassignment of PEs and avoiding any duplication of work. Additionally, we do not have to clear out the local fringe to avoid duplication, allowing for efficient re-assignment of PEs to their old job.
This leaves us with the problem of information loss. To deal with this, we can revisit the restart mechanism we introduced to deal with a similar problem in probabilistic loop detection. There we argued that correctly designed restarts would allow our planner to be complete even when using a loop detection mechanism suffering from false positives. For this we made no assumptions besides the false positive rater being less than 1. As Mallob guarantees that we will always have at least one PE, never loosing all information, we can simply model the loss of PEs and their local fringes as an extremely high false positive rate. From this it follows that restarts allow us to loose this local information while maintaining overall completeness.

Conclusion As we have seen, there are multiple approaches on how to handle a reduction in the number of available PEs while maintaining planner completeness. In our design, we decided to communicate the root node of our local search space to a random other PE, inserting it at the back end of the fringe. The other PE is chosen at random to avoid turning any single PE into a bottleneck. We choose this design, as it allows us to avoid the large overhead of communicating the whole fringe while not being overly reliant on restarts to achieve completeness. While restarts do offer us completeness from a theoretical perspective, times between restarts increase rapidly as run time increases. As such we are unwilling to loose large parts of our search space and instead allow for the risk of performing duplicate work. In addition to this, we keep the global loop detection information unchanged, hoping that even after a PE is no longer assigned to a job, other PEs will profit from the information it provides.

5.3.2 Handling Lost Messages

In the moldable version of CrowdHTN, we could made a fundamental assumption about all messages, namely that they were guaranteed to be delivered. In malleable CrowdHTN this is no longer possible. If PEs p_1, p_2 are both assigned to the same job, then p_1 may send a message to p_2 with p_2 being reassigned to a different job before the message arrives. Mallob deals with this by recognizing the message can no longer be handled and returning it to the sender. We go over the way we handle such return messages in 9 our implementation chapter.
However, the changing assignments of PEs to jobs imply an additional problem. The return message may be lost as well, if the original sender gets assigned to a different job before the return message can be received. Mallob provides no further handling mechanism for this case. One way to solve this problem would be to extend Mallob to forward such a message to the

job's root PE. In our design we instead chose to not handle this case for three reasons.
First, this case is highly contrived and we expect it to be of very little practical significance. It relies on a very specific sequence of events and could only present a problem if the message in question contained a work package. Second, by not handling this any further we simplify our design and implementation as we avoid special-casing the root PE. Third, as we choose to handle unassigned PEs by preserving the root of their local search space, information is preserved even if any of it's transitive children is lost in the moment. Due to this, the lost message does not represent a lost part of our search space.
We can further construct a case where a PE receives a work package and immediately passes it on due to a work request, subsequently loosing the search node. In this case we can still fall back to our restart mechanism to preserve completeness, as we expect this to be extremely rare.

6 Implementation

6.1 Mallob Integration

In this section we will give an overview over how we integrated CrowdHTN with Mallob. More information about how to do this for general problems can be found in the Mallob GitHub repository [1]. There are three steps we need to perform:

- Implement a way to read and encode a TOHTN instance
- Implement the Mallob job interface seen at 4
- Implement a way to encode a result for writing to file

As we discussed in section 5 on how we designed malleable CrowdHTN, we choose to communicate an instance by simply transferring the string contents of the instance file, making the first step easy. The third step, encoding a result for writing, is similarly easy as CrowdHTN already contained a mechanism to write a plan to the terminal. Most of the work was done in the second step which we will now discuss in more detail. As a general principle, CrowdHTN was kept as a separate library which is linked into Mallob. The implementation of the TOHTN job within Mallob is a wrapper around this library. This allowed for a clearer separation of concerns where our job implementation does not need to know anything about the specifics of TOHTN planning while CrowdHTN is agnostic of implementation details its environment, e.g. how messages are transmitted.
Both CrowdHTN and Mallob are implemented using the C++ programming language.

Algorithm 4: The Mallob job interface

```
1 void appl_start()
2 void appl_suspend()
3 void appl_resume()
4 void appl_terminate()
5 void appl_solved()
6 JobResult appl_getResult()
7 void appl_communicate()
8 void appl_communicate(source, mpi_tag, message)
9 void appl_memoryPanic()
```

Implementing the Job Interface In this paragraph we explain how we implemented the Mallob job interface for CrowdHTN while upholding the guarantees demanded by Mallob. For ease of reading we will leave out the *appl* prefix shared by all functions.
The Mallob job interface can be split into four parts. First, a worker in Mallob is implemented as a state machine with *start*, *suspend*, *resume* and *terminate* responsible for the transitions. The corresponding state diagram can be seen in figure **??**. Second, the two *communicate* calls allow for communication. For general communication we note that Mallob requires all communication calls to take place in the main thread, i.e. we may not send any messages in any threads we started to perform internal work. For ease of separation we further restrict ourselves to only send messages in the *communicate* calls. Third, we have the functions *solved* and *getResult* for general bookkeeping regarding solutions. Last, we have *memoryPanic* which signals the job that memory usage is critically high. We discuss it's use in the next paragraph.

[1] https://github.com/domschrei/mallob

As [32] writes, Mallob aims to achieve millisecond latencies. To enable this goal, we may not block the main thread any longer than a few milliseconds at most and must keep work performed directly in any of the job interface functions to a minimum. We achieve this by delegating all planning and handling of the CrowdHTN library to a separate work thread which is initialized in the *start* function. This work thread is the only thread ever directly interacting with CrowdHTN. Due to this, we avoid locking on CrowdHTN which also allows us to keep working at all times. State transitions are communicated to the work thread via a number of atomic variables, *suspend* and *resume* additionally use a condition variable to suspend and wake up the work thread.
To be able to keep communication to the *communicate* functions, the job and the work thread exchange messages via separate buffers. While these buffers do necessitate locking, we restrict the critical section to be at most a copy of a few bytes.
The last problem is the *start* call during which we need to parse our TOHTN instance, setup the CrowdHTN data structures and start our worker thread. Here both the parsing of the instance and, within CrowdHTN, computing the heuristic values may take longer than Mallob allows. For this reason, we have decided to place the initialization itself into a separate thread and return fast.

Added Fault Tolerance As a scheduler, within a single execution Mallob may work on any number of jobs making it necessary that jobs do not crash. This imposes additional challenges for TOHTN planning, as we have seen in section 2.2.2 that TOHTN planning is EXPSPACE-hard, meaning we may often run out of memory and will be subsequently shut down by the operating system. Luckily, the Mallob job interface we see in algorithm 4 does provide a function for this case. Mallob does periodically check available memory and if it threatens to run out triggers the *appl_memoryPanic()* function. In our case we have implemented it as clearing out both our local fringe of search nodes and the loop detection information, resetting the local search. Afterwards, an affected PE will simply resume the work stealing and message other PEs to re-join the work at reduced memory footprint. While this does means we loose parts of the search space, the alternative would be to immediately return without a plan. Additionally, with the restarting mechanism we introduced in section 3.2.4 CrowdHTN retains completeness even in this case.

Messages and dying workers In 5.3.2 we explained why we can no longer assume that messages will always be delivered and how Mallob implements a return messages mechanism to catch this. Now we will cover in detail how we respond to each kind of return message. Afterwards we will look at more complicated cases of workers dying and reappearing and show how they do not affect correctness of our planner
In malleable CrowdHTN, workers directly exchange three kinds of messages. These are *work requests*, *work packages*, i.e. a positive answer to a *work request*, and *negative answers*, i.e. the worker receiving the *work request* does not have any work to share. Getting a returned *work request* is equivalent to receiving a *negative answer*. We treat it the same way and the worker sends out another *work request* to a random other worker. When we receive a returned *work package*, we re-add it to the back of our local fringe. This ensures that no information is lost. If multiple *work packages* are returned to a worker, their order at the back of the fringe may change. We do not expect any significant impact from this, as the number of *work packages* which are in flight is limited by the number of PEs overall which is extremely low compared to the number of search nodes in a hierarchical planning problem. This leaves a returned *negative answer* as the last type to handle. We can simply ignore this type of return message.
Return messages are only one case of dying workers affecting our communication. In addition

to this, a worker may die, be terminated and the PE is then, through rebalancing, reassigned to the same job. Any messages meant for the original worker will be received by the new worker, as it belongs to the same job. For *work requests* this is not a problem, the new worker will respond like any other. Receiving *work packages* and *negative answers* will change our behavior. If the worker sent out a *work request* before receiving the *work package* meant for the previous worker, it may end up receiving two *work packages*. Similarly, receiving a *negative answer* while the worker has a *work request* on the way prompts the worker to send out an additional *work request*. In both cases, we integrate any additional *work packages* into the local fringe to not cut off parts of the search space and keep working.

6.2 Efficiently Handling Version Increases

In our malleable CrowdHTN implementation, version increases show up in a number of ways. They are necessitated by the global loop detection introduced in section 3.2 and further allow our DFS based planner to achieve completeness as explained in section 4. Due to the distributed fashion in which CrowdHTN operates, version updates are not perfectly synchronized and workers may be at different versions. We will now outline how correctness is ensured and how versions are propagated efficiently.
While versions between workers may differ, we must ensure that especially work packages of different versions are not mixed as to not duplicate parts of the search space. This is ensured by attaching the worker version to any outgoing messages. Upon receiving a message, a worker first decodes the version. If this incoming version is higher than the internal version, the internal version is updated, the local fringe and loop detection cleared and the message is then handled according to this new internal state. If the incoming version is lower, depending on message type it is ignored (e.g. for work packages) or responded to normally (e.g. for work requests). Including the version with all messages has an additional use when integrating new PEs. As they start out empty and without a way to know the current version, they will immediately send out a work request to a random other worker and receive both the current version and potentially their first work package, requiring no special handling.
Including the version in each message is already sufficient to propagate the version to all workers. However, if we disable global loop detection there are no regular broadcasts from the root to the other PEs. Additionally, the work represented by a search node and it's children may be arbitrarily large. While this reduces the amount of messages sent and is one of the strengths of work stealing in parallel TOHTN planning, it also results in a potentially slow propagation of version increases, having many workers perform outdated work. To counter this problem, whenever a version increase happens at the root PE we start a version broadcast along the binary tree structure of PEs. This ensures that all PEs adopt the new version in a timely manner.

6.3 Global Loop Detection

In section 3.2.4 we introduced a distributed loop detection mechanism based on regularly shared bloom filters. This leaves us with two problems, first performing the associated allreduction while the PEs assigned to the job may change at any time and secondly performing the restarts which are required if the bloom filter fills up.
In both cases we will make use of the specific way in which Mallob organizes the PEs assigned to a job which we have already explained in section 2.7. The two properties we rely on are the fact that PEs are internally structured as a binary tree with parent and child information

available to us and the fact that the root PE will remain assigned to a job during the job's full duration.

Performing the Reduction The all-reduction of our loop detection data is initiated by the root PE and performed in three phases.

- Initiating the reduction
- Aggregating information upwards
- Broadcasting the aggregated information

The root PE is responsible for initializing the all-reduction. It does so by starting a broadcast, sending an initialization message to all it's children. Upon receiving a reduction initialization message, a PE both forwards the message to its own children and prepares the local loop detection data. At the leaves, this data can immediately be sent upwards whereas inner nodes wait until they have received data from all children before performing their local aggregation and forwarding the result upwards. As we combine the bloom filters via a bitwise or operation the message size stays constant throughout. Once the root has received data from all children it once again starts a broadcast, this time containing the aggregated data.
Starting the reduction with the initial broadcast allows us to easily coordinate all PEs even as PEs may assigned to our job may change at any moment. Similarly, we have to deal with PEs leaving at any time. This may be communicated to us either through getting our initial broadcast returned as no receiver is available or by having the *appl_suspend()* function called on us by Mallob. In both cases we simply substitute the message of the missing PE with a response that simulates empty data. We note that, due to changing PEs, the sets of PEs which broadcast their data and which receive the aggregated data may be different. Furthermore, neither of these two sets needs to correspond to the actual tree of PEs assigned to the job at any given time, as this set may change during the broadcast.

Loop Detection Induced Restarts In section 3.2.4 we introduced a global loop detection mechanism based on regularly shared bloom filters. One of the problems this induces is that we need to induce restarts to increase the size of the bloom filter in order to avoid increasing false positive rates. The main problem here is that for different PEs the global bloom filter will fill up at different times. This is due to the fact that different PEs may be assigned to our job for different spans in time which may be further disjointed as PEs are suspended and subsequently reassigned to a job. However, to uphold our guarantees we want to restart all our PEs as soon as a single PE needs to do so.
To solve this, we rely on the fact that the root PE is guaranteed to remain assigned to a job during the job's full lifetime. Due to this, the root PE takes part in every single loop detection data exchange and it's global filter will contain at least as much data as any other PE's filter. This fact allows us to only ever check on the root PE whether the global bloom filter is full and institute a restart if needed. Doing so lets us avoid any problems that would stem from all PEs performing such checks, such as multiple PEs instituting restarts at the same time.

6.4 Improving the Search Node Exploration Algorithm

One of the main improvements we made to the internal workings of CrowdHTN is to reduce the number of search nodes ever explicitly represented. The main idea behind this optimization is that if the next task we need to resolve is an action, then our search node has only one possible child. As such, this search node does not represent a choice point in our search and we do not need to ever explicitly instantiate it.

More formally speaking, let $t = t_1, \ldots, t_n$ be our sequence of open tasks. Then let $t' = t_1, \ldots, t_k$ be the longest prefix of t which consists of only actions. If $k = n$, we create the next search node by applying all actions, checking preconditions and applying effects as we go. If $k < n$, we create the next search node by applying all actions in t' and then additionally resolving abstract task t_{k+1}.
In addition to reducing the size of our fringe by reducing the overall number of search nodes we create, we specifically hope to save both memory and run time by reducing the number of created and represented world states. This is due to the fact that in our old algorithm resolving tasks $t_1, \ldots, t_k$ would have necessitated the creation of k world states which would also not be shared as in our previously introduced scheme. Reducing the number of search nodes has additional benefits regarding loop detection. Using hash sets we reduce the memory footprint as there are fewer nodes inserted in the visited nodes set and using bloom filters the inherent probability for false positives is less of a problem the fewer nodes we check against the filter.
Lastly, we have used the definition of reductions to further reduce our memory footprint. Remember that each search node is identified by both world state s and open tasks tn. If the first open task in tn is compound, then any child nodes will share the same world state as applying a reduction only ever changes the open tasks. We replicate this in our program by having both search nodes use the same world state instance.

6.5 Efficiently Hashing Nodes of the Search Graph

As we described in section 3.2.3, we hash a search node by hashing all of it's open tasks as well as the full world state. While the size of the world state and thus the time required to hash a world state is bound by the number of ground predicates no such limit exists regarding the open tasks. In other words, hashing both world state and open tasks has an unbounded run time which limits the effectiveness of all hash based loop detection mechanisms. In this section we will describe how we manage to reduce the time required to hash the open tasks to $\mathcal{O}(h \cdot \max\{\#\text{ subtasks of } r | r \in reductions\})$ where a single predicate can be hashed in $\mathcal{O}(h)$.
Let n_1, n_2 be search nodes with n_2 a child of n_1. Let t_1, t_2 be their respective sequences of open tasks with t_1 containing at least 1 abstract task which can be resolved by a reduction r with m subtasks $t_{r_1}, \ldots, t_{r_m}$. Furthermore, let $t_1 = t_{1_1}, \ldots, t_{1_k}$ with $t_{1_1}, \ldots, t_{1_l}$ the longest prefix of only actions. Then $t_2 = t_{r_1}, \ldots, t_{r_m}, t_{1_{l+2}}, \ldots, t_{1_k}$, i.e. the subtasks of r concatenated with all of t_1 except the prefixed actions and the first abstract task. Assuming we compute our order dependent hash over a task network t by going from back to front, for hashing t_2 we can reuse the hash of $t_{1_{l+2}}, \ldots, t_{1_k}$, only computing the hash of $t_{r_1}, \ldots, t_{r_m}$.
Storing the hash with each open task does increase our memory footprint. We do consider the trade-off worth it as the hash is small and it allows us to transform our hash function from an unbound to a bound run time.

Table 3: List of parameters identifying a CrowdHTN configuration

Parameter	Value	Meaning
CrowdHTN Version	O	Old CrowdHTN
	N	New CrowdHTN
Loop Detection Method	Hs	Hash Set
	Bl	Local Bloom Filter
	Bg	Global Bloom Filter
	No	No loop detection
Presence of Restarts	R	Time dependent restarts are used
	/	No time dependent restarts are used

7 Experimental Evaluation

In this section, we first describe our experimental setup concerning both the instances we use as a benchmark as well as the hardware in use in 7.1.

7.1 Experimental Setup

For our performance tests we utilize the same reduced IPC 2020 benchmark as in [9], also giving our planners 900 seconds per instance.
Our tests were done on two machines. The first is a server with an Intel Xeon Gold 6138 processor with 4 sockets, 20 cores per socket and 2 threads per core clocked 2.00G Hz with around 750GB of RAM and running Ubuntu 20.04. We will call it PC1. The second is a server with an AMD EPYC 7702 processor with 1 socket with 64 cores and 2 threads per core clocked 2.00 GHz with around 1TB of RAM and running Ubuntu 20.04. We will call it PC2.

The performance tests were done on a server
We performed an additional test with an instrumented version of single-threaded CrowdHTN using random DFS and hash set based loop detection. The instrumentation allowed us to track information on the structure of our search graph and planner behavior, such as the prevalence of loops and search nodes created per second. This run was performed on the same set of instances as the performance test with up to 300 seconds per instance. The investigation was performed on a server with an AMD EPYC 7702 processor with 1 socket with 64 cores and 2 threads per core clocked 2.00 GHz with around 1TB of RAM and running Ubuntu 20.04.

Naming Scheme As our CrowdHTN planner contains multiple configuration options, we use a succinct naming scheme to identify them in the evaluation. The name for a CrowdHTN configuration has the following structure:

Cr ⟨CrowdHTN version⟩ ⟨#of PEs⟩ ⟨loop detection method⟩ ⟨presence of restarts⟩

A list of possible values for each category as well as their meaning is shown in table 3.

7.2 Optimizations in CrowdHTN

In 6.4 we described an improvement to our progression search which let us reduce the number of search nodes we instantiate. To evaluate the impact of this improvement we created an instrumented version of CrowdHTN which let us track this information during planning. We ran this version of CrowdHTN on our benchmark on PC2, using 1 PE, DFS and hash set

based loop detection while giving it 300 seconds per instance. We tracked the metadata for all instances, whether a plan was found or not. In addition to the information on actions and world states, we tracked the number of search nodes which were duplicates. The results are shown in table 4.
Compared to the other measures, the ratio of tasks which are actions is relatively consistent between domains. It varies from about one third to about two thirds of all tasks. The ratio is lowest for the Logistics-Learned-ECAI-16 domain at 28.2% and highest for Rover-GTOHP at 66.83%.
When it comes to shared world states, results vary more. On 11 out of the 24 test domains, a world state is on average shared by less than 10 search nodes. This is lowest for the Snake domain with only 1.4 search nodes per world state. On the other hand for 6 out of our 24 domains more than 1000 search nodes share one world state, going as far as $\sim 3.5 \times 10^8$ search nodes per world state for the Transport domain.
To sum these improvements up, our improved search node exploration is a clear benefit on all domains, reducing the number of search nodes we need to represent by at least 28%. Sharing world states is more mixed. While sharing is extreme on some domains, it does come at the cost of an additional pointer indirection which may be harmful on domains with little sharing.
Regarding loop detection, the results are similarly varied as with state sharing. On 7 out of 24 domains no duplicate nodes were encountered at all and on another 5 domains less than 1% of nodes were duplicates. At the other end of the spectrum we have Minecraft-Player and Logistics-Learned-ECAI-16 with $\sim$ 30% and Factories-simple with $\sim$ 44% of duplicate nodes.
We will return to these numbers in the evaluation of different loop detection techniques in 7.5.

7.3 Comparing to old CrowdHTN

- we win overall - we loose some coverage?! - specifically in Freecell-Learned-ECAI-16

7.4 Search Algorithms

In section 3.1 we presented four search algorithms that we implemented for CrowdHTN. Those algorithms are random DFS, heuristic DFS, A-star like and BFS. We have tested all four algorithms on our test instance set using 4 PEs and a local bloom filter without restarts for loop detection. The results of this test are visualized in figure 5, a summary of coverage and IPC score is presented in table 5.
Overall, random DFS performed best, followed by heuristic DFS, A-star like search and finally BFS with our best algorithm, random DFS, solving almost twice as many instances and having twice the IPC score of our worst algorithm, BFS. Additionally, we observe a hit-or-miss behavior in both our DFS implementations where plans are either found almost immediately or not at all. Out of the 50 instances solved by random DFS, only 18 were solved in more than 1 and out of these 18 only 9 were solved in more than 10 seconds. BFS on the other hand solves 14 out of 30 instances in more than a second and 11 of these 14 in over 10 seconds. As such, while overall worse performing it does seem to scale better with runtime.
Comparing our two DFS-based approaches, we see that random DFS performs better than heuristic DFS guided by our heuristic from section 3.1.3. We attribute this to the fact that we consciously limited our heuristic to information on the hierarchy available from the lifted instance to reduce the time spent on precomputation. Others, such as [22] argue that heuristics must utilize both hierarchy and world state information. Our findings corroborate this theory.

Table 4: Metadata about progression search on our benchmark

Domain	Action%	State Sharing	Loop%
AssemblyHierarchical	49.71	4793.8	0.006
Barman-BDI	33.56	10.4	4.866
Blocksworld-GTOHP	50.80	6.1	0.000
Blocksworld-HPDDL	49.34	83.4	1.690
Childsnack	53.94	28800.6	0.000
Depots	45.11	7.3	2.790
Elevator-Learned-ECAI-16	47.89	44.9	1.663
Entertainment	49.49	7.2	0.095
Factories-simple	31.09	2.3	43.815
Freecell-Learned-ECAI-16	44.42	2.5	0.000
Hiking	65.24	2.6	12.508
Logistics-Learned-ECAI-16	28.20	4.7	30.348
Minecraft-Player	32.21	4.4	29.896
Minecraft-Regular	31.34	3.4	0.000
Monroe-Fully-Observable	48.65	114.0	1.457
Monroe-Partially-Observable	48.29	106.6	3.418
Multiarm-Blocksworld	47.06	28.9	8.240
Robot	50.00	46207.3	0.002
Rover-GTOHP	66.83	3.5	0.000
Satellite-GTOHP	34.66	52.5	0.013
Snake	47.16	1.4	6.948
Towers	49.99	5409.3	0.000
Transport	36.98	347788104.5	0.000
Woodworking	49.83	21708.8	0.442

Figure 5: Plotting the number of solved instances per run time for CrowdHTN using DFS, heuristic DFS, A-star like search and BFS

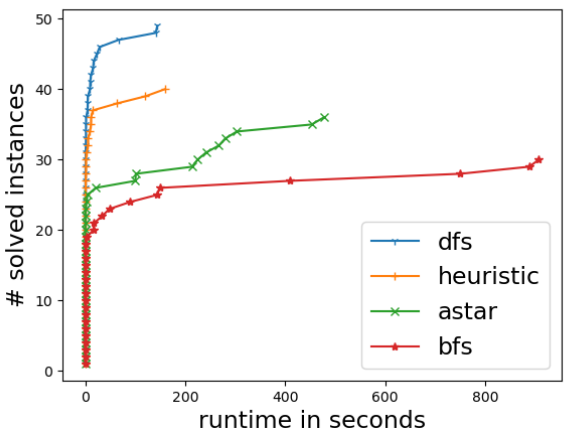
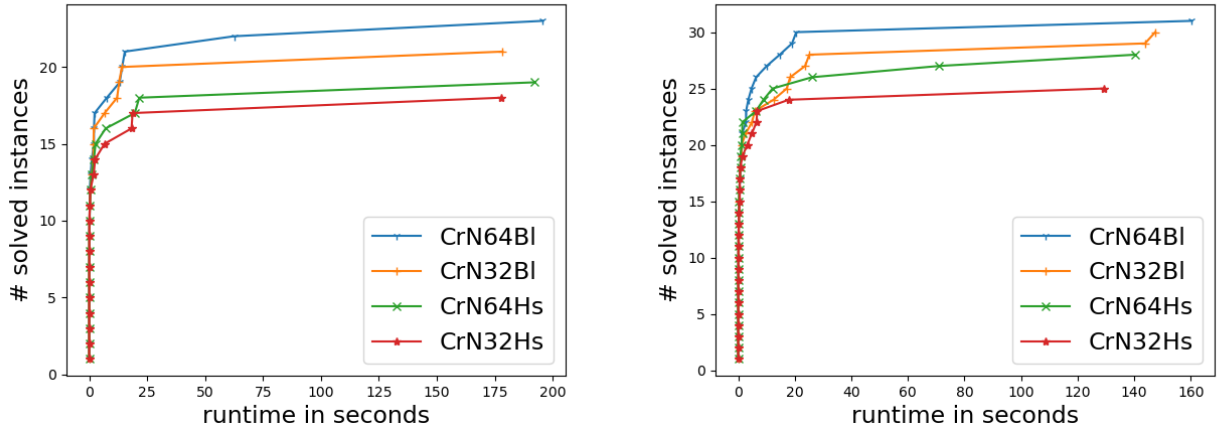


Table 5: Coverage and IPC score of our search algorithms using 4 PEs and a local bloom filter

Algorithm	Coverage	IPC Score
Random DFS	41.7%	43.09
Heuristic DFS	33.3%	35.60
A-star like	38.3%	27.13
BFS	25.0%	21.87

Figure 6: Plotting the number of solved instances per run time for CrowdHTN comparing hash set and local bloom filter based loop detection on 32 and 64 PEs



(a) Instances with $< 1\%$ duplicate nodes

(b) Instances with $> 1\%$ duplicate nodes

7.5 Local Loop Detection

- general loop detection: - loop hit rate - global loop hit rate - performance of hash set vs bloom filter

- add runs without any loop detection to the evaluation - split perf comparison on domains with and without loops - table which gives us domain and loop percentage

- as bloom filters perform best, the rest of the evaluation will focus on them

7.6 Probabilistic Restarts

- test the probabilistic restarts - with 900 seconds we expect $\sum_{t=1}^{899} \frac{1}{t} \approx 7.38$ restarts per run

Table 6: Coverage and IPC score while using hash set and local bloom filter based loop detection on 32 and 64 PEs

Configuration	Coverage	IPC Score
Bloom, 64 PEs	45.0%	47.37
Bloom, 32 PEs	42.5%	44.24
Hash set, 64 PEs	39.2%	41.84
Hash set, 32 PEs	35.8%	38.71

Figure 8: Evaluating CrowdHTN with a local bloom filter with and without restarts

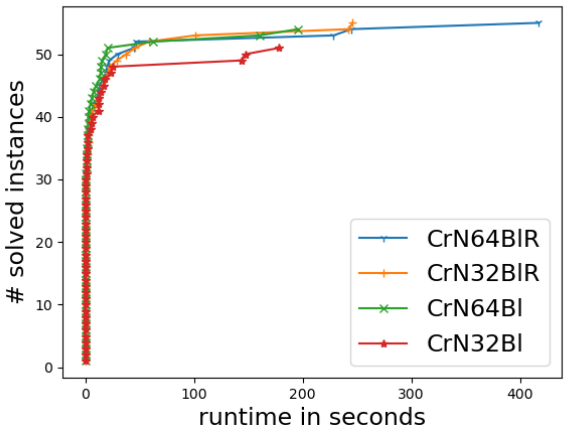
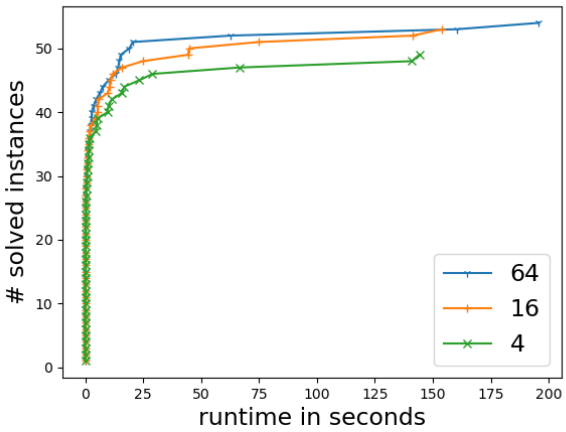


Figure 9: Plotting the number of solved instances per run time for CrowdHTN using DFS and a local bloom filter on 64, 16 and 4 PEs



7.7 Global Loop Detection

- test the global loop detection with restarts on 32 and 64 PEs on our full benchmark -

7.8 Scalability of CrowdHTN

- demonstrating the scaling behavior of CrowdHTN is hard due to the hit-or-miss of random DFS in TOHTN - we need an instance that is reliably solved - we need an instance that takes long enough to solve to demonstrate an effect - Monroe Fully Observable is such an instance - separate test on i10pc137 (mention again what it is like, CPU etc) - show scaling on a nice instance

Table 7: Evalutating CrowdHTN on 20 instances of the Monroe-Fully-Observable domain

	BL4		**BL16**		**BL64**		**BG4**		**BG16**		**BG64**	
	Time	IPC	Time	IPC	Time	IPC	Time	IPC	Time	IPC	Time	IPC
01	0.2	1.00	0.1	1.00	0.4	1.00	0.1	1.00	0.7	1.00	0.1	1.00
02	161.5	0.25	30.7	0.50	1.3	0.96	115.6	0.30	10.5	0.65	22.6	0.54
03	/	0.00	5.7	0.74	8.4	0.69	250.5	0.19	4.0	0.80	30.5	0.50
04	0.4	1.00	0.3	1.00	0.2	1.00	0.3	1.00	0.3	1.00	0.4	1.00
05	49.9	0.43	55.6	0.41	22.6	0.54	23.7	0.53	30.2	0.50	26.6	0.52
06	183.9	0.23	60.1	0.40	3.5	0.82	129.0	0.29	61.7	0.39	18.2	0.57
07	18.5	0.57	7.2	0.71	3.2	0.83	5.3	0.75	2.0	0.90	4.0	0.80
08	98.1	0.33	48.0	0.43	4.4	0.78	108.0	0.31	101.7	0.32	12.5	0.63
09	62.3	0.39	17.7	0.58	26.0	0.52	70.8	0.37	13.5	0.62	14.3	0.61
10	122.1	0.29	33.9	0.48	23.8	0.53	80.4	0.35	61.9	0.39	11.0	0.65
11	148.9	0.26	60.5	0.40	15.7	0.60	148.6	0.26	62.8	0.39	15.0	0.60
12	137.5	0.28	47.9	0.43	19.4	0.56	223.4	0.20	34.7	0.48	25.6	0.52
13	19.4	0.56	2.7	0.85	1.5	0.94	7.9	0.70	0.6	1.00	1.7	0.92
14	171.3	0.24	61.1	0.40	36.1	0.47	279.5	0.17	36.2	0.47	42.1	0.45
15	/	0.00	6.5	0.73	4.3	0.79	/	0.00	5.0	0.76	0.8	1.00
16	/	0.00	6.3	0.73	2.0	0.90	7.1	0.71	6.2	0.73	4.8	0.77
17	/	0.00	1.6	0.93	11.2	0.64	/	0.00	/	0.00	1.8	0.91
18	16.7	0.59	40.0	0.46	10.3	0.66	47.2	0.43	9.0	0.68	37.7	0.47
19	1.9	0.90	5.1	0.76	1.8	0.91	15.7	0.60	15.0	0.60	5.7	0.74
20	21.7	0.55	2.9	0.84	2.5	0.86	12.5	0.63	6.0	0.74	4.6	0.78
		7.88		12.78		15.01		8.81		12.43		13.98

7.9 Malleable CrowdHTN

- once again we need an instance which scales well - see previous section, Monroe-Fully-Observable gives us such instances - problem 11 specifically shows the behavior we are after - use Mallob on 32 PEs - give our job 60 seconds to solve - every 20 seconds, introduce another job with a time limit of 10 seconds which we are sure will not be solved - our Monroe instance oscillates between 32 and 16 PEs every 10 seconds - on average we have 40 PEs available - run test 100 times? - compare with 32 and 16 PEs (unhindered) - maybe also compare with a dummy job every 10 seconds which lasts for 5 seconds - same availability of PEs but more rebalancing - important regarding the way we deal with disappearing PEs, i.e., communicating the local search root to someone else

7.10 Conclusion

- final discussion stuff

Things to plot: - old Crowd on 4, 16, 64 cores - new Crowd on 4, 16, 64 cores - new Crowd with bloom filter, hash set and without ld, 32, 64 - new Crowd with bloom filter with(out) restarts, 32, 64 - new Crowd with global loop detection - sequential planners HyperTensioN and PANDA - Malleable vs non moldable Crowd

Table 8: Domain-wise comparison of the old and improved moldable CrowdHTN

	CrO4Hs		**CrO16Hs**		**CrO64Hs**		**CrN4Bl**		**CrN16Bl**		**CrN64Bl**	
Domain	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov
AssemblyHierarchical	**1.0**	20%	**1.0**	20%	0.98	20%	**1.0**	20%	**1.0**	20%	**1.0**	20%
Barman-BDI	1.79	40%	1.78	40%	1.74	40%	**2.0**	40%	**2.0**	40%	1.97	40%
Blocksworld-GTOHP	2.0	40%	1.99	40%	**2.49**	60%	2.0	40%	2.0	40%	2.39	60%
Blocksworld-HPDDL	**3.21**	80%	3.14	80%	3.01	80%	3.03	80%	3.02	80%	2.98	80%
Childsnack	2.6	80%	2.55	80%	2.37	80%	**2.96**	80%	2.93	80%	2.82	80%
Depots	3.63	80%	3.72	80%	3.6	80%	**3.92**	80%	3.86	80%	3.85	80%
Elevator-Learned-ECAI-16	4.06	100%	4.03	100%	3.86	100%	4.33	100%	**4.36**	100%	4.29	100%
Entertainment	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Factories-simple	1.91	40%	**1.93**	40%	1.86	40%	1.0	20%	1.0	20%	1.0	20%
Freecell-Learned-ECAI-16	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Hiking	**2.0**	40%	1.75	40%	1.7	60%	**2.0**	40%	**2.0**	40%	**2.0**	40%
Logistics-Learned-ECAI-16	2.57	60%	**2.87**	80%	2.79	80%	0.0	0%	0.0	0%	0.0	0%
Minecraft-Player	1.73	40%	1.7	40%	1.62	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%
Minecraft-Regular	3.14	80%	3.1	80%	2.99	80%	**3.6**	80%	3.58	80%	3.5	80%
Monroe-Fully-Observable	1.68	100%	2.45	100%	2.07	100%	1.92	60%	3.3	100%	**4.08**	100%
Monroe-Partially-Observable	0.93	20%	0.0	0%	0.82	20%	**1.0**	20%	0.0	0%	**1.0**	20%
Multiarm-Blocksworld	**1.0**	20%	**1.0**	20%	0.98	20%	0.0	0%	**1.0**	20%	**1.0**	20%
Robot	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%
Rover-GTOHP	3.7	100%	3.62	100%	3.26	80%	**4.56**	100%	4.55	100%	4.47	100%
Satellite-GTOHP	0.0	0%	0.0	0%	0.0	0%	0.0	0%	**1.0**	20%	0.0	0%
Snake	3.07	80%	3.14	80%	2.84	80%	3.1	80%	**3.89**	100%	3.4	80%
Towers	2.2	60%	2.2	60%	2.15	60%	**2.66**	60%	2.65	60%	2.61	60%
Transport	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	**1.0**	20%
Woodworking	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Instances: 120	44.2	47%	44.0	47%	43.1	48%	43.1	41%	46.1	44%	47.4	45%

8 Future Work

- work stealing is extremely helpful with malleability on the integrating new PEs side of things
- well-coordinated restarts solve a number of problems (completeness for random DFS, dealing with disappearing workers, probabilistic loop detection)

8.1 A Common (TO)HTN Interface

- be able to mix grounders, pruners, planners - maybe even loop detection! - be able to combine different pruners - use a lifted pruner early and then plug in a grounder and grounded pruner afterwards - PANDA seems to try to provide this - quite successful regarding the input format for HTN planning which finally makes planners easily comparable, advancing the state of the art - sadly not as successful for the rest of planning - we believe their to be great potential in

8.2 Combine Pruning Approaches

- HyperTensioN and Panda base their pruning on different paradigms - HyperTensioN's pruning could be extended to work on grounded instances where it would have even more information available - both these approaches are structurally different and could thus be used to compliment each other - whether the better pruning would offset the increased runtime cost remains to be seen

8.3 Advances runtime pruning techniques

In section 4 we saw the problems with current loop detection techniques regarding (TO)HTN planner completeness. Specifically, loop detection in heuristic progression search suffers from recursive tasks which introduce additional new tasks which have to be resolved after the recursive task itself. At the same time, the PANDA planner has shown that heuristic progression search with loop detection is a promising approach with state of the art performance [22], [19]. As a result, we propose that we take the notion of loop detection and expand it to a more generalized definition of dynamic pruning. In the past we were restricted to filtering out search nodes that are duplicates and as such already known. For future research, we could instead look at tasks sets and see whether they are still feasibly performing productive work. As an example take a task t which recursively resolves into two new instances of t. Additionally, let $\mathcal{Q}_t$ be the set of predicates which may be changed as a result of resolving t. Then any sequence of t which has length at least $2^{\mathcal{Q}_t}$ will necessarily perform duplicate work and any resolution which leads to an even longer sequence of t needs not be explored.
In this way, using improved dynamic pruning of unproductive task sequences may allow us to build planners which are both complete while at the same time realizing the performance gains seen in heuristic progression search.

8.4 Lifted Parallel Search

- the Crowd way of performing relatively simple search does not work out in the end - we can still see some scaling for the parallel case - lifted planning could massively shrink our search space (by an exponential factor in the number of nodes!) - lilotane provides much intelligence on how to perform lifted search - same as HyperTensioN - a lilotane-like behavior may be the best from a practical perspective, as it is more consistent - a search-based formulation may

be easier to both parallelize and adapt to a malleable context - any search may be plugged in where CrowdHTN currently resides

8.5 Improvements Unlocked by a Shared Memory Implementation

- we have decided to stay with CrowdHTN's way of performing grounding on-demand and just in time, as it lends itself particularly well to a malleable environment - specifically, having to perform a big chunk of work that is not yet possible to lead to a plan would impose a problem on the efficiency of new and short-lived workers - in a shared-memory environment, this could be done once by the root and then re-used by other workers, making better pruning and heuristics such as in PANDA available

8.6 Global Loop Detection

- we show how a global loop detection mechanism in (TO)HTN planning could work - so far, we use a very simple heuristic (encounter a node twice) for which nodes to share - more intelligent heuristics are probably helpful - we are interested in search nodes often reached, less in recursive tasks (can be resolved locally)

8.7 Intelligent Restarts

- we perform restarts according to the harmonic series - SAT solving already knows restarts, explore inner-outer and luby sequence for potentially better performance

References

[1] Alford, Ron, Gregor Behnke, Daniel Höller, Pascal Bercher, Susanne Biundo and David W Aha: *Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems.* In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

[2] Behnke, Gregor, Daniel Höller and Susanne Biundo: *Finding Optimal Solutions in HTN Planning-A SAT-based Approach.*

[3] Behnke, Gregor, Daniel Höller and Susanne Biundo: *On the complexity of HTN plan verification and its implications for plan recognition.* In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 25–33, 2015.

[4] Behnke, Gregor, Daniel Höller and Susanne Biundo: *totSAT-Totally-ordered hierarchical planning through SAT.* In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[5] Behnke, Gregor, Daniel Höller and Susanne Biundo: *Tracking branches in trees-A propositional encoding for solving partially-ordered HTN planning problems.* In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 73–80. IEEE, 2018.

[6] Behnke, Gregor, Daniel Höller, Alexander Schmid, Pascal Bercher and Susanne Biundo: *On succinct groundings of HTN planning problems.* In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9775–9784, 2020.

[7] Bender, Michael A, Martin Farach-Colton, Rob Johnson, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane and Erez Zadok: *Don't thrash: how to cache your hash on flash.* In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*, 2011.

[8] Bloom, Burton H: *Space/time trade-offs in hash coding with allowable errors.* Communications of the ACM, 13(7):422–426, 1970.

[9] Bretl, Colin, Niko Wilhelm Dominik Schreiber and Peter Sanders: *Parallel and Distributed TOHTN Planning.*

[10] Broder, Andrei and Michael Mitzenmacher: *Network applications of bloom filters: A survey.* Internet mathematics, 1(4):485–509, 2004.

[11] Buisson, Jérémy, Françoise André and Jean-Louis Pazat: *A framework for dynamic adaptation of parallel components.* In *International Conference ParCo*, volume 33, page 65, 2005.

[12] Cirne, Walfredo and Francine Berman: *A model for moldable supercomputer jobs.* In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 8–pp. IEEE, 2001.

[13] Erol, Kutluhan, James Hendler and Dana S Nau: *HTN planning: Complexity and expressivity.* In *AAAI*, volume 94, pages 1123–1128, 1994.

[14] Erol, Kutluhan, James Hendler and Dana S Nau: *Complexity results for HTN planning.* Annals of Mathematics and Artificial Intelligence, 18(1):69–93, 1996.

[15] Fan, Bin, Dave G Andersen, Michael Kaminsky and Michael D Mitzenmacher: *Cuckoo filter: Practically better than bloom.* In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[16] Fan, Li, Pei Cao, Jussara Almeida and Andrei Z Broder: *Summary cache: a scalable wide-area web cache sharing protocol.* IEEE/ACM transactions on networking, 8(3):281–293, 2000.

[17] Feitelson, Dror G: *Job scheduling in multiprogrammed parallel systems*. 1997.

[18] Georgievski, Ilche and Marco Aiello: *HTN planning: Overview, comparison, and beyond*. Artificial Intelligence, 222:124–156, 2015.

[19] Höller, Daniel and Gregor Behnke: *Loop Detection in the PANDA Planning System*. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 168–173, 2021.

[20] Höller, Daniel, Gregor Behnke, Pascal Bercher and Susanne Biundo: *Language Classification of Hierarchical Planning Problems*. In *ECAI*, pages 447–452, 2014.

[21] Höller, Daniel, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier and Ron Alford: *HDDL: An extension to PDDL for expressing hierarchical planning problems*. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9883–9891, 2020.

[22] Höller, Daniel, Pascal Bercher, Gregor Behnke and Susanne Biundo: *HTN planning as heuristic progression search*. Journal of Artificial Intelligence Research, 67:835–880, 2020.

[23] Hungershofer, Jan: *On the combined scheduling of malleable and rigid jobs*. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 206–213. IEEE, 2004.

[24] Magnaguagno, Maurício C, Felipe Rech Meneguzzi and LAVINDRA DE SILVA: *HyperTensioN: A three-stage compiler for planning*. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS), 2020, França.*, 2020.

[25] Mali, Amol Dattatraya and Subbarao Kambhampati: *Encoding HTN Planning in Propositional Logic*. In *AIPS*, pages 190–198, 1998.

[26] Nau, Dana, Yue Cao, Amnon Lotem and Hector Munoz-Avila: *SHOP: Simple hierarchical ordered planner*. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973, 1999.

[27] Nau, Dana S: *Current trends in automated planning*. AI magazine, 28(4):43–43, 2007.

[28] Ramoul, Abdeldjalil, Damien Pellier, Humbert Fiorino and Sylvie Pesty: *Grounding of HTN planning domain*. International Journal on Artificial Intelligence Tools, 26(05):1760021, 2017.

[29] Sanders, Peter and Dominik Schreiber: *Decentralized online scheduling of malleable NP-hard jobs*. In *European Conference on Parallel Processing*, pages 119–135. Springer, 2022.

[30] Schreiber, Dominik: *Lilotane: A lifted SAT-based approach to hierarchical planning*. Journal of Artificial Intelligence Research, 70:1117–1181, 2021.

[31] Schreiber, Dominik, Damien Pellier, Humbert Fiorino et al.: *Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning*. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 382–390, 2019.

[32] Schreiber, Dominik and Peter Sanders: *Scalable SAT solving in the cloud*. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 518–534. Springer, 2021.

[33] Sonmez, Ozan, Hashim Mohamed, Wouter Lammers, Dick Epema et al.: *Scheduling malleable applications in multicluster systems*. In *2007 IEEE International Conference on Cluster Computing*, pages 372–381. IEEE, 2007.

[34] Tucker, Andrew and Anoop Gupta: *Process control and scheduling issues for multiprogrammed shared-memory multiprocessors.* In *Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 159–166, 1989.

[35] Xie, Kun, Yinghua Min, Dafang Zhang, Jigang Wen and Gaogang Xie: *A scalable bloom filter for membership queries.* In *IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference*, pages 543–547. IEEE, 2007.