Karlsruher Institut für Technologie

Bachelorarbeit

Malleable TOHTN Planning using CrowdHTN and Mallob

Niko Wilhelm

Abgabedatum: 19.10.2012

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 27, 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

Algorithmenverzeichnis

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 - 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

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 ([20]): 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

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.

quote m formalis

Multiple definitions for HTN planning exist. In this work we build on the definition introduced in [17].

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**.*

directly solution ask ork if ving is ed be- method!

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 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.

d def- n: re- ng a re- s it the net- !

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 [12] and [13] 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, [12] and [13] showed early on that HTN instances can be used to simulate context-free languages. This was extended by [19] 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.

[2] ment
4 ways
comput
per bou
on the
network
size?

2.2.3 Differences from other Kinds of Planning

[25] 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.
[20] argue that HTN-planning is not simply a domain-configurable version of classical planning and argue on the basis that [12, 13] showed that HTN-planning is strictly more powerful compared to classical planning which is PSPACE-complete.
While we agree with [20], 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.

2.3 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

instead
AT-
l"?

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 ([23]), 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 ([29]) and Lilotane ([28]).
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 [29], 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 [20]. Plan space search, as the name says, searches the space of partial plans, aiming to fix flaws - take definitions from [20] 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 ([24]), HyperTensioN ([22]), PANDA ([20]) and our own planner CrowdHTN which will be presented in a later section 2.6.

Algorithm 1: Classical Progression Search for HTN as introduced in [20]

```
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')
```

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 ([26], [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].
Planners may choose to operate on either lifted or ground instances. A discussion on the trade-offs involved is found in [28]. 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 [29]),

lifted translation-based planners (Lilotane [28]) and also search-based planners that work on both lifted (HyperTensioN [22]) and ground representations (PANDA [20]).
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.

2.5 Malleability

In this work, we follow the classification of [16] 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 [31]. In practise, we see that most jobs run on supercomputers follow the moldable model [11]. The moldable model is also supported by programming environments such as MPI [21].
Systems that utilize malleable jobs have been shown to be highly efficient [16]. They achieve this in multiple ways. First, they allow for efficient scheduling, as the scheduler can reevaluate and change previously made decisions [31]. 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 [16], [21]. Second, they allow applications to utilize additional resources as they become available, leading to improved performance [21]. Lastly, [10] 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 [16]. 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 [16], [32]. 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 [16]. 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 [16] 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 **Ma**lleable **Lo**ad **B**alancer and the **M**ulti-tasking **A**gile **Lo**gic **B**lackbox [30]. 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 [27].

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 [30]. 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 TOHTN Metadata

- researchers in TOHTN planning have collected a number of test instances for TOHTN planning - provide some analysis of those instances in the context of modelling TOHTN planning as graph search - provide a foundation to discuss the effects of changes and improvements in the crowd planning framework

restarts: - how to propagate the version - how do new workers receive their versions - root controls the whole thing (lifetime of root is lifetime of job, always has maximum information/ nr of nodes) - independent hash functions: we change the seed with each version change - seed is initially distributed s.t. everyone starts with the same, change is deterministic with each version increase (later PEs can catch up)

4 Theoretical Improvements to the Crowd Planner

4.1 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 [20] 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
- Random breadth-first search
- Greedy best-first search
- A-star like search

4.1.1 Random Depth-First Search

Randomized depth-first search 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 to keep the random properties of the original CrowdHTN. This is done to avoid any pathological cases where e.g. the first method applicable to a task leads to an endless loop. We do not expect any differences in behavior or performance compared to the previous implementation.

4.1.2 Random Breadth-First Search

Randomized breadth-first search 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 depth-first-search. We do this as the number of search nodes may be exponential in the depth, e.g. if each task can be resolved by at least two reductions. As such, the order in which we explore the nodes of any layer may still have a big impact and in this way we avoid pathological behavior.
In general, we expect a higher memory footprint compared to depth-first search and assume that the planner will struggle with domains where either 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 queue. At the same time, we expect the performance of breadth-first search to be a lot more consistent than for depth-first search, as the layer at which we find a plan stays fixed for any single instance.
Overall, we do not expect high performance of our breadth-first search. It may however prove useful on some domains and help us understand and validate assumptions about the behavior of TOHTN problems.

4.1.3 Heuristic Search

Both depth-first and breadth-first search are unguided and do not depend the order of search node exploration on any information contained in those nodes. Other planners, such as PANDA, use heuristics to guide their search. We will describe general information about heuristics and their use in PANDA according to [20] and then describe how we try and adapt the use of heuristics for malleable TOHTN planning.

Heuristics in HTN in general and PANDA specifically The general idea of heuristics is to avoid exploring all of our search space. They 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 would be to adapt classical planning heuristics. This proves difficult, however, 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 such issues, the PANDA planner goes the other way around. 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 while remaining relatively efficient.

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 runtimes of grounding and can be accordingly high.
While such a grounding is already available in PANDA, CrowdHTN does not perform explicit grounding before planning. Doing so may prove interesting for a parallel planner where the grounding would only be performed once. In a malleable environment without shared memory we can expect this grounding to take place every time a new worker is added, adding a high startup cost. Worse, a short-lived worker may be interrupted while still grounding, never getting around to any planning work. 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 easily be used in malleable planning. The goal is to have little startup overhead and to retain the efficient evaluation at each planning step. While this necessarily limits any precomputation to the lifted instance and the precision of our heuristic comes with problems as discussed in the previous paragraphs, we hope to still find some performance gains on at least some problem instances.
As heuristic value, we simply 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 remaining depth for each action is initialized to zero. The remaining depth for each abstract task is initially unknown. In each step we loop over all tasks t whose remaining depth is unknown. For each method m of task t we check the depth of all subtasks. If all depths are known, the method depth is set to the sum of all subtask depths plus one. For each task we choose the minimum value over all available reductions.
If in any iteration we do not get a new depth for at least one task, we stop the computation. Any task which does not have an assigned depth at this point is not resolvable at all and can be ignored.

Let t_a be the set number of abstract tasks, t_p be the number of primitive tasks and m the number of methods. A single iteration takes time in $\mathcal{O}(t_a + m)$. The number of overall iterations required is bound by t_a, as in any iteration at least one abstract task will be assigned it's final score, as we do not have any negative cycles. This gives us an overall runtime bound of $\mathcal{O}(t_p + t_a \cdot (t_a + m))$. As all those numbers are relating to the lifted instance we expect the overall runtime to be small in practise. To evaluate our heuristic while planning, we now

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))}
```

need to look not only at one but at the whole sequence of open tasks and calculate the sum of our heuristic over those tasks. While the naive solution gives us linear runtime in the number of open tasks, we can stretch the computation over task instantiation and reuse parts of it to perform heuristic evaluation in $\mathcal{O}(1)$ at runtime. [**?**] 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.

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 guided depth-first search and an A-star like search.
The implementation of depth-first search 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 depth-first search. We have to note that this is still not necessarily deterministic. Randomness comes into play both when two reductions have the same heuristic score - which happens for different instantiations of the same method - and when performing work stealing in a parallel setting.
In addition to the guided depth-first search, we also implemented an A-star like search where a node's value is the sum of the heuristic value and the number of applied reductions. 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. The hope is that the heuristics 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 pathological cases.

4.1.4 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 effect the performance characteristics

Table 1: Completeness of the different search algorithms in CrowdHTN

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

of our planner. The search algorithm has a more fundamental impact than that, however, and may effect the property of completeness. In this section we want to give a short overview over the completeness of each of the algorithms. A short overview can be seen in table 1. Note that this discussion is only about the algorithms without any modifications. In section 4.2 we discuss loop detection mechanisms to improve those algorithms which are so far not listed as complete.

Depth-first search 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 to before the start of this 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 and the goal is found even if other paths may lead into endless loops.

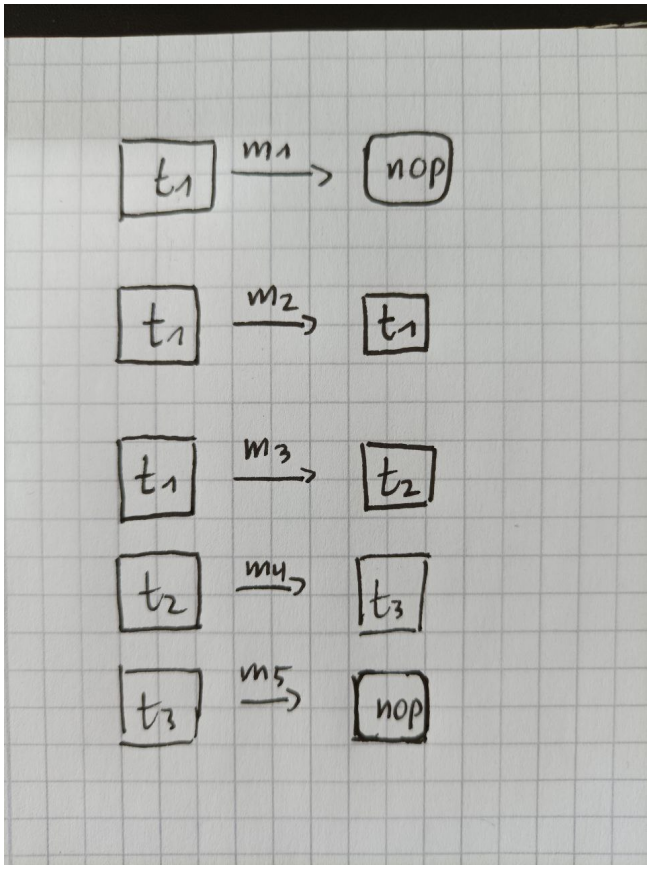
Breadth-first search on the other hand is trivially complete. While the exploration order within each layer is random we can provide an upper bound on the number of steps required to explore a given search node n on layer i. One such bound is the sum of all layer sizes from 0 up to and including i.

Next in our list is heuristic depth-first search. Similar to random depth-first search it may run into an endless loop. Compared to depth-first search, however, heuristic depth first search 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 1. Here rectangles stand for tasks, rounded rectangles for actions and arrows to reductions. Assume we want to resolve task t_1 where the preconditions of m_1 do not hold while those of m_2, m_3, m_4, m_5 are fulfilled. Our heuristic will want us to resolve t_1 via m_1 which does not work. Then the next shortest path to a full resolution is through m_2, m_1. m_2 will be applied successfully to recurse, m_1 will fail again and the cycle continues like this forever. The path through m_3, m_4, m_5 has length 3 and will thus never be taken.

Lastly, we implemented our heuristic 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. We can easily extend the incompleteness of random and heuristic depth-first search to the parallel case. If we have an upper bound p on the number of parallel processes we can build a domain as in figure 1 which does not induce 1 but p endless looping opportunities that either trigger our heuristic or may simply be taken due to random chance. In this way, using parallelism does not get us to completeness but it does improve the our chances of at least 1 process taking the correct path.

Figure 1: A pathological case in our new HTN heuristic



Switch
'comple
ness' fo
'loosing
parts of
our sear
space'?

replace
panding
bloom f
with sca
able blo
filters a
the corr
name

4.2 Loop Detection

In recent years it has become clear that the recursive nature of (TO)HTN instances poses its own set of challenges to (TO)HTN planners. As a result, mechanisms to perform loop detection have become an active area of research with both HyperTensioN [22] and PANDA [18] exploring it. In this section we will discuss loop detection specifically in the context of parallel and distributed (TO)HTN planning. We start out with a discussion of loop detection techniques in other planners such as PANDA and HyperTensioN in section 4.2.1. This is followed by a short overview of how CrowdHTN specifically differs and how this changes our base assumptions in section 4.2.2. Afterwards we first explore loop detection as it is already present in CrowdHTN in section 4.2.3. We conclude by exploring how approximate-membership-query (AMQ) data structures can be used in (TO)HTN 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 4.2.4.

4.2.1 Loop Detection in Other (TO)HTN Planners

Loop detection in (TO)HTN planning is a recent phenomenon and was introduced in 2020 by the HyperTensioN planner ([22]) with the so-called 'Dejavu' technique. Dejavu works by extending the planning problem, introducing primitive tasks and predicates that track and identify when a particular recursive compound task is being 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 [18] and means that HyperTensioN is not complete. [18] 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 [22], loop detection was introduced to the PANDA planner in [18]. Similar to HyperTensioN, PANDA keeps it's loop detection information separate in a list of visited states, $\mathcal{V}$. Search nodes, identified by a tuple (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 speed up comparisons, $\mathcal{V}$ is separated into buckets according to a hash of s. To then identify whether $tn \in \mathcal{V}[s]$ multiple algorithms are proposed. 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. Both hashing-based and direct comparison as used in PANDA have a performance cost linear in the size of tn. The loop detection in PANDA improves upon the one in HyperTensioN insofar as it is able to detect loops where equivalent states where reached independently.

4.2.2 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.
Crowd identifies search states as a tuple of (s, tn) of world state s and task network s, similar to PANDA and uses hashing to efficiently identify duplicate search states. 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 use probabilistic loop detection.
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 using the full state information becomes infeasible as those full states would have to be communicated. If we perform loop detection only locally, we expect to suffer from decreased performance the higher the degree of parallelism. I.e., if two branches of our search tree contain the same search node, the probability that those two branches are encountered on different processors is higher the more processors we have.

4.2.3 Perfect Loop Detection

One simple way to perform loop detection which is similarly used in PANDA ([18]) is to use a hashset 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 program.
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. For performance reasons we chose to combine 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.

4.2.4 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. Not all (TO)HTN planners make this assumption. As an example, the HyperTensioN planner does allow false positives to occur, leading to a high-performing planner at the cost of completeness as parts of the search space are never explored [22]. 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 TOHTN loop detection specifically. 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 Approximate-membership-query (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 be more memory efficient [7]. They were introduced with the bloom filter in 1970 [8]. Since then both variations of bloom filters, such as counting bloom filters [15], and other AMQ data structures have been introduced, among them quotient filters [7] and cuckoo filters [14].
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 can be defined by three numbers, the number of bits in the filter m, initially all set to zero, the number of hash functions used 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 in 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 query in time $\mathcal{O}(k \cdot t)$ where t 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 [15]. Instead of a single bit per element, multiple bits are used which contain a counter tracking the number of elements belonging to any one 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 [9]. 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

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

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 low probability of false positives. To fix this and guarantee an upper bound on the rate of false positives even in dynamic sets, the concept of scalable bloom filters was introduced [33]. 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. When we perform a membership query we now have to check at all levels of the filter. While there is a performance overhead, it was shown that 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 an unnecessarily 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 forego mechanisms as used in 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 the fact that false positives become a possibility. As a result, we can expect to wrongfully cut off 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 (TO)HTN 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 an expanding bloom filter with maximum false positive rate $p < 1$ and a hash function which maps search nodes to uniformly independent values. Let us further assume we perform progression search on a solvable (TO)HTN problem where the shortest path from start to goal is of length l. Then $(1-p)^l < 1$ is an upper bound on the probability that we are unable to solve the problem even though a solution exists.
From this we see that $\lim_{l\to\infty}(1-p)^l \to 0$. 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 ot re-run our planner. There are three main constraints.

(i) As the (TO)HTN 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 t 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 total number of restarts we end up with $\sum_{t=1}^{\infty} \frac{1}{t}$. This is the harmonic series and diverges, giving us the required unbounded number of restarts. As t grows, the probability of restarts decreases, allowing for increasingly long runs, fulfilling the third constraint.
This mechanism allows us to restore completeness to our planner while utilizing approximate loop detection. 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 previous section 4.2.2 we already mentioned that loop detection in distributed (TO)HTN planning comes with unique problems. Specifically, current loop detection techniques do not include ways to efficiently share the loop detection informa-

tion between PEs. As a result, a search node is only fully filtered out once each worker has encountered it at least once. This is less of a problem for recursive tasks which are quickly re-encountered by our 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 4.2.3 on current loop detection mechanisms, they do suffer 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 message overhead. 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. 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 this loop detection mechanism to the distributed case.

Compared to hash sets, loop detection based on bloom filters offers 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, the merging operation of two bloom filters is very efficient. To merge two bloom filters, we perform 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 adding 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 the local case, we introduced expanding 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 either 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 expanding 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 upon filling our global filter. 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. This limits the number of restarts needed and increases the time between subsequent loop detection induced restarts, once again allowing for arbitrarily long runs to preserve completeness of our planner. Additionally, we limit the amount of information present in our filter to further reduce the number of restarts we need. We achieve 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.

4.3 Completeness Properties of (TO)HTN Planners

In our presentation on different search algorithms implemented in CrowdHTN we already did a short discussion on the completeness of different algorithms in section 4.1.4. 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 there always exists a plan up to 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 it either. 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 search 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 ([29]) and Lilotane ([28]). 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 as they are.
In contrast to this, we have the space of search-based planners, starting with HyperTensioN [22]. For HyperTensioN, the authors themselves note that their inbuilt loop detection mechanism suffers from false-positives with no mechanism to mitigate them [22]. 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 [20]. For loop detection PANDA offers hashing based mechanisms both with and without a fallback to full search node comparison [18]. 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 which does not suffer from 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.

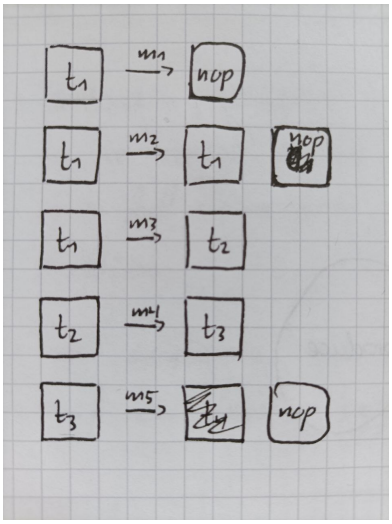
Loop detection and completeness As we have seen, heuristic search on its own may increase planner performance but comes at the cost of completeness. We will now explore the implications of combining heuristic search with loop detection to see how this changes the overall situation. In this paragraph we are only interested in loop detection mechanisms that do not suffer from false positives as this automatically disqualifies a planner from being complete. Additionally, we argue from the perspective of progression search where each search node is uniquely identified by the combination of open tasks and world state.
In general, loops are only a problem in (TO)HTN 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. Here 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

In all three cases, a task recursing into itself also implies that the set of legal orderings is a subset of the original set of orderings.
The first case is the easiest to detect and to 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 a better 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 (TO)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 are guided back into still unexplored parts of the search space. In practice we can often obtain a lower 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 task 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 for our previous case, this is not possible here, as the number of open tasks and thus the number of sets of open tasks is unbounded. More specifically, loop detection as used in PANDA and CrowdHTN is unable to handle this case. Figure 2 provides an example of an instance which would guide our proposed CrowdHTN heuristic into the recursion while not creating loops, as each recursion changes the open tasks by extending the sequence. As mentioned in the previous paragraph, we assume that similar cases can be constructed for other (TO)HTN heuristics.

Restarts and completeness So far we have seen that search algorithms with a BFS-like behavior are easily complete. Similarly, we have seen that heuristic search may suffer from pathological cases. This leaves DFS as an open case, where any plan can theoretically be found but not every plan will, due to getting lost in infinite loops. In the previous paragraph we have looked at how loop detection aims to fix this problem and have seen that it does not work for all cases. Now we will look at the restart mechanism we introduced in section 4.2.4 to mitigate false positives in our approximate loop detection and see how it affects overall completeness.

Figure 2: Pathological instance for our proposed heuristic that is not caught by loop detection



We can make a similar argument as in our previous discussion. In random DFS any path has probability $p > 0$ to be taken. It follows that, as the number of restarts approaches infinity, the probability to take this path at least once approaches 1. This has the additional constraint that we do not only need an unbounded number of restarts but additionally need an unbounded number of runs in between restarts with length at least t for any $t > 0$. Without this constraint we may take the right path but always terminate before we are able to complete it. Our restart mechanism fulfills both of these constraints, using probabilistic restarts with probability $\frac{1}{t}$ after t seconds. With this mechanism, DFS in CrowdHTN gives us a complete planner.

Conclusion In this section we have taken a look at completeness in (TO)HTN 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 easiest to turn into a complete planner while heuristic search may always suffer from pathological cases. Additionally, we see that loop detection is able to solve many but not all problems with recursive (TO)HTN instances. In the end, we introduce a restart mechanism and argue that it is able to mitigate problems with false positives in loop detection and allows random DFS to achieve completeness in all cases.

5 Malleability in the Context of CrowdHTN

- we need to distribute the problem description efficiently (as we need to do it time and time again) - efficiently here means little communication overhead - choose to do the lifted instance - ease of implementation, we do not see it as a core issue

- integrate new workers efficiently - work stealing is nice here

- deal with disappearing workers - either design a scheme to preserve global knowledge or be able to deal with loss of information - in our case we deal with loss of information - two parts: loosing information stored in the fringe of a terminated node and loosing messages of dying workers - loosing information stored in the fringes: - multiple options: - send back nothing, loose parts of the search space - send back the root, redo parts of the search (also, loop detection!) - send back everything, takes much communication

- our loop detection scheme already implies that we loose parts of the search space and we have measures in place to deal with this fact (restarts) - for this reason we go with this approach

- loosing information due to lost messages: - Mallob has a mechanism to return messages - however, messages can still be lost (return message and original sender dies in the meantime) - we could change Mallob to send such messages to the root worker - however, this would turn the root into a bottleneck - again, we have mechanisms in place to deal with overall loss of information without loosing completeness - at the same time, we need to adapt our handling of return messages to ensure all workers stay in a valid state and do not get stuck

- getting wrong information (worker dies, is replaced, gets message meant for old worker)

5.1 Rest of the Stuff

- as mentioned by [27], upgrading a moldable to a malleable job is easy for portfolio solvers where a number of parallel but independent search strategies is involved - [27], the application itself is responsible to handle the removal/ addition of workers

- upgrade from moldable to malleable task according to [16] as introduced in section 2.5 - we can no longer assume that messages always arrive and get a response - we want to achieve completeness - when sending a message, we always want to send it to another As discussed in section 2.6, CrowdHTN in its original form is a moldable program, with the number of parallel workers fixed for each single execution. To adapt CrowdHTN to work as a malleable program within Mallob, we have to address three key concerns.

(i) Messages sent to workers who are suspended or terminated before it arrives

(ii) Integrating new processes to work efficiently

(iii) Dealing with workers dying without loosing completeness

To help understanding, this section explicitly discerns between Mallob workers and Crowd workers.

termi-
y

Messages sent to dead workers

t intro
e split
stays
' and
replace'

Receiving worker stays dead As Mallob runs each worker in a different processes communicating via message passing, we never have a full view of the current global state of our process, i.e., which other Mallob workers are assigned to the same job. Instead we only ever get information about this that might be outdated. As a result, when a message such as a work request is sent to another worker, we cannot be sure that the receiving worker is in a position to actually respond. Luckily, Mallob does provide us with a mechanism to detect such messages. Each Mallob worker knows which job it currently works on and all messages are tagged with their job as well. If a message belonging to job J_i is received by a worker working on job $J_k, k \neq i$, the message is simply tagged with a *returned to sender* flag and sent back. Now we can simply adapt CrowdHTN to deal with each message both if it is received normally and if it is received as a return message. On normal messages, nothing changes. On return messages we do the following:

- **Work request**: we treat this like a negative response. This means we set the *has active work request* flag to false and simply send out another work request at the next chance.
- **Positive work response**: this is the node we sent out ourselves. As to not loose any information, we simply re-insert the node at the front of our local fringe. While this may slightly change the order of the nodes at the front, we expect the number of outgoing work packages at any point to be small and the disturbance to be limited. In case of random searches, it should not negatively affect our algorithm.
- **Negative work response**: we do not have to care about this and can simply ignore it.
- **Ack**: we do not have to care about this and can simply ignore it.

While this mechanism should deal with most cases of workers dying, we can always construct pathological cases in which the return message mechanism fails. One such exchange can be seen in graphic 3. One way of dealing with this would be to change the way Mallob handles messages whose receiver is no longer valid. E.g., if a return message was delivered to a worker who changed jobs in the meantime, one could instead find out the root worker of the accompanying job and send the message there instead. This way no information would be lost until we decide to terminate the whole job at which point this would be unavoidable and okay by the user.

hat the
always
at least
ng as
thing

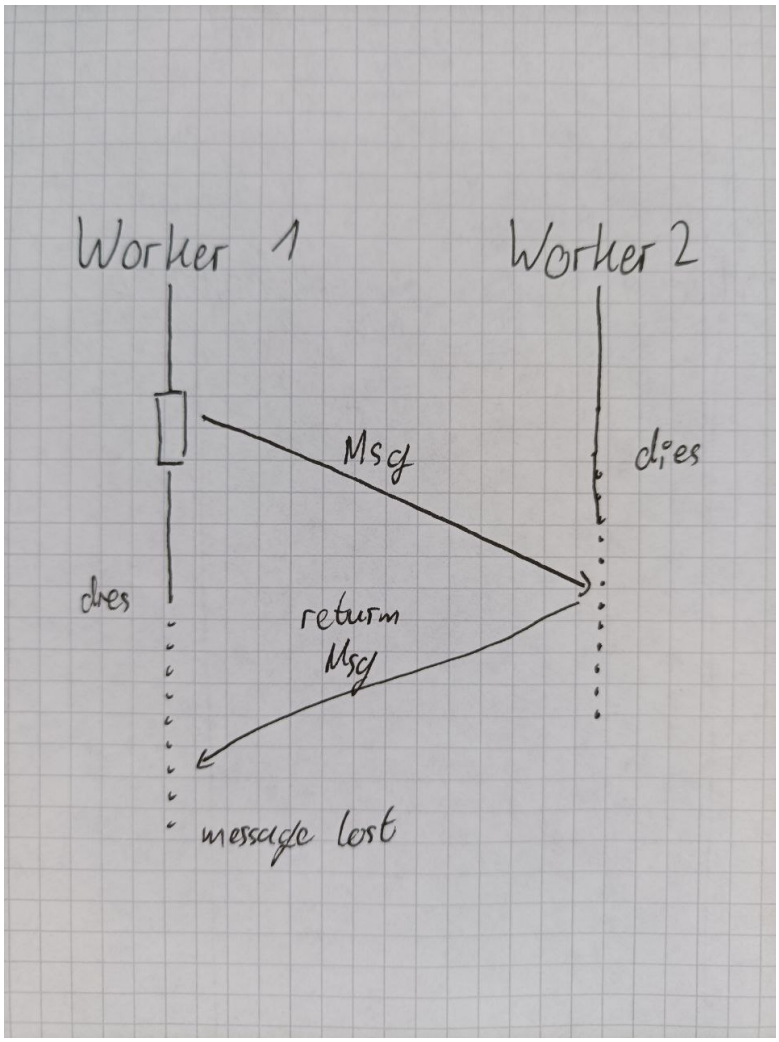


Figure 3: Example diagram of a lost message

To avoid having to change Mallob, we instead elected to make our adaptation of CrowdHTN resistant to such information loss.

write ab loop det tion and restarts

Receiving worker gets replaced by a new worker - we have no way to detect this case - some messages we can absorb - a lone ack: we don't care, instead of decreasing the number of outgoing packages we set the count to max(0, count -1) - a work request: can be responded to as if it was meant for us - work responses pose a problem: a node may send out a work request, die, a new node starts, starts it's own work request and receives the response afterwards - dafuq, this is complicated - buuut, it can happen - what if we just accept it? Worst case we get two positive responses, take both of them and deal with it - number can decrease again if two negative responses arrive before sending off a new request - cycle will interrupt anyways upon the first positive response (yay!) - everything is fine dot jay pee gee

are mes sages re ceived b suspend jobs?

Integrating new workers As CrowdHTN uses work stealing to distribute work and perform load balancing, new workers can be seamlessly integrated into a running job. Upon construction all workers are treated the same, whether they are part of the initial batch or appear at a later point. The root worker is initialized with the initial search node, all other nodes are empty. To get new work, a worker sends a work request to a random other node and then goes from there. The very same process can be used to integrate a new worker.

ob sim-
olves
ame
n ev-
ode,
lelism
vial,
eed to
y about
nfor-
on

Dealing with the information loss of dying workers As mentioned before, dying workers can lead to a loss of information. A previous paragraph discussed what this means for messages sent between workers. There is a second part of dying workers that we need to adjust to - the loss of the local fringe. There are three main ways in which we could deal with this loss.

- Communicate the local fringe to the parent or root worker
- Communicate the root search node of the local search to the parent or root worker
- Accept the loss of information and thus of parts of our search space

We will discuss these strategies one by one.

Communicate the local fringe - cleanest solution in a way - no information is ever lost - problems: - time needed for encoding the fringe is unbounded - an efficient encoding of the shared open tasks and world state would complicate the implementation - we need to ensure that the message containing our local fringe is not lost - i.e. it has to be sent to the root worker as it is guaranteed to live as long as anyone else - this imposes additional memory pressure on our root if many nodes terminate

Communicate the root back - this is guaranteed to not loose any information - however, we will have to re-do some work - communication of only a single node is rather easy, we already routinely do this - the pressure on the root worker is not too bad either, we could even re-integrate the node into the local fringe without much trouble - poses question regarding global loop detection - to not loose any part of our search space, we have to remove any node from global loop detection which originated from the dying worker, i.e. remove conservatively to not loose completeness - i.e., we can use this design efficiently but will have to re-do a lot of work

Accept the information loss - the simplest thing to do - we do not disturb the randomness of our work stealing by imposing additional work onto the root worker - performance is no problem at all - however, we loose parts of the search space - in combination with distributed bloom filters this may be the best solution either way - we simply wait for a future restart in which we do not loose the specific work

6 Implementation

- Mallob interface (Job) - all methods need to return fast - have a separate worker thread - where possible communicate via atomic bools, not locking

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 on how we designed malleable CrowdHTN, we choose to communicate an instance by simply transferring the 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. As [30] 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

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

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 4.2.4 CrowdHTN retains completeness even in this case.

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 4.2 and further allow our DFS based planner to achieve completeness as explained in section 4.3. 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 4.2.4 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 4.2.4 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 Adapting CrowdHTN for Use in Mallob

- CrowdHTN is no longer a standalone process - one way would be to launch CrowdHTN as a new process from within Mallob - instead re-architect CrowdHTN to prepare it as a library for external use - the abstractions of CrowdWorker and CooperativeCrowdWorker - the TohtnMultiJob (representing a crowd worker) contains a thread which executes CrowdHTN internally - CrowdHTN no longer worries about how messages are sent and received (completely agnostic to the mechanism) - on the Mallob side, messages are received and written to a buffer protected by mutex - the crowd thread takes all available messages and forwards them to CrowdHTN - similarly, Crowd generates messages, hands them to Mallob and stops caring

k name!

- communication between Mallob and CrowdHTN happens via atomic boolean flags (except for the mutex'd message buffers) - this ensures that all Mallob Job functions respond fast - even initialization of CrowdHTN (parsing, starting the work thread) happens within it's own thread to return fast

7 Improvements to CrowdHTN

7.1 Reducing Memory Consumption

7.1.1 Efficiently Storing the Preceding Plan

- no longer use a vector of USignature - instead use an ImmutableStack<USignature> - this way we can go from linear overhead to constant memory per search node! - actually, with A star or loop detection via hashset it gets like, really complicated - argh - still, lots of sharing! - lets protocol the thing (memory logging overall)

7.1.2 Reducing Copies of the World State

- if the next open task is an action, there is only a single possible way to resolve it - in other words, a search node whose next open task is an action always has zero or one children - this is not a very interesting node, either the preconditions hold or they do not - such nodes are no longer represented explicitly - instead, apply all actions from the sequence of open task up until the first abstract task is encountered - then also resolve that abstract task

Add gra
showing
many fe
nodes a
represe

7.1.3 Only Saving 'Potentially Interesting' Nodes

7.2 Efficiently Hashing Nodes of the Search Graph

Time is
actually
constan
world st
-> what
the exp
time?

Overview:

- the hash of a node consists of two parts
- 1: the hash of the open tasks
- 2: the hash of the world state
- We do not care about the hash of the preceding plan. Nodes with open tasks and world state equivalent have equivalent plans leading to a goal (somewhat similar to the Nerode relation) and we do not care about optimality. How he reach this point with equivalent remaining options is thus not of interest to us
- The hash of the world state can be large, but the number of elements is ultimately bound for any particular instance
- The length of the preceding plan is unbounded

Open tasks:

- Care needs to be taken to use an order independent hash function for the world state, as the underlying representation as a set does not guarantee us any particular iteration order, especially between nodes (might depend on the order of things inserted into the set!)
- Alternatively we could have chosen some other ordered representation, e.g. maintain the world state as a sorted list of predicates with a defined order. This would imply extra work we are unwilling to do.
- For open tasks, we save not only the task itself but additionally the hash of all open tasks from first to current one
- The order in which we hash open tasks is from oldest to newest one
- On applying a Reduction, we push the new open tasks onto the open task stack and compute each of their hashes combined with their predecessors. Each of these hashing operations is completed in O(1)

- Effectively, this means that each task is hashed exactly once
- As we already have to push each task onto the open tasks, inserting an additional O(1) operation on pushing does not change the asymptotic runtime

World state:

- In section XXX we discussed how each copy of the world state can be shared by many search nodes to reduce the memory footprint
- Instead of hashing the world state each time on demand, we can store a shared tuple of world state and it's hash
- This way, we only hash the world state once, reusing the computation
- Is this actually a time saving?
- Future work: the hash of the world state is order independent (sum of squares of individual hashes), to not worry about forcing any fixed iteration order. Utilize this and wrapping maths to hash only the differential of

7.3 Preceding Plan

In the initial implementation each node stored the full preceding plan as a sequence of all reductions that were applied so far. This leads to roughly quadratic overhead (sum over 1..n, only roughly as not each step increases the length. Wait, is it roughly, then? Probably, as the fraction of steps that increment the preceding plan should be kinda constant) This duplication was not needed. The newer implementation instead stores an optional<reduction> in each node. I.e., the preceding reduction is stored if one exists, nothing if there isn't one. When the preceding plan is needed, either for communication or to extract a plan, the current search path is iterated and all reductions are accumulated.

7.4 Lazy Instantiation of Child Nodes

lazy instantiation works on the basis of finding all free variables of a method and creating Reductions based on all possible combinations

Initial implementation: instantiate all possible reductions, filter out any with not fulfilled preconditions,then shuffle them

Problems: this spends both time and memory instantiating reductions that might never be needed for the rest of the search We effectively save not only the current path, but also follow all possible branches to a distance of 1

Solution: lazily create reductions as needed How to do this (first way): adapt the argiterator from Lilotane into the CrowdHTN code base. Adapt it to only substitute the arguments that are not already determined by the corresponding task (arguments)

To achieve randomization: each domain is iterated to create the substitutions Each time we build such an iterator, we randomize the order within the domains for this specific iterator This will lead to different orders

Further ideas: each time domains 1..k have been fully iterated, increment k+1 by 1, then shuffle the order of domains 1..k

For n total domains, each time domains 1..n-1 have been iterated, remove the current value from domain n, then shuffle all domain orders

Compare the runtimes of eager and lazy instantiation, check at which point it is worth it to incur the (potential) additional overhead of lazy instantiation Compare on multiple domains?

Check different metrics for comparison (size of domains, number of parameters, number of potential children (product of sizes))

Potential problems: We need quite a bit of state (domains, current index into each domain) to perform lazy instantiation The order is not truly random. We iterate some domains faster/ more often than others. What if the important change is in a domain which is iterated slowly? More random order makes this easier

A potential solution: space filling curves Advantages: little state (can just be incremented), iterates all dimensions equally Disadvantage: fixed order. With shuffling within each domain might be random again

Space filling curves come with the restriction of being strictly continuous We do not need this property. All we are interested in is an easy to compute fixed order in which the whole space is iterated where each permutation is hit exactly once We want only self-avoiding curves, to not hit any instantiation twice (could loop detection just fix that? But it would be a bad fix)

7.5 Using Domain Meta-Information

inspiration taken from HyperTensioN

taking preconditions of tasks and lifting them up into methods Only do this if the precondition is guaranteed to also apply to the method (cannot be achieved before the respective task?) This allows us to stop exploring paths earlier

7.5.1 Restarts Under Loop Detection

- bloom filters are of fixed size. If we propose a maximum false-positive rate, they will fill up at some point - this necessitates the need for restarts - bloom filter architecture: for local loop detection, use an expanding bloom filter (citation!) - this allows us to not incur any restarts due to local state - for global loop detection, we use a fixed-size bloom filter - if a node is encountered twice for local loop detection, add it to the global loop detector, to communicate

- we know that in mallob, the tree of workers is always a 'full' tree, except for the lowest level - i.e. we may be missing some nodes to the right end of the lowest level - especially, the root node always survives - as a result, the root node always receives all the messages about shared loop detection data - i.e., if the global loop detection bloom filter on any node is full, it follows that the global loop detection bloom filter on the root node is also full - the opposite does not hold (i.e. a new node enters just as the root node bloom filter fills up) - as a result, we can simply delegate the question of whether to restart to the root node and have it done in a centralized way, simplifying our communication patterns

7.5.2 Reaching Consensus on Search Version

- each search version corresponds to a restart with subsequent doubling in size of the global loop detector - keeping versions in sync is of importance - we do not want to loose any work - we do not want to insert search nodes from a wrong version into our bloom filter (especially old node into new filter, other way around is discarded either way) - each message between workers is extended by their internal version - if the version of a received message is higher than the internal version, the internal version is increased and then acted accordingly - new workers get updated to the current version the first time they ask for a work package - work requests and responses do not suffice as a version updating mechanism, as work packages may be arbitrarily large and messages arbitrarily rare -

7.5.3 Completeness Under Loop Detection

- any single iteration may be incomplete, e.g. if initial node and goal node hash to the same values - with restarts, we have uniform hashes, compute the chance of colliding each time - this necessitates changing seeds with each restart - soon, we will not collide - yay, plan can be found!

- we do loose termination - so far, termination could be known if the search tree gets too deep (cite limits known from other TOHTN paper) - however, this is infeasible, as RAM is actually finite (refer to known branching factors!) - i.e., we do not know the hash of a goal node, as a result we cannot exclude the possibility of it simply being cut off each time so far - we also do not know the hash of all nodes just before a goal node, etc (the path could always only be a single node wide) - as a result, we cannot terminate at any point, as we cannot exclude the plan hiding in some always-cut-off part of the search space - loosing termination is not a big deal anyways, I guess?

8 Experimental Evaluation

- describe i135 - 900 seconds per instance

- putting numbers to memory savings and how many nodes we save from instantiation (also saves our loop detection!) - show scaling on a nice instance - hash set vs bloom filter (both after optimizations!) - comparison of search algorithms - show just how bad BFS is? (branching factor + minimum layer through Lilotane) - test the probabilistic restarts - with 900 seconds we expect $\sum_{t=1}^{899} \approx 7.38$ restarts per run

- test the global loop detection - test the malleability

9 Future Work

9.1 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

9.2 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

9.3 Advances runtime pruning techniques

In section 4.3 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 [20], [18]. 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.

9.4 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

9.5 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

9.6 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

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

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

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.