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 September 5, 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.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Introduction

1.1 Motivation

1.2 Research Goal

- Provide a performant parallel TOHTN planner by improving upon the Crowd planner - Provide integration of TOHTN into the Mallob malleable load balancer - Compare performance of parallel to malleable TOHTN planning -

1.3 Thesis Overview

2 Theoretical Foundations

2.1.1

2.1 TOHTN Formalism

In this section we first define what HTN and TOHTN problems are from a formal perspective 2.1.1. Afterwards we take a short look at the algorithmic worst case complexity of HTN and TOHTN planning 2.1.2.

2.1.1 Defining TOHTN Planning Problems

e more alisms

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

Multiple definitions for HTN planning exist. In this work we build on the definition introduced in [4].

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 careterms 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****.*

link dire
to resolu
on of a
network
resolvin
defined
fore met

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.

extend
nition: r
ving a t
removes
from the
task net
work!

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.

2.1.2 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 [2] and [3] 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 PSPACE-hard.
Regarding the general relationship of hierarchical planning to complexity theory, [2] and [3] showed early on that HTN instances can be used to simulate context-free languages. This was extended by [6] who showed that TOHTN instances correspond exactly to context-free grammars.
In addition to planning itself, the problem of plan verification was studied. Here, [1] 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.

ide a
on for
nax.
n of a
net-
until
n is
l or
not

- the depth of our task network (until a plan is found) can be exponential in the input size - plan length is up to exponential in depth

2.1.3 Differences from other Kinds of Planning

a place
e this
Is con-
d in
or
ke's
s

[9] 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.
[7] argue that HTN-planning is not simply a domain-configurable version of classical planning and argue on the basis that [2, 3] showed that HTN-planning is strictly more powerful compared to classical planning which is PSPACE-complete.
While we agree with [7], one can still use HTN planning without using the full complexity of the model, using it instead to provide more efficient and guided versions of classical planning problems.

2.2 Techniques to solve TOHTN planning problems

- SAT-based - search-based - lifted vs grounded

2.3 Malleable Algorithms

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

4 Implementation

5 Improvements to CrowdHTN

5.1 Reducing Memory Consumption

5.1.1 Efficiently Storing the Preceding Plan

- Use the ImmutableStack<T> structure
-

5.1.2 Reducing Copies of the World State

5.1.3 Only Saving 'Potentially Interesting' Nodes

5.2 Making Crowd More CPU Efficient

5.3 Efficiently Hashing Nodes of the Search Graph

is not
lly
ant for
l state
hat is
xpected
?

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:

s not
emen-
et,
care
tually

- 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

section,
to it

Only pu
node int
loop det
tion as
explored
we can
ways ke
explorin
the loca
de (the
we are
liding w
may be
bitrarily
back in
fringe)

5.4 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.

5.5 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/

Evaluat
how ma
ny world
state ha
we actu
compute
both ca
Hashing
world st
of a noc
which is
timately
explored
(and wh
is other
not need
is waste
Number
computa
ons on l
hashing
number
nodes c
ted - nu
ber of n
remaini
in fringe
plan fou

Section
longer v

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)

5.6 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

5.7 Global and Memory Efficient Loop Detection

So far: each worker has a hash set of each node ever encountered (note: we define node equality through the sequence of open tasks and the set of predicates that is the world state. Depth in the graph and preceding plan are ignored) Advantages: allows for perfect local loop detection, as we can always fall back to equality checks in case of hash collisions. Problems: This approach leads to a massive memory overhead, as the world state is duplicated countless times. This also leads to increased run times just to manage the growing hash set. This is also not suited for global loop detection. Global loop detection would need some mechanism to communicate seen nodes. Nodes are too large to communicate all of them, though. If we only communicate some nodes instead of all we run the risk of a node "going pastthat barrier of known nodes and still exploring the swathes of known nodes beyond the "barrierTODO: what about open tasks (memory consumption)? Or was that an ImmutableStack with little overhead?

Idea: drop the precondition of perfect loop detection with no false positives. For loop detection use a bloom filter with a set of hash functions for search nodes. This comes with a fixed, configurable memory overhead per worker. As a result, overall memory consumption should - drop - be more predictable To communicate nodes we can just communicate the bit array that makes up our filter and combine them via bitwise OR. This ensures that communication volume is also fixed. It is independent of both size of nodes (i.e. length of open tasks sequence) and number of explored nodes (since last round of communication). In addition to communicating the bloom filter we communicate the number of newly added nodes. Then each worker knows an upper bound for the total number of nodes that are part of it's bloom filter (might be an overestimation if two workers insert identical nodes as the node would be counted twice). As bloom filter performance/ false positives degrade with the number of elements inside it, we can restart our work depending on the number of elements in the bloom filter.

- loop detection information is shared at most every second (or however long it takes for the communication to complete) - as a result, when the root node increases it's version, a broadcast is started that communicates the new version to everyone else - the version in normal messages is still kept. In case a message arrives before the version broadcast, the version is increased even earlier. New nodes likely get their version with the first work package.

5.7.1 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

5.7.2 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 -

5.7.3 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?

6 Theoretical Improvements to the Crowd Planner

6.1 Search Algorithms in Crowd

6.2 Loop Detection

This section will discuss loop detection as it is used in (TO)HTN planning in general. It will start with an overview over loop detection in other HTN planners in section 6.2.1. This is followed by a discussion of the simplifying assumptions we can make for TOHTN planning (section **??** - distributed loop detection (communication and merge operations become important!) - perfect loop detection - probabilistic loop detection (approximate membership query)

- [8] domains can have introduce infinite loops, without some kind of loop detection we loose completeness (and performance)

Make su to alway talk abo isomor- phic tas network instead equivale Quote [Define i morphis

6.2.1 Loop Detection in Other HTN Planners

Loop detection in HTN planning is a recent phenomenon and was introduced in 2020 by the HyperTensioN planner ([8]) 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 [5] and means that HyperTensioN is not complete. [5] 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.

quote al loop det on in gr search?

quote al distribu loop det on in gr search?

In response to [8], loop detection was introduced to the PANDA planner in [5]. 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.

6.2.2 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.

refer to impleme tation p where th open ta are repr sented a sequenc

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.

Refer to the imp

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 full states would have to be communicated. If we perform loop detection only locally, we predict 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 chance that those two branches are encountered on different processors is higher the more processors we have.

6.2.3 Perfect Loop Detection

One simple way to perform loop detection which is similarly used in PANDA ([5]) 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 . Additionally, we have to keep both s and tn around for all nodes ever encountered, increasing the memory footprint of our program. In case of distributed loop detection, communicating full states would lead to a large communication overhead. Especially, we can expect each node to be larger than those sent as work packages, as those are optimized to have a small tn which would not hold for nodes encountered in our loop detection.

e about
ex-
ntial
n!

6.2.4 Bloom Filters

e about
a quo-
filter
the
ion! (or
ast why
s not
n)

Quotient filter: - do we need the original elements to re-insert them? - do we need to communicate whole hashes to combine filters? (worse communication size!)

Advantages: - communication takes less effort (a lot lower size of data!) - hashing in O(1)? Or at least in a lot easier - we can pre-hash the open tasks (can be of exponential depth (requires a good argument, as the path down the task network could be of width 1?)), thus turning hashing into O(1) - in case of hash collisions we need to walk the whole sequence of open tasks - in practice this is even worse, as our open tasks are saved in a tree-like manner to allow sharing of the tasks between search nodes - this means we wildly jump through memory for hashing, adding another constant factor - we could save on this constant factor by duplicating the open tasks for each node and saving them sequentially, but leading to a lot higher memory footprint (might be quadratic compared to what it is already (if any fixed fraction of nodes have at least 2 children, maybe?)) - compromise between performance and false-positive rate (better for correctness than just comparing a single hash)

e
s

math

6.2.5 Distributed Loop Detection

- two new considerations - memory footprint becomes more important for communication (effectively an all-to-all operation -> quote Sanders' book!) - efficient merge of loop detection data is needed - communicating everything might still be inefficient for bloom filters ()

7 Malleable TOHTN Planning

- ensure completeness - ensure performance - work stealing is kinda nice here

8 Grounding and Pruning

[6] - a ground HTN instance has (worst-case) a size that is exponential in the arity of task names and predicates

9 Experimental Evaluation

10 Future Work

Literatur

[1] Behnke, Gregor, Daniel Höller und 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*, Band 25, Seiten 25–33, 2015.

[2] Erol, Kutluhan, James Hendler und Dana S Nau: *HTN planning: Complexity and expressivity.* In: *AAAI*, Band 94, Seiten 1123–1128, 1994.

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

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

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

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

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

[8] Magnaguagno, Maurício C, Felipe Rech Meneguzzi und 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.

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