ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie

Institut für
Künstliche Intelligenz

Approaching Acyclic HTN Planning through Specialized Heuristics and a Translation to Classical Planning

Masterarbeit an der Universität Ulm

Vorgelegt von
Linus Diepold

Gutachter:
Prof. Dr. Birte Glimm
Dr. Pascal Bercher

Betreuer:
Cornelia Olz

2024

Fassung July 8, 2024

Abstract

HTN planning in its most general form is undecidable, because of its high expressiveness and complex decomposition structures. A lot of HTN sub-classes have a much lower complexity due to imposing restrictions to the formalism. A very powerful restriction to a decomposition structure is the prohibition of cyclic relations. Without the possibility of infinite loops, a finite amount of methods can be used resulting in task networks that are limited in size. This thesis takes advantage of said property, presenting algorithms to calculate decomposition limits for methods and tasks imposed by acyclic decomposition structure. These limits among other adaptations will be used to improve the performance of current HTN planners to better deal with acyclic HTN problems. Specifically adaptations to the Delete- and Ordering-Relaxation Heuristics of Höller et al. [2020a] are presented. Total ordered acyclic HTN planning represents a even further relaxed formalism matching the expressiveness and complexity of classical planning, both being PSPACE-complete. In order to enable the use of classical planning techniques, this thesis contributes a translation of acyclic TO HTN problems to classical planning problems.

Contents

1 Introduction

Planning in the context of artificial intelligence is the process of selecting and organizing tasks, that achieve a specific goal when executed. Planning has many fields of application such as robotics, manufacturability analysis and game AI (Ghallab et al. [2004]). Especially the field of HTN planning has come to great prominence in the last couple of years (Bercher et al. [2019]). The most distinctive feature of hierarchical planning is its decomposition structure. Breaking down very complex high-level tasks till they become primitive actions seems intuitive since it mirrors human problem solving techniques (Erol et al. [1994]). Yet through recursion it can lead to highly complex problems creating infinite loops and deadlocks. Efforts were made to reduce the complexity of HTN planning, by imposing restrictions (Alford et al. [2014], Nau et al. [1999]). Creating sub-classes of HTN planning in the process, e.g. HTN planning with task insertion as proposed by Geier and Bercher [2011]. Different sub-classes of HTN planning can vary strongly in complexity (Alford et al. [2015]). These sub-classes can be used to design heuristics to assist planning for unrestricted problems (Höller et al. [2020a]).

One of the most restricted sub-classes of HTN planning is acyclic HTN planning. In acyclic HTN planning, cyclic relations inside the decomposition structure are prohibited. This removes the possibility of self-sustaining decomposition cycles. Task networks of acyclic HTN problems are therefore limited in size. This property enables the calculation of decomposition limits for each task and method. Such limits describe how often a task/method can occur/be applied in a given task network. These limits can be calculated through analysing the decomposition tree of a given acyclic HTN problem. A decomposition tree can be traversed from its roots to its leaves or vice versa, resulting in a Top-Down and a Bottom-Up approach. This thesis proposes distinct algorithms for both approaches.

Most planning systems like the PANDA-Planing system (Höller et al. [2021]) are equipped with a multiple heuristics and planning strategies. When dealing with acyclic HTN problems, most planning strategies and heuristics do not take full advantage of their restricted decomposition structure. These adaptations may prove useful since acyclic HTN problems are easy to solve but not uncommon. For example, the International Planing Competition (IPC) of 2020 covered total ordered HTN problems over 24 domains, from which 8 included acyclic HTN problems. Such adaptations might also be useful when dealing with cyclic HTN problems, since parts of their decomposition structure might be acyclic as well and can therefore also benefit from acyclic HTN planning techniques.

This thesis explores specialized adaptations for dealing with acyclic HTN problems on the example of Delete- and Ordering-Relaxation Heuristics (DOR-Heuristics) introduced by Höller et al. [2020a]. The DOR-Heuristics are based on a set of constraints which will be used to compute the relaxed problems in a IP/LP model. Some constraints prevent the usage of self sustaining cycles, which can be removed when dealing with acyclic HTN problems. Also constraints can be added limiting the use of tasks and methods, using the algorithms mentioned above. This thesis will explore whether and how those adaptations will improve the performance of DOR-heuristics for acyclic HTN planning.

Classical planning, in contrast to HTN planning, is a way older field reaching back to the work of Fikes and Nilsson [1971]. Since then a lot of different approaches and heuristics have been researched (Shleyfman et al. [2015], Pommerening et al. [2017]). With that research a lot of very sophisticated solving techniques have emerged (Helmert [2006], Fickert et al. [2018]). There has been some research already to deploy classical planning techniques to HTN planning like Höller et al. [2018] or Höller et al. [2019]. Also some research has been done translating HTN problems into classical problems (Alford et al. [2016], Höller [2021] and Behnke et al. [2022]). These translation are very appealing since they convert a complex and highly expressive formalism into a simple and efficient to solve formalism. These formalism often use bounds to limit the size of task networks in the HTN problem. Acyclic TO HTN problems are limited in size and are PSPACE-complete just as classical planning problems. Therefore a direct translation without any loss of expressiveness is possible. This thesis proposes an encoding that achieves such a translation.

2 Formal Framework

2.1 Classical Planning

In this section a formalism for classical planning named STRIPS will be presented. The formalism was first introduced by Fikes and Nilsson Fikes and Nilsson [1971].

Definition STRIPS Planning Problem
A *STRIPS planning problem* is defined by the tuple (L, A, s_0, g, δ).

- L is a set of propositional state features
- A is a set of action names
- $s_0 \in 2^L$ is the initial state
- $g \subseteq L$ is the goal state
- δ is a set of functions $\{prec, add, del\}$ with $A \to 2^L$

The set L consists of propositional variables used to describe states. A state $s \in 2^L$ is defined by the state features that hold in it. The execution of an action can cause a state transition $s \to s'$. An action a is applicable if and only if $s \supseteq prec(a)$. If a is applicable in state s its application will cause a state transition $s \to s'$ with $s' = (s \backslash del(a)) \cup add(a)$.

Definition Total-Order Solution
A sequence of actions $\langle a_1, \ldots a_n \rangle$ is a solution to a STRIPS planning problem if and only if:

- a_1 is applicable in s_1 causing a state transition $s_1 \to s_2$
- for every consequent s_i the action a_i is applicable with $1 \leq i \leq n$
- all state features that are true in the goal state have to be true in s_{n+1} i.e. $s_n \subseteq g$

2.2 Hierarchical Task Network

In this section a HTN planning formalism will be presented. The following formalism was first introduced by Geier and Bercher Geier and Bercher [2011].

Definition: Hierarchical Planning Problem P
A *Planning Problem* P is defined by the tuple $(L, C, A, M, tn_I, s_I, g, \delta)$.

- L is a set of propositional state features
- C is a set of compound tasks
- A is a set of primitive actions
- M is a set of methods
- tn_I is the initial task network
- s_I is the initial state
- g is the goal state
- δ is set of functions $\{prec, add, del\}$ with $A \to 2^L$

(L, A, s_0, g, δ) are similarly defined as in the Section 2.1. And similarly to classical planning a sequence of actions is being searched that causes a state transition from s_I to g. Instead of just inserting tasks into such a sequence in HTN planning such actions have to be decomposed from compound tasks inside of the initial task network tn_I using the methods in M. HTN planning is using a set of tasks $N = A \cup C$ consisting of two kinds of tasks: primitive actions A and compound tasks C. For this purpose a HTN planning problem features a set of methods M that can decompose compound tasks. Tasks are organized in task networks that also contain ordering constraints which defines an ordering of the contained tasks.

Definition: Task Network
A *task network* tn is defined by the tuple $(T, \prec, \alpha)$.

- T is a set of unique task identifiers
- $\prec \subseteq T \times T$ is a set of ordering constraints
- $\alpha : T \to N$ is function that maps unique task identifiers to tasks

Using a set of unique tasks identifiers is necessary since a task can occur multiple times in the same network. Generally speaking these constraints do not have to infer a total order on every task in T. A total order would define a fixed sequence of tasks. In contrast a partial order would only contain constraints to some tasks in T. Which results in a multiple possible sequences of tasks that could satisfy such constraints. In this work we will deal exclusively with total ordered task networks.

Definition: Method m
A *method* m is defined by tuple (c, tn_m).
It can decompose the compound task $c \in C$ inside of the task network tn into its subtasks.

Definition: Decomposition $tn_1 \xrightarrow{t,m} tn_2$
A task network $tn_1 = (T_1, \prec_1, \alpha_1)$ can be decomposed into $tn_2 = (T_2, \prec_2, \alpha_2)$ using method $m = (c, tn_m)$ if and only if there is $t \in T_1$ and $\alpha_1(t) = c$ and a task network $tn' = (T', \prec', \alpha')$ that is equal to tn_m but without containing task ids of tn_1(i.e. $T_1 \cap T' = \emptyset$.

$$\begin{aligned} tn_2 =& ((T_1 \setminus \{t\}) \cup T', \prec_1 \cup \prec' \cup \prec_D, \alpha_1 \cup \alpha') \\ \prec_D =& \{(t_1, t_2) | (t_1, t) \in \prec_1\} \cup \\ & \{(t_1, t_2) | (t, t_2) \in \prec_1\} \end{aligned}$$

Definition: Solution Network tn_{sol}
A *Task Network* $tn_{sol} = (T_{sol}, \prec_{sol}, \alpha_{sol})$ is a solution to a *Planning Problem* P if and only if:

- tn_{sol} can be obtained by a sequence of decompositions of tn_I
- all task id's $t \in T_{sol}$ are mapped to primitive actions i.e. $\alpha_{sol}(t) \in A$
- there is a sequence of tasks id's $\langle t_1 \dots t_n \rangle$ that satisfy all ordering constraints in $\prec_{sol}$, with $\langle \alpha_{sol}(t_1) \dots \alpha_{sol}(t_n) \rangle$ being applicable in s_0 and resulting in g

Since all definitions, needed to describe and solve a HTN planning problem were established, decomposition trees will be introduced. The decomposition tree aids in formalizing the decomposition structure imposed by the initial task network tn_I, the compound tasks C and their methods M. The formalism that will be used, is analogue to the formalism of Geier and Bercher [2011] but simplified in regards to ordering constraints since they will not be needed for the algorithms in section 3.

Definition: Decomposition Tree g
A *Decomposition Tree* g is defined by the tuple (T, E, α, β).
(T, E) is a tree with T representing the set of Nodes and E representing the edges. Every node is labeled with a task name which is mapped by the function $\alpha : T \to N$ The nodes that are labeled by compound tasks are further labeled with methods by the function $\beta : T \to M$. Let $t \in T$ be a node that is labeled by an compound task $\alpha(t) = c$ and a method $\beta(t) = m$ with $m = (c, tn_m)$ and children in g $ch(g, t)$. The task network induced by $ch(g, t)$ is isomorphic to tn_m i.e. they differ only in the task ids.

3 Max-Task and Max-Method Algorithm

One of the properties that distinguishes acyclic HTN problems from cyclic HTN problems is the limited size of decomposable task networks. Which task can be inserted in a HTN plan is restricted to tasks that can be decomposed from existing compound tasks. This is in contrast to classical planning where tasks can be inserted arbitrarily (if their preconditions are satisfied). In acyclic HTN problems there is a further restriction: A compound task cannot decompose into itself (not even indirectly). This means the maximum occurrences of a task is restricted to the task in the initial task network and the limited decomposition capabilities of said task. The maximum occurrences of a task is further restricted by either their preconditions or the preconditions of their subtasks. However for the sake of simplicity, we deal in this chapter exclusively with the restrictions imposed by the decomposition structure. These maxima are still correct even if they might not be perfectly accurate. These maxima can be calculated by analysing the decomposition tree (as defined in Section 2.2) of the given acyclic HTN problem. In the sections below two calculation methods are presented. One that analyses the decomposition tree starting from its root progressing towards its leaves (top-down) and one that starts from its leaves towards its root (bottom-up). Having an efficient way to calculate maximum occurrences of tasks and methods, can assist in designing specialized heuristics for HTN planner (see Chapter 4). 5

3.1 Example: Transportation

In this section an acyclic HTN problem is presented to serve as an example for the following algorithms. This problem instance belongs to a domain of transportation problems and is adapted to be acyclic. As defined in Section 2.2 a HTN problem consists of a tuple $(L, C, A, M, tn_I, s_I, g, \delta)$.
The problem is about delivering two packages per truck to two different destinations. There are four locations in total, these locations are connected like shown in Figure 3.1. The locations are connected through one-way roads, which is a necessary condition to fulfill the property of acyclicity. The formal definition of said tuple can be found in the appendix A.

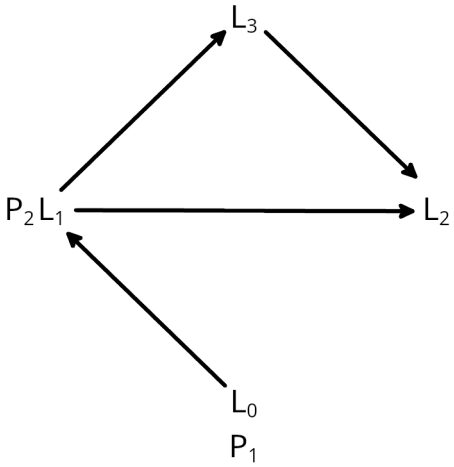

Figure 3.1: The road network to connect the locations in the problem described in appendix A P1 need to be delivered to L1 and P2 need to be delivered to L2

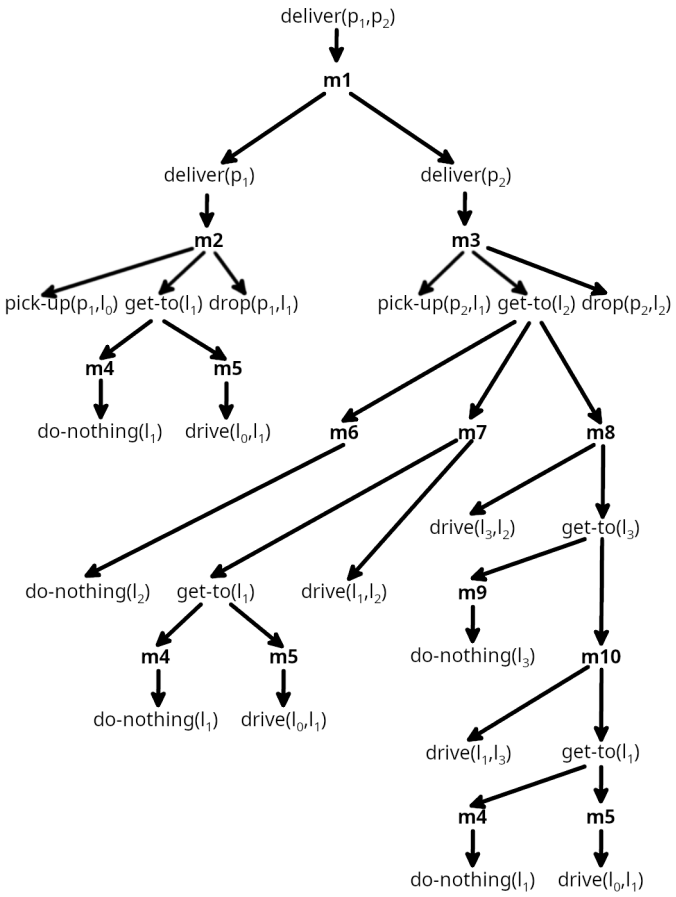

Figure 3.2: Decomposition tree of the transport problem described in appendix A

In the following the decomposition structure of the example visualized in Figure 3.2 will be analysed. The max-action/method algorithms will derive the maximum amount of decompositions of a given task/method. Although every primitive action listed above is unique, we will use the following algorithms to calculate the maximum amount of drive actions. For demonstration purposes we will ignore the variables attached (in this case the respective locations). As can be seen in Figure 3.2 the decomposition tree has six occurrences of drive actions. With the right choice of methods four of them can be reached simultaneously, which is also the maxima that will be calculated by the max-action algorithm. However this maxima does exclusively consider the decomposition structure. If the restriction of the state features would be considered as well the maxima would shrink to three drive actions. For example the action $drive(l_0, l_1)$ occurs three times in the decomposition tree but every solution of the underlying problem can contain said action only once, which becomes evident by analysing the δ functions: $prec(drive(l_0, l_1)) = del(drive(l_0, l_1)) = at(truck, l_0)$. Since there is no action a with $add(a) = at(truck, l_0)$ and $at(truck, l_0) \in s_I$ it can be concluded that $drive(l_0, l_1)$ can only be executed once.

3.2 Top-Down

3.2.1 Top-Down-Task Algorithm

```
   Data: tn_I − InitialNode, task_x ∈ A ∪ C
   Result: max_task_x
 1 max_task_x ← 0;
 2 leaves ← ∅ ;
   /* create a list of all tasks contained in the current node with a
      boolean to indicate if it got decomposed                          */
 3 for task ∈ tn_I do
 4 |   node ← (task, false);
 5 end
 6 fringe ← node;
 7 while fringe ≠ ∅ do
 8 |   curr ← fringe.pop;
 9 |   isLeaf ← true;
10 |   for (task, isDecomposed) ∈ curr do
11 |   |   if task ∈ C and isDecomposed ≠ false then
12 |   |   |   for method ∈ {m ∈ M|m decomposes task} do
13 |   |   |   |   isDecomposed ← true;
14 |   |   |   |   fringe ← curr ∪ (subtasks(method), false);
15 |   |   |   end
16 |   |   |   isLeaf ← false;
17 |   |   end
18 |   end
19 |   if isLeaf then
20 |   |   leaves ← curr;
21 |   end
22 end
23 for leaf ∈ leaves do
24 |   max_task_x ← max(occurences(leaf, task_x), max_task_x);
25 end
26 return max_task_x
```

Figure 3.3: Pseudocode for the Top-Down-Task algorithm

The Top-Down-Task algorithm as shown in Figure 3.3 calculates the maximum amount of decompositions of any task, by analysing the decomposition tree from root to the leaves. The root is the initial tasks network tn_I and leaves are the nodes containing primitive actions only. Every node in a decomposition tree does contain only a singular task or method. In order to calculate maximum occurrences it is important to also

capture every previous decomposed task. This is realized in the node data structure containing a list of all task previously decomposed and yet to decompose(indicated by a boolean variable). The node data structure used in the algorithm can be very memory intensive since its size grows exponentially with the depth of the decomposition tree. The algorithm starts with the initial node containing the tasks in tn_I and gradually decomposes every task, creating new nodes in the process. Every possible decomposition creates a new node containing all tasks of the parent node and all subtasks of the applied method. If a node contains only tasks that are tagged as decomposed or primitive the node will be declared as leaf node. If all leaves are calculated they can be compared to find the node with most occurrences of $task_x$, since all leaf nodes represent all combinations of possible decompositions in the given HTN problem. The amount of occurrences of $task_x$ in the node with the most occurrences is the maximum amount of decompositions of $task_x$. The list below depicts some of the leaf nodes when running the Top-Down-Task algorithm on the example of 3.1, drive Actions are highlighted and below every leaf the methods used are enlisted. The methods are depicted in the same sequence as they got chosen in line 12 of Figure 3.3.

$leaf_1 = \{deliver(p_1, p_2), deliver(p_1), deliver(p_2), pick-up(p_1, l_0), get-to(l_1),$
$drop(p_1, l_1), do-nothing(l_1), pick-up(p_2, l_0), get-to(l_2), drop(p_2, l_2), do-nothing(l_2)\}$

$methods_1 = \{m_1, m_2, m_4, m_3, m_6\}$

$leaf_2 = \{deliver(p_1, p_2), deliver(p_1), deliver(p_2), pick-up(p_1, l_0), get-to(l_1),$
$drop(p_1, l_1)$,$\mathbf{drive(l_0, l_1)}$,$pick-up(p_2, l_0), get-to(l_2), drop(p_2, l_2), get-to(l_1),$
$\mathbf{drive(l_1, l_2)}, do-nothing(l_1)\}$

$methods_2 = \{m_1, m_2, m_5, m_3, m_7, m_4\}$

$leaf_3 = \{deliver(p_1, p_2), deliver(p_1), deliver(p_2), pick-up(p_1, l_0), get-to(l_1),$
$drop(p_1, l_1)$,$\mathbf{drive(l_0, l_1)}$,$pick-up(p_2, l_0), get-to(l_2), drop(p_2, l_2), get-to(l_3),$
$\mathbf{drive(l_3, l_2)}, get-to(l_1)$,$\mathbf{drive(l_1, l_3)}, get-to(l_1), do-nothing(l_1)\}$

$methods_3 = \{m_1, m_2, m_5, m_3, m_8, m_{10}, m_4\}$

$leaf_4 = \{deliver(p_1, p_2), deliver(p_1), deliver(p_2), pick-up(p_1, l_0), get-to(l_1),$
$drop(p_1, l_1)$,$\mathbf{drive(l_0, l_1)}$,$pick-up(p_2, l_0), get-to(l_2), drop(p_2, l_2), get-to(l_3),$
$\mathbf{drive(l_3, l_2)}, get-to(l_1)$,$\mathbf{drive(l_1, l_3)}, get-to(l_1)$,$\mathbf{(drive(l_0, l_1)}\}$

$methods_4 = \{m_1, m_2, m_5, m_3, m_8, m_{10}, m_5\}$

These leaf nodes now can be compared, and it can be concluded that $leaf_4$ has the most occurrences drive actions(four) which is the maximum that will be calculated by the algorithm. This example further shows how the maxima refers to the decomposition

structure only, since the leaf nodes do not represent valid solutions to the HTN problem. While the primitive actions in the leaves two and three do represent a solution, $leaf_1$ does not satisfy the precondition of $drop(p_1, l_1)$ and $drop(p_2, l_2)$ and $leaf_4$ does not satisfy the preconditions of one of the $drive(l_0, l_1)$ actions.

3.2.2 Top-Down-Method Algorithm

Data: $tn_I - InitialNode,\ method_x \in M$
Result: max_{method_x}
$max_{method_x} \leftarrow 0$;
$leaves \leftarrow \emptyset$;
`// create a list of all decomposeable tasks and all used methods`
$fringe \leftarrow (\emptyset, tasks(tn_I))$;
while $fringe \neq \emptyset$ **do**
  $(currMethods, currTasks) \leftarrow fringe.pop$;
  if $currTasks = \emptyset$ **then**
    $leaves \leftarrow currTasks$;
  else
    for $task \in currTasks$ **do**
      $(newMethods, newTasks) \leftarrow (currMethods, currTasks)$;
      for $method \in \{m \in M | m\ decomposes\ task\}$ **do**
        $remove(newTasks, task)$;
        $newTasks \leftarrow abstractSubtasks(method)$;
        $newMethods \leftarrow (method)$;
        $fringe \leftarrow (newMethods, newTasks)$;
      end
    end
  end
end
for $leaf \in leaves$ **do**
  $max_{method_x} \leftarrow max(occurences(leaf, method_x), max_{method_x})$;
end
return max_{method_x}

Figure 3.4: Pseudocode for the Top-Down-Method algorithm

Analogous to the Top-Down-Task algorithm the Top-Down-Method algorithm does also iterate through all possible sequences of decompositions. These algorithms mainly differ in the way they keep track of their nodes. While the Top-Down-Task algorithm stores all the previously decomposed tasks in the nodes, the Top-Down-Method algorithm removes already decomposed tasks and stores the used methods instead. Therefore leaf nodes in

the Top-Down-Method algorithm consists of methods only. Below leaf nodes for the example of 3.1 are listed with highlighted occurrences of m_5. $leaf_1 = \{m_1, m_2, m_4, m_3, m_6\}$

$leaf_2 = \{m_1, m_2, \mathbf{m_5}, m_3, m_7, m_4\}$

$leaf_3 = \{m_1, m_2, \mathbf{m_5}, m_3, m_8, m_{10}, m_4\}$

$leaf_4 = \{m_1, m_2, \mathbf{m_5}, m_3, m_8, m_{10}, \mathbf{m_5}\}$

As can be seen $leaf_4$ has the most occurrences of m_5 which is also the maximum of two.

3.3 Bottom-Up

3.3.1 Bottom-Up-Task Algorithm

Data: $node - InitialNode$, $Task - task_x$
Result: $contribution, postCalcContribuiton$

```
1  max_task_x ← 0;
2  for task ∈ A ∪ C do
3  |  if task ∈ C then
   |  |  /* If the contribution of a task is non-primitive its
   |  |     contribution will be tagged with -1                    */
4  |  |  contribution[task] ← −1;
5  |  else
6  |  |  contribution[task] ← 0;
7  |  end
8  end
9  contribution[task_x] ← 1;
   /* Every task that can directly or indirectly produce task_x
      through decomposition will be queued up.  First component is
      the decomposition method, second component is the produced task
      */
10 producer ← (∅, task_x);
```

```
11 while producer ≠ ∅ do
12     (currMethod, currTask) ← producer.pop;
13     primitiveContribution ← true ;
14     if currMethod ≠ ∅ then
15         currProducer ← decomposedTask(currMethod);
16         newContribution ← 0;
17         for subtask ∈ subtasks(currMethod) do
18             if contribution[subtask] > −1 then
19                 newContribution += contribution[subtaks];
20             else
21                 newPostCalc ← subtask ;
22                 primitiveContribution ← false;
23             end
24         end
25         if primitiveContribution then
26             contribution[currProducer] ← max(contribution[currProducer], newContribution);
27         else
28             contribution[currProducer] ← −1;
29             postCalcContributions[currProducer] ← (newContribution, newPostCalc);
30         end
           /* producer(t) := set of tuples (m,t) where task_t can be
              produced as subtask of m                                  */
31         for (m, t) ∈ producer(currProducer) do
32             if ¬producer.contains((m, t)) then
33                 producer ← (m, t);
34             end
35         end
36     end
37 end
38 return contribution, postCalcContribuiton
```

Figure 3.5: Pseudocode for the Bottom-Up-Task algorithm

```
Data: contribution, postCalcContribuiton
Result: max_task_x
/* The following describes the post-calculation algorithm that
   calculates all non-primitive contributions                    */
1  postCalcFinished ← false;
2  while ¬postCalcFinished do
3  |  postCalcFinished ← true;
4  |  for task ∈ Tasks do
5  |  |  if postCalcContributions[task] ≠ ∅ then
6  |  |  |  postCalcFinished ← false;
7  |  |  else
8  |  |  |  contribution[task] ← 0 ;
9  |  |  end
10 |  |  currTaskFinished ← true for
   |  |   (currContribution, postCalc) ∈ postCalcContributions[task] do
11 |  |  |  if postCalc ≠ ∅ then
12 |  |  |  |  currTaskFinished ← false
13 |  |  |  end
14 |  |  |  for postCalcTask ∈ postCalc do
15 |  |  |  |  if contribution[postCalcTask] > −1 then
16 |  |  |  |  |  remove(postCalc, postCalcTask);
17 |  |  |  |  |  currContribution += contribution[postCalcTask];
18 |  |  |  |  end
19 |  |  |  end
20 |  |  end
21 |  |  if currTaskFinished then
22 |  |  |  for currContribution ∈ postCalcContributions[task] do
23 |  |  |  |  contribution[task] ← max(contribution[task], currContribution);
24 |  |  |  end
25 |  |  end
26 |  end
27 end
28 for task ∈ containedTask(tn_I) do
29 |  max_task_x += contribution[task];
30 end
31 return max_task_x
```

Figure 3.6: Pseudocode for the post-calculation of the Bottom-Up-Task algorithm

The Bottom-Up-Task algorithm as shown in Figure 3.5 calculates the maximum occurrences of $task_x$. Starting by analysing the tasks that can be decomposed into $task_x$ (i.e. direct producers). After that the tasks that can be decomposed into $task_x$ (i.e.

indirect producers) will be analysed. This process will be repeated until every task that might produce $task_x$ (through a series of decompositons) is analysed. When analysing a task its contribution to the maximum needs to be calculated. The contribution of a $task_y$, towards the maximum of $task_x$, is defined by the maximum amount of $task_x$ that can be produced when decomposing $task_y$. If $task_y$ can be decomposed into primitive actions or $task_x$ exclusively its contribution is considered primitive i.e. it can be expressed as an integer. In the example of section 3.1 the task $get - to(l_1)$ can produce $do - nothing(l_1)$(by applying m_4 or $drive(l_0, l_1)$(by applying m_5). When calculating the maximum of $task_x = drive(l_0, l_1)$ the contribution of $get - to(l_1)$ will be considered primitive and has the value one, since it can produce $task_x$ up to one time. The contribution of $task_x$ itself is defined as one. If the contribution of every task is known the maximum occurrences of $task_x$ can be calculated by: $\sum_{task \in tn_I} contribution[task]$. Contribution of tasks like $deliver(p_2)$ are considered non-primitive and can not be calculated right away. Two of the three methods that decompose $deliver(p_2)$ contain non-primitive subtasks. Those could be potentially further decomposed into an unknown amount of $task_x$. As long as the contribution of these non-primitive subtasks is unknown(e.g. $get - to(l_1)$ and $get - to(l_3)$) the contribution of $deliver(p_2)$ is also unknown. Therefore the ordering for calculating contributions becomes important. The calculation of contributions that can not be calculated right away will be postponed to the post-calculation part of the algorithm. For that purpose every non-primitive contribution will be saved as tuple of int(already known contribution) and set of tasks (subtasks with unknown contribution). Since non-primitive contributions can not be compared, a task can have multiple non-primitive contribution. For example $deliver(p_2)$ can have the non-primitive contributions for $drive$ of $(1, \{get - to(l_1)\})$ and $(1, \{get - to(l_3)\}$. The contributions of such tasks will be tagged with the value -1. In the post-calculation part every task that has no contribution assigned to it(neither primitive or non-primitive) is not related to $task_x$ in the decomposition tree and has therefore a contribution of 0. A non-primitive contribution in post-calculation will be either calculated from the new known contributions or it will be further postponed. This process will then be repeated till every contribution will be primitive and calculated. In the list below there are contributions of the compound tasks using the example of 3.1 with $task_x = drive$ before and after post-calculation.

Contributions after Calculation

- $contribution[drive] = 1$
- $contribution[get - to(l_1)] = 1$
- $contribution[get - to(l_2)] = \{(0, \emptyset), (1, \{get - to(l_1)\}), (1, \{get - to(l_3)\})\}$
- $contribution[get - to(l_3)] = \{(0, \emptyset), (1, \{get - to(l_1)\})\}$
- $contribution[deliver(p_1)] = \{(0, \{get - to(l_1)\}$
- $contribution[deliver(p_2)] = \{(0, \{get - to(l_2)\}$
- $contribution[deliver(p_1, p_2)] = \{(0, \{deliver(p_1, p_2)\}$

Contributions after Post-Calculation

- $contribution[drive] = 1$
- $contribution[get - to(l_1)] = 1$
- $contribution[get - to(l_2)] = 3$
- $contribution[get - to(l_3)] = 2$
- $contribution[deliver(p_1)] = 1$
- $contribution[deliver(p_2)] = 3$
- $contribution[deliver(p_1, p_2)] = 4$

Since $tn_I = \{deliver(p_1, p_2)\}$ the algorithm will conclude $drive_{max} = 4$.

3.3.2 Bottom-Up-Method Algorithm

In the bottom-up approach the method and the task algorithm are nearly identical. A method can only be used if the compound task that it decomposes exists. For the top-down approach the underlying data structure could be simplified. Already decomposed tasks did not need to be stored for the top-down-method algorithm. Such simplifications do not apply to the bottom-up approach. Therefore no additional algorithm will be presented.

3.4 Comparison: Bottom-Up and Top-Down

When comparing both approaches the most significant differences is the memory usage. As alluded to earlier, the memory usage of the Top-Down algorithm scales exponentially with the depth of the Decomposition Tree. This is partially due to the property of the Top-Down algorithms to calculate every leaf node at once. Although this makes it convenient to modify the algorithm to calculate every maxima at once, the memory usage is still overwhelming. Since the ordering in which tasks have to be analysed is non-deterministic in the Bottom-Up approach it is also harder to implement and less intuitive to understand. On the Bottom-Up approach is way better at singling out one task and analysing exclusively nodes that relate to said task. Top-Down can not predict which branches are relevant for a single task since they might not appear till the leaf nodes. Bottom-Up will only calculate contributions for tasks that can potentially produce the task in question. But it has to kept in mind that this property comes with the downside of potential redundancies when calculating the maxima of every task. The leaves node of the Top-Down approach can be reused, while the contribution of the Bottom-Up approach need to be recalculated for every task in the Tree.

4 Adaptation of the Delete- and Ordering-Relaxation Heuristic

A very common way to solve HTN planning problems is by using heuristically guided search algorithms. The two main strategies for search algorithms for HTN Planning are decomposition-based search and progression-based search Höller et al. [2020b]. At first most heuristics were designed to be used for decomposition-based search (e.g. Bercher et al. [2017]), but in recent times a lot of heuristics for progression-based search have emerged (e.g. Höller et al. [2019]). These heuristics deal with cyclic planning problems the same way as they do with acyclic planning problems. Acyclic problems are limited in the size of their solutions and have properties that can be used by such heuristics. Acyclic TO HTN problems are even solvable in PSAPCE as shown by Alford et al. [2015]. To explore the effectiveness of using acyclic planning problem properties in heuristic search the DOR-Heuristic from Höller et al. [2020a] will be serving as an example. In this chapter we present some adaptations to increase its performance on acyclic HTN planning problems. Later we will empirically evaluate the performance of the DOR-Heuristic and its adaptations that have been integrated into the PANDA-Framework Höller et al. [2021] using the progression algorithm by Höller et al. [2020b].

4.1 Delete and Ordering-free HTN planning

Delete and Ordering-free (DOF) HTN planning problems form a relaxed subclass of HTN planning problems(see section 2.2). As the name suggests in DOF HTN planning problems, the set of delete-effects of all actions are empty and there are no ordering constraints, neither in the initial task network nor in the task networks defined by the methods. HTN planning is generally speaking undecidable Alford et al. [2015] but as shown in Höller et al. [2020a] DOF HTN planning is NP-Complete.

4.2 Calculation of the Heuristic

In order to calculate the heuristical values for a search node in a search algorithm, the HTN problem will be relaxed by removing all delete effects and ordering constraints. Then the goal distance starting from the original search node will be calculated. But

instead of calculating the distance in the original problem the distance will be calculated in the DOF problem. The calculation is realized using an integer programming model. Said goal distance then will be used as a heuristic value for the original search node.

4.3 ILP Model

Integer Programming is a mathematical optimization method, that is trying to find assignments for integer variables in order to optimize a linear function. Simultaneously these integer variables need to satisfy some constraints(in the form of linear equations). Since we want to find the smallest possible goal distance, a function resembling the goal distance needs to be minimized. The function that is being used in the DOR-Heuristic is shown below.

$$min \sum_{n \in N} T_n + \sum_{m \in M} M_m \tag{O}$$

In this formula T_n and M_m represent a set of variables that indicate how often the (primitive or compound) task n and the method m occur in the solution. The constraints that ensure the executability of the resulting tasks are organized in three groups. In total there are 13 constraints:

- **C1-C6:** State feature constraints that ensure every precondition gets properly established and the goal state will eventually be achieved
- **C7-C8:** Decomposition constraints that ensure every compound task and primitive action in the solution can get derived from the given decomposition structure
- **C9-C13:** Further decomposition constraints to prevent the IP model from using self sustaining cycles

Höller et al. [2020a] proofs that for every valid assignment of the variables in the IP model there is a solution for the encoded DOR HTN problem and vice versa.

4.4 Removal of Constraints

As stated in Höller et al. [2020a], C7-C8 specify the relationship between tasks and methods in a decomposition tree. For cyclic problems C9-C13 are needed to prevent the IP model from building self sustaining cycles. Therefore in a acyclic Delete and Ordering-Relaxed HTN problem a valid assignments to the variables in the first 8 constraints are sufficient to proof there is a solution for the encoded problem. C9-C13 can therefore be removed from the model in order to create performance improvements on acyclic domains.

4.5 Additional Constraints

In cyclic HTN problems certain tasks can reproduce themselves indefinitely often, which not only needs to be prevented by any solving mechanism (in case of DOR-Heuristic by C9-C13). This also means it is generally speaking impossible to calculate a limit for the occurrences of tasks in the solution. Acyclic HTN problems in contrast to cyclic HTN problems can only produce a limited amount of tasks through its decomposition structure. Therefore a maximum amount of occurrences of tasks (and consequentially their methods) can be calculated. These values can be used to create two additional constraints to the ILP that can assist its calculation. In section 3 algorithms are presented to calculate such values. Given the values of $T_{n,max}$ and $M_{m,max}$ each representing the maximum amount of decompositions of task n and occurrences of method m respectively. The following constraints can be added to the ILP Model.

$$\forall t \in N : T_n \leq T_{n,max} \tag{C14}$$
$$\forall m \in M : M_t \leq M_{m,max} \tag{C15}$$

Since the ILP does not have to calculate $T_{n,max}$ and $M_{m,max}$ itself, adding such constraints will likely improve the average calculation time. On the flip side these improvements have to make up for the calculation time of the maxima. This raises the question on how often these values have to be calculated. Inside of the progression algorithm of Höller et al. [2020b] the heuristic will be recalculated for every expanded search node. Therefore for every consequent search node the ILP will also be recalculated, which opens up the ability for recalculating the maxima. Generally speaking there are three possible ways for updating the max values:

- Recalculate the maxima every search node
- Calculate the maxima at the root node and subtract every already expanded task/used method
- Calculate the maxima at the root node and reuse the same values

These possibilities represent a trade of between computation time and accuracy of the maxima. Recalculating every node will result in the smallest maxima induced by the decomposition structure, but it is also very time consuming. Only calculating the maxima once is the cheapest method to compute resulting in the highest maxima(which are still correct maxima). Updating the maxima through subtraction of already expanded tasks/used methods represents a compromise between the two extremes.

4.6 Evaluation

Here the adaptations of removing and adding constraints to the ILP Model of the DOR-Heuristic will be practically evaluated. For this purpose the DOR-Heuristic was deployed in the PANDA-Framework using a progression algorithm. The algorithm has been deployed with a time-limit of 60s per problem and 6GB of Memory using a Intel core I7-7700K Processor. The HTN domains and problem instances for the evaluation were introduced for the International Planning Competition 2020. Of these problems all acyclic instances from five different domains have been selected for evaluation. For fair (average) runtime comparisons between different configurations of the ILP Model only the problem instances that get solved by every configuration will be used.

4.6.1 Removal of Constraints

		C1-C8		C1-C13	
Domain	#instances	#solved	avg. runtime in s	#solved	avg. runtime in s
Barman	20	9	10.272	9	10.618
Rover	2	2	0.336	2	0.305
Woodworking	30	14	3.759	14	3.835
Childsnack	26	8	8.695	8	9.061
Minecraft	42	39	9.644	39	10.338

Figure 4.1: Runtime Comparison of the DOR-Heuristic using a limited and full set of constraints Runtime comparisons refer to the instances that got solved by every configuration

As shown in Figure 4.1 removing C9-C13 will result in some slight increase in runtime performance, but not enough to solve additional problems

4.6.2 Additional Constraints

The two figures below compare the different update methods using C1-C8 and C14-C15 as constraints.

Domain	#instances	update every node #solved	update with subtraction #solved	only root node #solved
Barman	20	4	9	10
Rover	2	2	2	2
Woodworking	30	12	14	14
Childsnack	26	0	8	8
Minecraft	42	23	38	39

Domain	#instances	update every node avg. runtime in s	update with subtraction avg. runtime in s	only root node avg. runtime in s
Barman	4	29.574	4.738	4.590
Rover	2	0.7625	0.434	0.3305
Woodworking	12	12.213	3.824	3.564
Childsnack	8	-	27.774	26.791
Minecraft	23	23.157	2.145	1.884

Figure 4.2: Comparison of different constraint update methods, ordered by accuracy of the maxima from left to right in descending order. Runtime comparisons refer to the instances that got solved by every configuration

As can be seen in figure 4.2 the runtime does significantly increase with a more accurate update mechanism. Hence the improvements in the computation time of the ILP does not outweigh the additional computation time of the update mechanism. Solving the maxima only once is therefore most effective method to minimize the computation time. In order to investigate whether adding C14-C15 will pose a improvement we will use said mechanism going forward.

4.7 Results

Now that the impact of removing and adding constraints has been discussed the question arises:"which configuration is the best?". In order to answer this question all combinations of constraints need to be compared. Figure 4.3 and figure 4.4 compares the the amount of produced solution and the respective runtime for the four combinations:

- **C1-C13** The default configuration without any adaptations
- **C1-C8** Removing constraints
- **C1-C15** Adding constraints
- **C1-C8 & C14-C15** Adding constraints and removing constraints

Domain	#instances	C1-C13 #solved	C1-C8 #solved	C1-C15 #solved	C1-C8 & C14-C15 #solved
Barman	20	9	9	10	10
Rover	2	2	2	2	2
Woodworking	30	13	14	14	14
Childsnack	26	8	8	8	8
Minecraft	42	39	39	39	39

Figure 4.3: Comparisons of the amount of solved problems

Domain	# instances	C1-C13	C1-C8	C1-C15	C1-C8 & C14-C15
Barman	9	23.597	22.826	20.193	20.785
Rover	2	0.305	0.336	0.321	0.330
Woodworking	13	8.218	8.055	7.583	7.658
Childsnack	8	29.451	28.259	26.597	26.791
Minecraft	39	11.134	10.440	9.662	9.695

Figure 4.4: Comparisons of the average runtime in seconds. Runtime comparisons refer to the instances that got solved by every configuration

In terms of solved problems the four different configurations seem to perform very similarly. However when comparing the runtimes some differences become apparent. Adding C14 and C15 does yield performance increase across all domains and all configurations. Removing constraints C9-C13 has a much smaller impact in comparison. Despite expectations the configuration featuring C1-C15 is even slightly faster then the configuration with C1-C8 & C14-C15.

5 Translation Acyclic TO HTN to Classical TO Planning

5.1 Motivation

Classical planning is as the name suggest a simpler and older formalism reaching back to the work of Fikes and Nilsson [1971]. Since then a lot of very sophisticated solving techniques have evolved(e.g. Helmert [2006], Fickert et al. [2018]). Hierarchical planning emerged more recently and gained increased attention in the last couple years (Bercher et al. [2019]). But since most of classical planning research was already established beforehand. A lot of research on hierarchical planning was assisted by classical planning. Examples for this are the groundings of Behnke et al. [2020] and the heuristics of Höller et al. [2018]. There are also some approaches to translate hierarchical into classical planning problems (e.g. Höller [2021] and Behnke et al. [2022]). Translations like Alford et al. [2016] rely on a progression bound that limits the maximum amount of tasks a task network can contain. Acyclic TO HTN problems are bound in size by definition therefore performing a translation should be simpler with a limited encoding size. Also classical planning shares the same complexity class with acyclic TO HTN Planning: PSPACE-complete. Therefore a translation without loss of expressiveness should be possible.

5.2 Translation Goal

As defined in section 2 a HTN problem is a tuple $P_{htn} = (L, C, A, M, tn_I, s_I, g, \delta)$, whereas a classical planning problem consists of the tuple $P_{classical} = (L, A, s_0, g, \delta)$. To avoid ambiguities a classical problem will be referred to with the tuple $P_{classical} = (L_c, A_c, s_c, g_c, \delta_c)$. A solution to a TO HTN problem is given through a solution network tn_{sol} containing a total ordered sequence of primitive actions. This formalism is very similar to a total-ordered solution of a classical planning problem which consists of total ordered sequence of actions that we refer to as cl_{sol}. The main difference of a TO HTN solution to a classical TO solution are the restrictions imposed by the decomposition structure of the HTN problem. In order to perform a translation from $P_{htn} \rightarrow P_{classical}$ that contains the same set of solutions, the properties of the decomposition structure need to be encoded into the δ_c functions of its actions A_c. With newly defined state features, actions and updated δ functions a new $P_{classical}$ can be defined.

In $P_{classical}$ only task can be inserted into cl_{sol} if they could also be decomposed in P_{htn}.

5.3 Encoding

A method $m = (c, tn)$ decomposes the task c into the task network tn. The (total-ordered) task network $tn = (T, \prec, \alpha)$ of method describes a sequence of tasks (compound tasks or primitive actions). This sequence can be denoted as $\langle T_1, .., T_i, .., T_n \rangle$ with $\alpha(T_i) = \begin{cases} c_{T_i} & \alpha(T_i) \subseteq C \\ a_{T_i} & \alpha(T_i) \subseteq A \end{cases}$
Without loss of generality, the method $m0 = (c0, tn_I)$ will be added to M. For the translation $P_{classical} = (L_c, A_c, s_c, g_c, \alpha_c)$ will be initialized with $(L, \emptyset, s_I \cup c0(start), g \cup c0(end), \emptyset)$. By iterating through of every method, actions and state features will be added. Additions to $P_{classical}$ will be indicated by $\Rightarrow$.

$\forall c \in C \Rightarrow L_c \cup \{c(start), c(end)\}$
$\forall m \in M\ m = (c_h, tn)$ with $tn = (\langle T_1, .., T_i, .., T_n \rangle, \alpha)$
iterate through every task T_i *in* tn, *additions to* $P_{classical}$ *depend on* $\alpha(T_i)$

if $\alpha(T_1) = c_{T_1}$
$\Rightarrow A_c \cup c[m, 1] \Rightarrow st(m, 1) \cup L_c$
$\Rightarrow prec(c[m, 1]) = \{c_h(start)\}$
$\Rightarrow del(c[m, 1]) = prec(c[m, i])$
$\Rightarrow add(c[m, 1]) = \begin{cases} c_h(end) & n = 1 \\ \{c_{T_1}(start), st(m, 1)\} & else \end{cases}$

if $\alpha(T_1) = a_{T_1}$
$\Rightarrow A_c \cup a[m, 1] \Rightarrow st(m, 1) \cup L_c$
$\Rightarrow prec(a[m, 1]) = prec(a_{T_1}) \cup \{c_h(start)\}$
$\Rightarrow del(a[m, 1]) = del(a_{T_1}) \cup prec(a[m, i])$
$\Rightarrow add(c[m, 1]) = add(a_{T_1}) \cup \begin{cases} c_h(end) & n = 1 \\ \{st(m, 1)\} & else \end{cases}$

if $\alpha(T_i) = c_{T_i}$

$\Rightarrow A_c \cup c[m,i] \Rightarrow st(m,i) \cup L_c$

$$\Rightarrow prec(c[m,i]) = \begin{cases} \{c_{T_{i-1}}(end), st(m,i-1)\} & \alpha(T_{i-1}) \subseteq C \\ \{st(m,i-1)\} & \alpha(T_{i-1}) \subseteq A \end{cases}$$

$\Rightarrow del(c[m,i]) = prec(c[m,i])$

$\Rightarrow add(c[m,i]) = \{c_{T_i}(start), st(m,i)\}$

if $\alpha(T_i) = a_{T_i}$

$\Rightarrow A_c \cup a[m,i] \Rightarrow st(m,i) \cup L_c$

$$\Rightarrow prec(a[m,i]) = prec(a_{T_i}) \cup \begin{cases} \{c_{T_{i-1}}(end), st(m,i-1)\} & \alpha(T_{i-1}) \subseteq C \\ \{st(m,i-1)\} & \alpha(T_{i-1}) \subseteq A \end{cases}$$

$\Rightarrow del(a[m,i]) = del(a_{T_i}) \cup prec(a[m,i])$

$\Rightarrow add(a[m,i]) = \{st(m,i)\} \cup add(a_{T_i})$

if $\alpha(T_n) = c_{T_n}$

$\Rightarrow A_c \cup c[m,n] \cup c[m,n+1] \Rightarrow st(m,n) \cup L_c$

$$\Rightarrow prec(c[m,n]) = \begin{cases} \{c_h(start)\} & n = 1 \\ \{c_{T_{n-1}}(end), st(m,n-1)\} & \alpha(T_{n-1}) \subseteq C \\ \{st(m,n-1)\} & \alpha(T_{n-1}) \subseteq A \end{cases}$$

$\Rightarrow del(c[m,n]) = prec(c[m,n])$

$\Rightarrow add(c[m,n]) = \{C_{T_n}(start), st(m,n)\}$

$\Rightarrow prec(c[m,n+1]) = \{C_{T_n}(end), st(m,n)\}$

$\Rightarrow del(c[m,n+1]) = prec(c[m,n+1])$

$\Rightarrow add(c[m,n+1]) = c_h(end)$

if $\alpha(T_n) = a_{T_n}$

$\Rightarrow A_c \cup a[m,n]$

$$\Rightarrow prec(a[m,n]) = \begin{cases} \{c_h(start)\} & k = 1 \\ \{c_{T_{n-1}}(end), st(m,n-1)\} & \alpha(T_{n-1}) \subseteq C \\ \{st(m,n-1)\} & \alpha(T_{n-1}) \subseteq A \end{cases}$$

$\Rightarrow del(c[m,n]) = prec(c[m,n])$

$\Rightarrow add(c[m,n]) = c_h(end)$

Since subtasks can only be introduced in a HTN problem through the application of a method. The application of every method in $m \in M$ need to be encoded into equivalent state features, actions and δ functions. Every subtask of every method will be translated into a unique action. Subtasks of a method $m = (c_h, tn)$ are allowed into cl_{sol} if $c_h(start)$ is satisfied. $c_h(start)$ will be consumed by the first subtask of a method, to prevent

subtasks of other methods decomposing c_h to be inserted. The last subtask has $c_h(end)$ as add effect, signaling that all subtasks were inserted. To ensure subtasks will be applied in the TO imposed by tn, the subtask state features are introduced: $st(m, i)$. The ith subtaks may only be inserted if the precondition $st(m, i-1)$ is satisfied, deleting it afterwards. The ith subtask then adds $st(m, i)$ allowing the next subtask to be applied. If a subtask is a compound task c it will receive additionally $c(start)$ as add-effect allowing subtasks of c to be inserted afterwards. The successor of a compound subtask c will receive additionally $c(end)$ as precondition ensuring that the subtasks of some $m = (c, tn)$ will be inserted in between. A special case occurs if the last subtask of a method is a compound task. Then there is no successor task with $c(end)$ as precondition, therefore a new action will be created with $c(end)$ as precondition and $c_h(end)$ as add-effect (in addition to its subtask precondition/effects). If a subtask is a primitive action $a \in A$ and it will be translated to $a[m, i]$, it will keep the preconditions, add- and delete effects of a from P_{htn}.

5.4 Example

To demonstrate the translation, the Acyclic HTN Problem from section 3.1 will be used. The formal definition of P_{htn} is described in Appendix A. The formal definition of the translated problem $P_{classical}$ is described in Appendix B.

For this translation the effects of $m2 = (c2, tn)$ with $tn = (\langle T_1, T_2, T_3 \rangle, \alpha)$ will be encoded into three unique actions. $c2$ relates to two state features: $c2(start)$ and $c2(end)$ which will be used in the following endcoding.
The first action is:

a[m2,1]
prec: c2(start), at(truck,location0), at(package1,location0)
add: st(m2,1), loaded(package1)
del: c2(start), at(package1,location1)

Since $\alpha(T_1) = a5 \subseteq A$, $a[m2, 1]$ will keep the precondition, add- and delete effects of $a5$. As first subtask of a decomposition of $c2$, $a[m2, 1]$ will receive $c2(start)$ as precondition and delete-effect. To enable the application of the next subtask, $st(m2, 1)$ is introduced and used as add-effect of $a[m2, 1]$.
The second action is:

c[m2,2]
prec: st(m2,1)
add: c4(start), st(m2,2)
del: st(m2,1)

A precondition of the second subtask, need to ensure the first subtask was applied beforehand. Therefore $st(m2, 1)$ is a precondition and a delete-effect of $c[m2, 1]$. Since $\alpha(T_1) = c4 \subseteq C$, $c[m2, 1]$ will receive $c4(start)$ as add-effect allowing for subtasks of some $m = (c4, tn)$ to be inserted. To enable the application of the next subtask, $st(m2, 2)$ is introduced and used as add-effect of $c[m2, 2]$.
The third action is:

a[m2,3]
prec: st(m2,2), c4(end), at(truck,location1), loaded(package1)
add: c2(end), at(package1,location1)
del: st(m2,2), c4(end), loaded(package1)

A precondition of the third subtask, need to ensure the second subtask was applied beforehand. Therefore $st(m2, 2)$ is a precondition and a delete-effect of $a[m2, 3]$. Since $\alpha(T_1) = a7 \subseteq A$, $a[m2, 3]$ will keep the precondition, add- and delete effects of $a7$. $a7$ was preceded by $c4$ in tn, therefore it has the additional precondition of $c4(end)$ to ensure subtasks of some $m = (c4, tn)$ were inserted beforehand. As last subtask of $m2$, $a[m2, 3]$ also has $c2(end)$ as add-effect to signal that a set of subtasks of some $m = (c2, tn)$ was inserted.

6 Conclusion

In this thesis the complexities and challenges of acyclic HTN planning has been explored. Structures in acyclic decomposition trees were investigated, resulting in the max-task/max-action algorithms. They calculate the maximum occurrences of a given task/method when decomposing a task network in an acyclic HTN problem. These algorithms were designed with a intuitive but memory intensive top-down approach and a more complicated but less memory consuming bottom-up approach.

Putting these algorithms to use, adaptations for the DOR-Heuristics from Höller et al. [2020a] were proposed. The DOR-Heuristics are based on a delete- and ordering-free sub-class of HTN planning. The goal distance for delete and ordering free HTN problems can be calculated through an IP/LP model with 13 constraints. These constraints were adapted to better deal with acyclic HTN problems. The constraints which relate to cyclic decomposition structures were removed. Additionally constraints that limit the occurrence of methods and tasks were added. These limits were calculated using the proposed max-task/max-action algorithms. With these adaptations the DOR-Heuristic was deployed in the PANDA-Framework using a progression algorithm to solve acyclic HTN problems. The adaptations were tested both independently and combined on 120 acyclic TO HTN problems over 5 domains from the International Planing Competition 2020. Also different update mechanism for the max-task/max-methods constraints were compared. While the removal of constraints led to mixed results, the additional constraints calculated by the bottom-up algorithms showed significant improvements.

Lastly an encoding to translate acyclic TO HTN problems into classical TO problems was proposed. By iterating through all methods of the HTN problem, new state features and actions were introduced. These additional features and actions serve to mimic the effects of the methods in the original HTN problem. Creating a classical problem with similar expressiveness and complexity featuring at most $|M| \cdot |C|$ new actions. Enabling the deployment of sophisticated classical planning heuristics and solving techniques for acyclic TO HTN problems.

A Formal Definition: Transport Example

The propositional state features L are:

l1 at(package1,location0)

l2 at(package2,location0)

l3 at(package1,location1)

l4 at(package2,location1)

l5 at(package1,location2)

l6 at(package2,location2)

l7 at(package1,location3)

l8 at(package2,location3)

l9 at(truck,location0)

l10 at(truck,location1)

l11 at(truck,location2)

l12 at(truck,location3)

l13 loaded(package1)

l14 loaded(package2)

The compound tasks C are:

c1 deliver(package1,package2)

c2 deliver(package1)

c3 deliver(package2)

c4 get-to(location1)

c5 get-to(location2)

c6 get-to(location3)

The methods M decompose the following tasks into their respective total-ordered subtasks:

m1 decompose: deliver(package1,package2)
subtasks: deliver(package1), deliver(package2)

m2 decompose: deliver(package1)
subtasks: pick-up(package1,location0), get-to(location1), drop(package1,location1)

m3 decompose: deliver(package2)
subtasks: pick-up(package2,location0), get-to(location2), drop(package2,location2)

m4 decompose: get-to(location1)
subtasks: do-nothing(location1)

m5 decompose: get-to(location1)
subtasks: drive(location0,location1)

m6 decompose: get-to(location2)
subtasks: do-nothing(location2)

m7 decompose: get-to(location2)
subtasks: get-to(location1), drive(location1,location2)

m8 decompose: get-to(location2)
subtasks: get-to(location3), drive(location3,location2)

m9 decompose: get-to(location3)
subtasks: do-nothing(location3)

m10 decompose: get-to(location3)
subtasks: get-to(location1), drive(location1,location3)

The primitive actions A with their respective mappings of δ are:

a1 drive(location0,location1)
prec: at(truck,location0) **add**: at(truck,location1) **del**: at(truck,location0)

a2 drive(location1,location3)
prec: at(truck,location1) **add**: at(truck,location3) **del**: at(truck,location1)

a3 drive(location1,location2)
prec: at(truck,location1) **add**: at(truck,location2) **del**: at(truck,location1)

a4 drive(location3,location2)
prec: at(truck,location3) **add**: at(truck,location2) **del**: at(truck,location3)

a5 pick-up(package1,location0)
prec: at(truck,location0), at(package1,location0) **add**: loaded(package1)
del: at(package1,location0)

a6 pick-up(package2,location1)
prec: at(truck,location1), at(package2,location1) **add**: loaded(package2)
del: at(package2,location1)

a7 drop(package1,location1)
prec: at(truck,location1), loaded(package1) **add**: at(package1,location1)
del: loaded(package1)

a8 drop(package2,location2)
prec: at(truck,location2), loaded(package2) **add**: at(package2,location2)
del: loaded(package2)

a9 do-nothing(location1)
prec: at(truck,location1) **add**: ∅ **del**: ∅

a10 do-nothing(location2)
prec: at(truck,location2) **add**: ∅ **del**: ∅

a11 do-nothing(location3)
prec: at(truck,location3) **add**: ∅ **del**: ∅

B Transport Example translated into STRIPS

The propositional state features L from the original HTN Problem are:

- **l1** at(package1,location0)
- **l2** at(package2,location0)
- **l3** at(package1,location1)
- **l4** at(package2,location1)
- **l5** at(package1,location2)
- **l6** at(package2,location2)
- **l7** at(package1,location3)
- **l8** at(package2,location3)
- **l9** at(truck,location0)
- **l10** at(truck,location1)
- **l11** at(truck,location2)
- **l12** at(truck,location3)
- **l13** loaded(package1)
- **l14** loaded(package2)

The additional state features from the translation regarding the compound tasks are:

- **l15** c1(start)
- **l16** c1(end)
- **l17** c2(start)
- **l18** c2(end)
- **l19** c3(start)
- **l20** c3(end)
- **l21** c4(start)
- **l22** c4(end)
- **l23** c5(start)
- **l24** c5(end)
- **l25** c6(start)
- **l26** c6(end)

The additional state features from the translation regarding the subtasks are:

l27 st(m1,1)

l28 st(m1,2)

l29 st(m2,1)

l30 st(m2,2)

l31 st(m3,1)

l32 st(m3,2)

l33 st(m7,1)

l34 st(m8,1)

l35 st(m10,1)

The primitive actions A_c with their respective mappings of δ_c are:

c[m1,1] **prec**: c1(start) **add**: c2(start), st(m1,1)
del: c1(start)

c[m1,2] **prec**: c2(end) **add**: c3(start), st(m1,2)
del: c2(end)

c[m1,3] **prec**: c3(end) **add**: c1(end)
del: c3(end)

a[m2,1] **prec**: c2(start), at(truck,location0), at(package1,location0)
add: st(m2,1), loaded(package1)
del: c2(start), at(package1,location1)

c[m2,2] **prec**: st(m2,1) **add**: c4(start), st(m2,2)
del: st(m2,1)

a[m2,3] **prec**: st(m2,2), c4(end), at(truck,location1), loaded(package1)
add: c2(end), at(package1,location1)
del: st(m2,2), c4(end), loaded(package1)

a[m3,1] **prec**: c3(start), at(truck,location1), at(package2,location1)
add: st(m3,1), loaded(package2)
del: c3(start), at(package2,location1)

c[m3,2] **prec**: st(m3,1) **add**: c5(start), st(m2,2)
del: st(m2,1)

a[m3,3] **prec**: st(m3,2), c5(end), at(truck,location2), loaded(package2)
add: c3(end), at(package2,location2)
del: st(m3,2), c5(end), loaded(package2)

a[m4,1] **prec**: c4(start), at(truck,location1)
add: c4(end) **del**: c4(start)

a[m5,1] **prec**: c4(start), at(truck,location0)
add: c4(end), at(truck,location1)
del: c4(start), at(truck,location0)

a[m6,1] **prec**: c5(start), at(truck,location1)
add: c5(end) **del**: c5(start)

c[m7,1] **prec**: c5(start)
add: c4(start), st(m7,1) **del**: c5(start)

a[m7,2] **prec**: st(m7,1), c4(end), at(truck,location1)
add: c5(end), at(truck,location2)
del: st(m7,1), c4(end), at(truck,location1)

c[m8,1] **prec**: c5(start)
add: c6(start), st(m8,1) **del**: c5(start)

a[m8,2] **prec**: c6(end), st(m8,1), at(truck,location3)
add: c5(end), at(truck,location2)
del: c6(end), st(m8,1), at(truck,location3)

a[m9,1] **prec**: c6(start), at(truck,location3)
add: c6(end) **del**: c6(start)

c[m10,1] **prec**: c6(start)
add: c4(start), st(m10,1) **del**: c6(start)

a[m10,2] **prec**: c4(end), st(m10,1), at(truck,location1)
add: c6(end), at(truck,location3)
del: c4(end), st(m10,1), at(truck,location1)

Bibliography

Daniel Höller, Pascal Bercher, and Gregor Behnke. Delete-and ordering-relaxation heuristics for htn planning. In *Proceedings of the 29nd international joint conference on artificial intelligence(IJCAI)*, pages 4076–4083, 2020a.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

Pascal Bercher, Ron Alford, and Daniel Höller. A survey on hierarchical planning-one abstract idea, many concrete realizations. In *Proceedings of the 28th international joint conference on artificial intelligence (IJCAI)*, pages 6267–6275, 2019.

Kutluhan Erol, James A Hendler, and Dana S Nau. *Semantics for hierarchical task-network planning*. 1994.

Ron Alford, Vikas Shivashankar, Ugur Kuter, and Dana Nau. On the feasibility of planning graph style heuristics for htn planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, pages 2–10, 2014.

Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on artificial intelligence (IJCAI)*, volume 2, pages 968–973, 1999.

Thomas Geier and Pascal Bercher. On the decidability of htn planning with task insertion. In *Proceedings of the 22nd international joint conference on artificial intelligence(IJCAI)*, pages 1955–1961, 2011.

Ron Alford, Pascal Bercher, and David Aha. Tight bounds for htn planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 7–15, 2015.

Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. The panda framework for hierarchical planning. *KI-Künstliche Intelligenz*, 35(3):391–396, 2021.

Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. Heuristics and symmetries in classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

Florian Pommerening, Malte Helmert, and Blai Bonet. Higher-dimensional potential heuristics for optimal classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Maximilian Fickert, Daniel Gnad, Patrick Speicher, and Jörg Hoffmann. Saarplan: Combining saarland's greatest planning techniques. *IPC2018–Classical Tracks*, pages 10–15, 2018.

Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. A generic method to guide htn progression search with classical heuristics. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, pages 114–122, 2018.

Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. On guiding search in htn planning with classical planning heuristics. In *Proceedings of the 28th international joint conference on artificial intelligence (IJCAI)*, pages 6171–6175, 2019.

Ron Alford, Gregor Behnke, Daniel Höller, Pascal Bercher, Susanne Biundo, and David Aha. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive htn problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 26, pages 20–28, 2016.

Daniel Höller. Translating totally ordered htn planning problems to classical planning problems using regular approximation of context-free languages. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 159–167, 2021.

Gregor Behnke, Florian Pollitt, Daniel Höller, Pascal Bercher, and Ron Alford. Making translations to classical planning competitive with other htn planners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 9687–9697, 2022.

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

Pascal Bercher, Gregor Behnke, Daniel Höller, and Susanne Biundo. An admissible htn planning heuristic. In *Proceedings of the 26th international joint conference on artificial intelligence (IJCAI)*, pages 480–488, 2017.

Gregor Behnke, 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.

Eidesstattliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig angefertigt zu haben. Dabei wurden nur die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ulm, den 7. Juni, 2024 _____

(Linus Diepold)