Lecture *Hierarchical Planning*

Chapter: *Complexity Results for Plan Existence*

Dr. Pascal Bercher

Institute of Artificial Intelligence,
Ulm University, Germany

Winter Term 2018/2019

(Compiled on: February 20, 2019)

ulm university universität uulm

Overview:

What are Complexity Studies?

Complexity analysis studies the computational hardness of a decision problem. In this lecture we study:

- The *plan existence problem*:
  How hard is it to decide whether a problem $\mathcal{P}$ has a solution?
- The *plan verification problem*:
  How hard is it to decide whether a given plan is actually a solution?

Why are we Interested in Complexity Studies?

Benefits of complexity studies:

- We know how to design algorithms:
  - If a problem is undecidable, any terminating algorithm must be wrong. Similarly: if a problem is $\mathbb{NP}$-*complete*, it is not a good idea to design a decision procedure that runs in polynomial time.
  - If the complexity of a problem is not known, at which runtime should we aim? $\mathbb{P}$? $\mathbb{EXPTIME}$?
- We can identify special cases to be exploited by algorithms. Example: heuristics! (Most of them exploit special cases that can be decided in $\mathbb{P}$.)
- Insights may also allow for compilation techniques.
- Last, but *not-at-all least*: they help understanding the problem! (Understanding the problem should always be the first step.)

Decidability, Undecidability

- A problem is *decidable* if there is an algorithm that, for each possible input, terminates after a finite time with the correct solution (i.e., *true* or *false*).
- More formally, a set of natural numbers $N \subseteq \mathbb{N}$ is called decidable if the function $\chi_N : \mathbb{N} \to \{0, 1\}$ can be computed, where:

$$\chi_N(n) = \begin{cases} 1 & \text{if } n \in N \\ 0 & \text{otherwise} \end{cases}$$

- A problem is called *undecidable* if it is not decidable.

Semi-decidability

- A problem is semi-decidable if there is an algorithm that, for each possible input, terminates eventually in case the correct answer is *yes*. For instance, breadth-first-search usually serves as proof for the semi-decidability.
- More formally, a set of natural numbers $N \subseteq \mathbb{N}$ is called semi-decidable if the function $\chi_N : \mathbb{N} \to \{undef, 1\}$ can be computed, where:

$$\chi_N(n) = \begin{cases} 1 & \text{if } n \in N \\ undef & \text{otherwise} \end{cases}$$

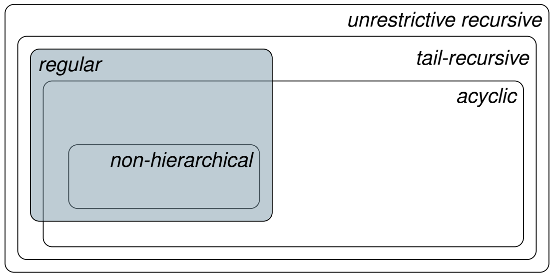
→ Corollary: Each decidable problem is semi-decidable.

- Note: semi-decidable problems (sets) are also called, among others, *recursively enumerable*.

Overview

Which properties make the plan existence problem easier?

- Task insertion.
- Total order of all task networks.
- Recursion. Methods are:
  - *acyclic*: no recursion.
  - *regular*: only one compound task, which is the last one.
  - *tail-recursive*: arbitrary many compound tasks, only the last one is recursive.



Totally Ordered Problems, Problem Definition

An HTN planning problem $\mathcal{P}$ is called totally ordered if:

- All decomposition methods are totally ordered, i.e., for each $m \in M$, $m = (c, tn)$, tn is a totally ordered task network.
- In case $\mathcal{P}$ uses an *initial task network* tn_I rather than an *initial task* c_I, then tn_I needs to be totally ordered as well.

Regular Problems, Problem Definition

- A task network $tn = (T, \prec, \alpha)$ is called *regular* if
  - at most one task in T is compound and
  - if $t \in T$ is a compound task, then it is the last task in tn, i.e., all other tasks $t' \in T$ are ordered before t.
- A method (c, tn) is called regular if tn is regular.
- A planning problem is called regular if all methods are regular.

Note: In case the planning problem features an initial task network, a problem is defined as regular if this network is regular, too. (Although this restriction in not necessary with regard to the results that base upon it.)

Tail-Recursive Problems, Informal Problem Definition

Informally, *tail-recursive* problems look as follows:

- limited recursion for all tasks in all methods
- non-last tasks have a more restricted recursion

Formally, the restrictions on recursion are defined in terms of so-called *stratifications*.

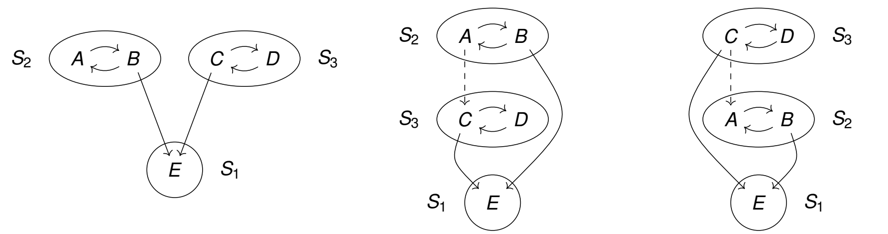
Stratifications

A stratification is defined as follows:

- A set $\leq \subseteq C \times C$ is called a *stratification* if it is a total preorder (i.e., reflexive, transitive, and *total*)

Stratifications: Example

(Non-)Examples for Stratifications:

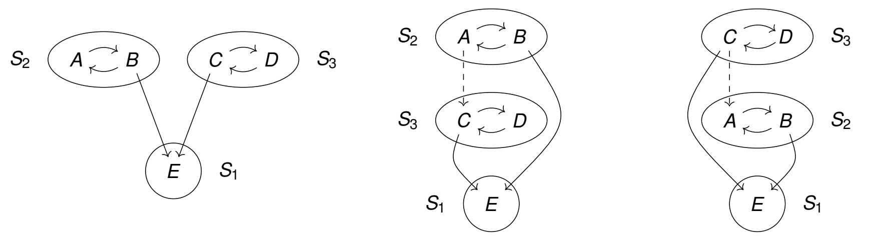


(a) Relation $\leq_a$. (b) Stratification $\leq_b$. (c) Stratification $\leq_c$.

- $\leq_a = \{(A, B), (B, A), (C, D), (D, C), (E, B), (E, C)\}$
- $\leq_a$ is not a stratification, as it is not total

Stratifications: Example

(Non-)Examples for Stratifications:



(a) Relation $\leq_a$. (b) Stratification $\leq_b$. (c) Stratification $\leq_c$.

- $\leq_a = \{(A,B),(B,A),(C,D),(D,C),(E,B),(E,C)\}$
- $\leq_b = \{(A,B),(B,A),(C,D),(D,C),(E,B),(E,C),(C,A)\}^*$
- $\leq_c = \{(A,B),(B,A),(C,D),(D,C),(E,B),(E,C),(A,C)\}^*$

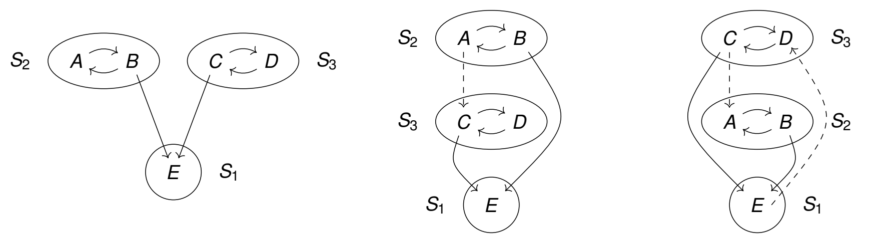
Stratifications

A stratification is defined as follows:

- A set $\leq \subseteq C \times C$ is called a *stratification* if it is a total preorder (i.e., reflexive, transitive, and *total*)
- We call any inclusion-maximal subset of C a *stratum* of $\leq$ if for all $x, y \in C$ both $(x,y) \in \leq$ and $(y,x) \in \leq$ hold.
- The *height of a stratification* is the number of its strata.

Stratifications: Example

(Non-)Examples for Stratifications:



(a) Relation $\leq_a$. (b) Stratification $\leq_b$. (c) Stratification $\leq_c$.

- $S_1 = \{E\}$, $S_2 = \{A,B\}$, and $S_3 = \{C,D\}$ are strata
- $\leq_b$ and $\leq_c$ have a height of 3.
- If we add, e.g., an edge from E to D in $\leq_c$, i.e., the tuple (D,E), then we only have *a single* stratification with height 1.

Tail-Recursive Problems,Problem Definition

An HTN problem $\mathcal{P}$ is called *tail-recursive* if there is a stratification $\leq$ on the compound tasks C of $\mathcal{P}$ with the following property:

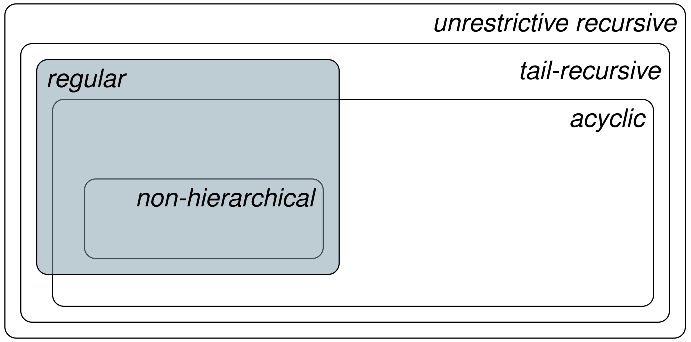
For all methods $(c,(T,\prec,\alpha)) \in M$ holds:

- If there is a *last* task $t \in T$ that is compound (i.e., $\alpha(t) \in C$ and for all $t' \neq t$ holds $(t',t) \in \prec$), then $(\alpha(t),c) \in \leq$.

  This means: the last task (if one exists) is at most as hard as the decomposed task c.
- For any non-last task $t \in T$ with $\alpha(t) \in C$ it holds $(\alpha(t),c) \in \leq$ and $(c,\alpha(t)) \notin \leq$.

  This means: any non-last task is easier (on a lower stratum) than the decomposed task c.

Overview of Problem Classes



Notes:

- Do not confuse these *problem classes* with the *language classes*!
- Totally ordered problems are not shown because this restriction is independent of all the ones depicted.

Problem Definition

- Decision problem: given a planning problem $\mathcal{P}$, does $\mathcal{P}$ possess a solution?
- For which problems do we already know their complexities?
  - STRIPS with positive preconditions and effects: in $\mathbb{P}$.
  - as before, but *k-length*: $\mathbb{NP}$*-complete*.
  - STRIPS with arbitrary preconditions and positive effects: $\mathbb{NP}$*-complete*.
- So, what's still missing?
  - STRIPS with arbitrary effects. Will show: $\mathbb{PSPACE}$*-complete*.
  - HTN planning under several restrictions (cf. *problem classes*).
  - TIHTN planning.

Complexity of the General Case, Membership

Theorem

Let $\mathcal{P}$ be a classical planning problem. Deciding whether $\mathcal{P}$ has a solution is $\mathbb{PSPACE}$*-complete*.

Proof, Membership:

- Maximal plan length that needs to be considered: 2^n with $n = |V|$.
- But we still only need polynomial space:
  - For all states s_1, s_2, we want to know whether there is a plan from s_1 to s_2. This is done via asking:
  - Is there a plan of length $\leq n$ from s_1 to s' and another from s' to s_2?
  - (This reduces the hardness of the plan existence problem of length $2n$ to two problems of length n each.)
  - By iterating over all states, this requires polynomial space.

Complexity of the General Case, Hardness

Theorem

Let $\mathcal{P}$ be a classical planning problem. Deciding whether $\mathcal{P}$ has a solution is $\mathbb{PSPACE}$*-complete*.

Proof, Hardness:

- We encode a space-bounded Turing-machine into a STRIPS problem.
- An operator checks the current state and tape content.
- The operators' effects encode the successor state and tape changes.
- Number of operators is proportional to number of transitions times tape squares.

More Complexity Results

There are several further cases that can be studied, e.g.:

- Take the number of preconditions/effects into account (special cases are often revealed via looking into the reductions).
- Perform a fixed parameter study.
- Perform *partial* relaxations by ignoring only some parts (e.g., delete effects) of the model.
- Take dependencies between actions into account (they can be represented as graphs, the properties of which can be exploited).

Undecidability Proof

Theorem

HTN planning is undedicable.

Proof:
Reduction from the language intersection problem of two context-free grammars: given G and G', is there a word ω in both languages $L(G) \cap L(G')$?

- Construct an HTN planning problem $\mathcal{P}$ that has a solution if and only if the correct answer is *yes*.
- Translate the production rules to decomposition methods. That way only words in $L(G)$ and $L(G')$ can be produced.
- Any solution tn contains the word ω – encoded as action sequence – twice: once produced by G and once produced by G'. The action encodings ensure that no other task networks are executable.

Undecidability Proof, cont'd – by Example

Proof idea by example:

$$\text{Let } G = (\overbrace{\Gamma = \{H, Q\}}^{\text{non-terminal symbols}}, \overbrace{\Sigma = \{a, b\}}^{\text{terminal symbols}}, \overbrace{R}^{\text{production rules}}, \overbrace{H}^{\text{start symbol}})$$

and $G' = (\Gamma' = \{D, F\}, \Sigma' = \{a, b\}, R', D)$ be formal grammars.

Production rules R: $H \mapsto aQb$ $\quad Q \mapsto aQ \mid bQ \mid a \mid b$

Production rules R': $D \mapsto aFD \mid ab$ $\quad F \mapsto a \mid b$

Undecidability Proof, cont'd – by Example

Proof idea by example:

Contructed HTN problem with desired solution set:

$$\mathcal{P} = (V, \overbrace{\{H, Q, D, F\}}^{C}, \overbrace{\{a, b, a', b'\}}^{P}, \delta, M, \overbrace{\{v_{turn:G}\}}^{\text{initial state}}, tn_I, \overbrace{\{v_{turn:G}\}}^{\text{goal description}})$$

$$V = \{v_{turn:G}, v_{turn:G'}\} \cup \{v_a, v_b\}$$

$$\begin{aligned}\delta = \{&(a, (\{v_{turn:G}\}, \{v_{turn:G'}, v_a\}, \{v_{turn:G}\})),\\ &(b, (\{v_{turn:G}\}, \{v_{turn:G'}, v_b\}, \{v_{turn:G}\})),\\ &(a', (\{v_{turn:G'}, v_a\}, \{v_{turn:G}\}, \{v_{turn:G'}, v_a\})),\\ &(b', (\{v_{turn:G'}, v_b\}, \{v_{turn:G}\}, \{v_{turn:G'}, v_b\}))\}\end{aligned}$$

$M = M(G) \cup M(G')$ (translated production rules of G' and G')

$$tn_I = (\underbrace{\{t, t'\}}_{T}, \underbrace{\emptyset}_{\prec}, \underbrace{\{(t, H), (t', D)\}}_{\alpha})$$

Implications of Undecidability

So, HTN planning is undecidable... What does it mean?

- There cannot be a single algorithm that terminates with the correct "answer" (i.e., a solution or *fail*, meaning that no solution exists) for every possible problem.
- But are there *any* termination guarantees?
- That is: could it be that an algorithm never terminates *independent* of whether there is a solution?
- *In principle*, according to the result shown so far: yes.
- However, for HTN planning: no! In case there is a solution we can prove this eventually (we just never know when, i.e., whether this is still going to happen).
- In other words: HTN planning is also(!) semi-decidable. *undecidable + semi-decidable* is also called *strictly semi-decidable*.

Semi-decidability Proof, cont'd

Theorem

HTN planning is semi-decidable.

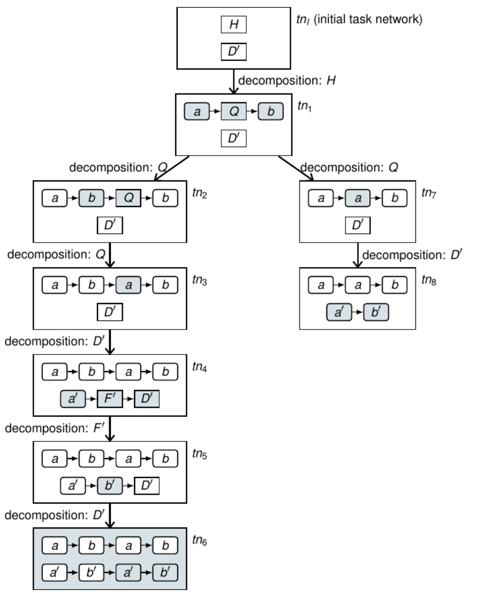
Proof:
Reminder: We need to find a function $\chi_N : M \to \{undef, 1\}$ with:

$$\chi_N(n) = \begin{cases} 1 & \text{if } n \in N \\ undef & \text{otherwise} \end{cases}$$

(Here, M is the set of all HTN planning problems. N is its subset of problems with a solution.)

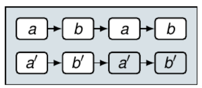
Let $n = \mathcal{P}$. Define χ_N as a BFS procedure (starting with the initial task network) that returns 1 if and only if it discovered a solution to $\mathcal{P}$ (we can also return *undef* in case it can prove it to be unsolvable).

Influence of Task Insertion

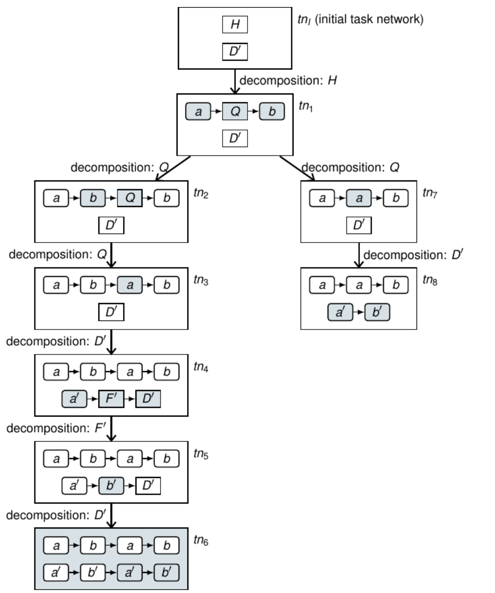


Recap: A task network is a solution if it contains the same word ω twice.

Task network tn_6 is a solution!

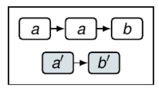


Influence of Task Insertion

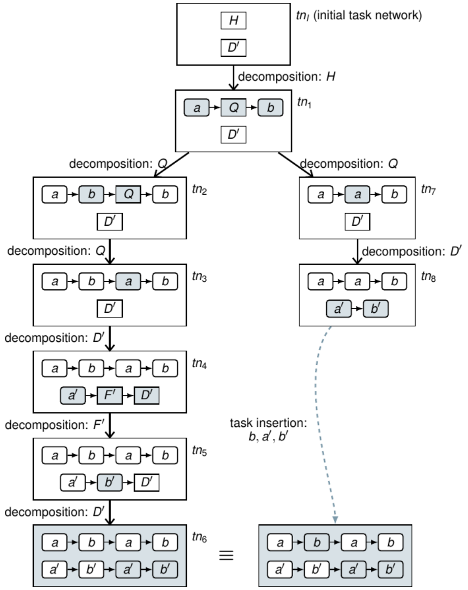


Recap: A task network is a solution if it contains the same word ω twice.

Task network tn_8 is no solution!

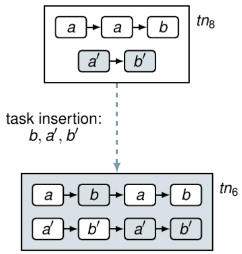


Influence of Task Insertion

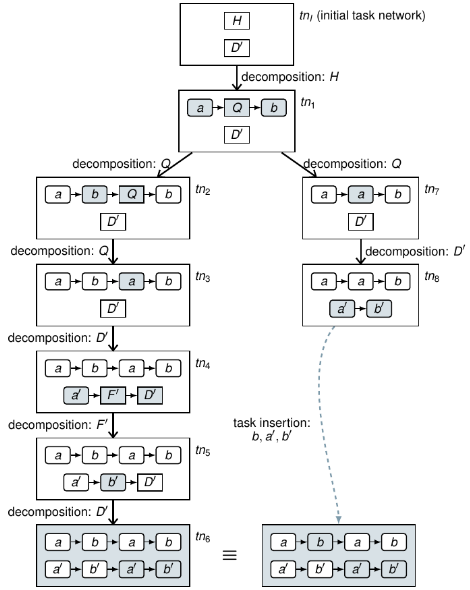


Recap: A task network is a solution if it contains the same word ω twice.

Influence of task insertion:



Influence of Task Insertion

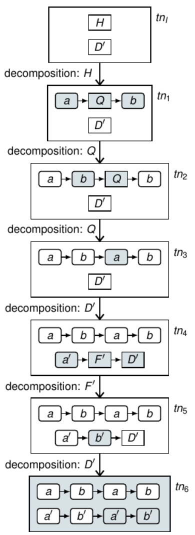


Recap: A task network is a solution if it contains the same word ω twice.

Observation:

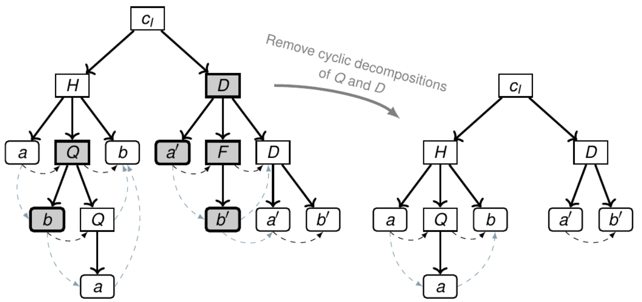
In TIHTN planning, recursion is not required.

Complexity of TIHTN Planning (Membership)

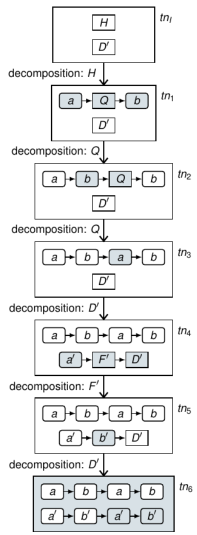


Theorem: TIHTN planning is in **NEXPTIME**

Idea: Restrict to *acyclic* decompositions, fill the rest with task insertion, and verify.



Complexity of TIHTN Planning (Membership)



Theorem: TIHTN planning is in **NEXPTIME**

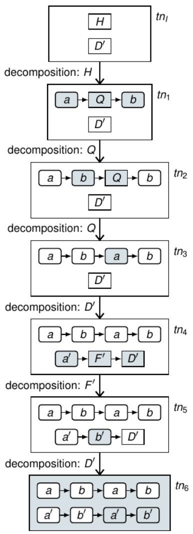
1. Step: Guess an acyclic decomposition:

The guessed decomposition tree describes at most $b^{|C|+1}$ decompositions.

(C = set of compound tasks)
(b = size of largest task network in the model)

Verify in $O(b^{|C|+1})$ whether the tree describes a correct sequence of decompositions.

Complexity of TIHTN Planning (Membership)



Theorem: TIHTN planning is in **NEXPTIME**

2. Step: Guess the actions and orderings to be inserted.

The (guessed) decomposition tree results into a task network with at most $\leq b^{|C|+1}$ tasks.

Between each two actions, at most $2^{|V|}$ actions need to be inserted to achieve the next precondition.

($|V|$ = number of state variables)

Complexity of TIHTN Planning (Hardness)

- We can show that the previous bound is *tight*, i.e., TIHTN planning is $\mathbb{NEXPTIME}$*-complete*.
- To show hardness, we reduce a non-deterministic (exponential)time-bounded Turing Machine to TIHTN planning.
- The proof is not provided in this lecture.

Implications of TIHTN Results

- Recursive models are equivalent to their non-recursive versions.
- None of the restrictions of the hierarchy matters for TIHTN problems.
- TIHTN problems are less expressive than HTN problems (also cf. language results).

Complexity of Totally Ordered TIHTN Planning

Theorem

Deciding whether a totally ordered TIHTN planning problem has a solution is $\mathbb{NEXPTIME}$*-complete*.

Proof, Membership:
Like before, but now, we need to guess less (the order is already given).

Proof, Hardness:
The previous reduction already used a totally ordered TIHTN problem.

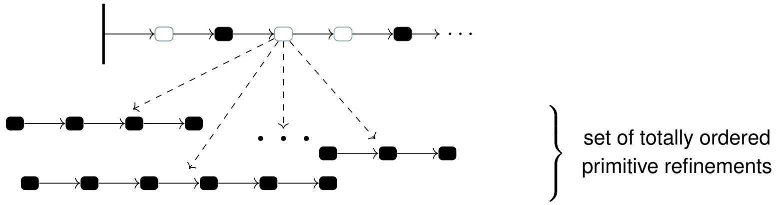
Complexity of Totally Ordered HTN Planning

Theorem

Deciding whether a totally ordered HTN planning problem has a solution is $\mathbb{EXPTIME}$-*complete*.

Intuition of Membership:

- Since plans are totally ordered, the only means of choosing the right refinement for a given compound task is to produce a suitable successor state.



- There are only finitely many states that can be produced by the refinements of a given compound task.

Complexity of Totally Ordered HTN Planning (Memberhsip)

Theorem

Deciding whether a totally ordered HTN planning problem has a solution is $\mathbb{EXPTIME}$-*complete*.

Proof, Membership:

- Create a table $2^V \times (C \cup P) \times 2^V \times \{\top, \bot, ?\}$ to store:
  - s, p, s', x with $x \in \{\top, \bot\}$ to express whether the primitive task p is applicable in s creating a state satisfying s'.
  - s, c, s', x with $x \in \{\top, \bot\}$ to express whether the compound task c has a primitive refinement that is applicable in s creating a state satisfying s'.
- Algorithm:
  - Initialize the table (with all states and tasks) with value ?.
  - Perform bottom-up approach: start with all primitive tasks, then continue with all compound tasks that admit a primitive refinement.
  - Continue as long as at least one value ? is changed.

Complexity of Totally Ordered HTN Planning (Hardness)

Theorem

Deciding whether a totally ordered HTN planning problem has a solution is $\mathbb{EXPTIME}$-*complete*.

Proof, Hardness:

- We reduce from a 2-player game, which is $\mathbb{EXPTIME}$-*complete*.
- The proof is not provided in this lecture.

Complexity of Acyclic HTN Planning (Membership)

Theorem

Deciding whether an acyclic HTN planning problem has a solution is $\mathbb{NEXPTIME}$-*complete*.

Proof, Membership:

Do the same as for TIHTN problems, but without the task insertion part:

- Guess at most $b^{|C|+1}$ decompositions.
  (C = set of compound tasks.)
  (b = size of largest task network in the model.)
- Verify in $O(b^{|C|+1})$ whether the decompositions can be applied in sequence.
- Guess a linearization of the resulting task network.
- Verify applicability of resulting linearization in $O(b^{|C|+1})$.

Complexity of Acyclic HTN Planning (Hardness)

Theorem

Deciding whether an acyclic HTN planning problem has a solution is $\mathbb{NEXPTIME}$*-complete*.

Proof, Hardness:

- Almost the same proof as to TIHTN planning: We reduce from a non-deterministic turing machine, but now don't allow task insertion.
- The proof is not provided in this lecture.

Complexity of Regular HTN Planning (Membership)

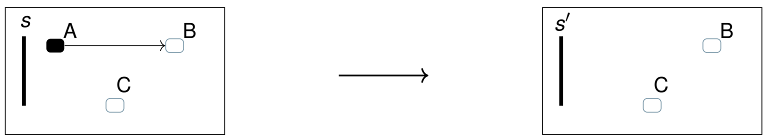
Theorem

Deciding whether a regular HTN planning problem has a solution is $\mathbb{PSPACE}$*-complete*.

Proof, Membership:

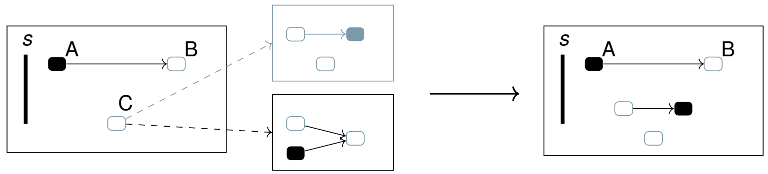
- Rely on progression search.
- Until the compound task gets decomposed, all primitive tasks have been "progressed away".
- That way, the size of any task network is bounded by the size of the largest task network in the model.

Recap and Example: HTN Progression Search



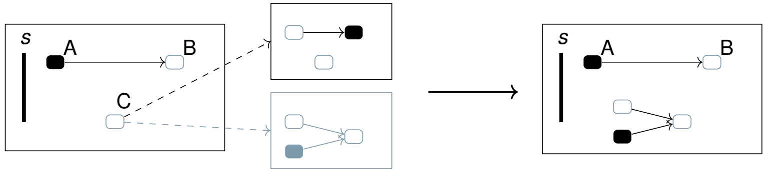
- Always progress tasks that are a possibly first task in the network.
- Here, these are the tasks *A* and *C*.
- In case the chosen task to progress next is:
  - primitive: apply it and progress the state.

Recap and Example: HTN Progression Search



- Always progress tasks that are a possibly first task in the network.
- Here, these are the tasks *A* and *C*.
- In case the chosen task to progress next is:
  - primitive: apply it and progress the state.
  - compound: decompose it.

Recap and Example: HTN Progression Search



- Always progress tasks that are a possibly first task in the network.
- Here, these are the tasks *A* and *C*.
- In case the chosen task to progress next is:
  - primitive: apply it and progress the state.
  - compound: decompose it.

Complexity of Regular HTN Planning (Hardness)

Theorem

Deciding whether a regular HTN planning problem has a solution is $\mathbb{PSPACE}$-*complete*.

Proof, Hardness: Every STRIPS problem $\mathcal{P}_{STRIPS}$ can be canonically expressed by a totally ordered regular HTN problem $\mathcal{P}$:

- The actions in $\mathcal{P}_{STRIPS}$ are primitive tasks in $\mathcal{P}$.
- There is just one compound task X generating all possible action sequences: for all $p \in P$, we have a method mapping X to p followed by X.
- For the base case, we have a method mapping X to an artificial primitive task encoding the goal description.
- The initial task is X.

Complexity of Tail-Recursive HTN Planning (Membership)

Theorem

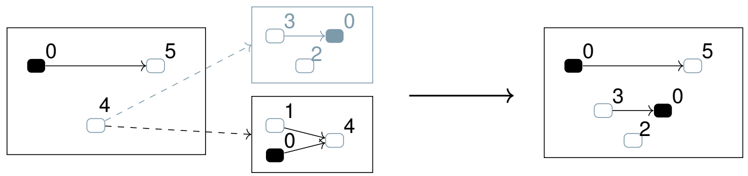
Deciding whether a tail-recursive HTN planning problem has a solution is $\mathbb{EXPSPACE}$-*complete*.

Proof, Membership:

- Again, rely on progression search. Until the last task gets decomposed, all tasks ordered before it have been "progressed away".
- Only the decomposition of a last task might let the current stratification height unchanged.
- The decomposition of non-last tasks results into tasks of strictly lower stratum.
- From this, we can calculate a *progression bound* – a maximal size of task network created under progression.
- We get progression bound $k \cdot m^h$, with k size of initial task network, m size of the largest method, and h stratification height.

Recap and Example: Progression Search with Tail-Recursive HTNs

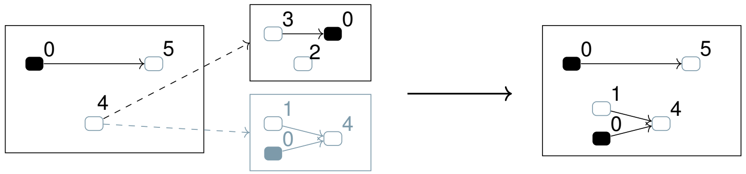
Consider the following initial task network of size 3:



- Using a method *without* last task increases the size,
- but "such decompositions" can only occur finitely often (limited by the stratification height).

Recap and Example: Progression Search with Tail-Recursive HTNs

Consider the following initial task network of size 3:



- Using a method *with* last task increases the size,
- and a task with the same stratification height remains(!),
- but "this can not increase the size arbitrarily", because the tasks ordered before it have to be progressed away before the remaining task can be decomposed again.

Complexity of Tail-Recursive HTN Planning (Hardness)

Theorem

Deciding whether a tail-recursive HTN planning problem has a solution is $\mathbb{EXPSPACE}$*-complete*.

Proof, Hardness:

- To show hardness, we reduce a (exponential)space-bounded Turing Machine to HTN planning.
- The proof is not provided in this lecture.

Summary

- We studied the computational complexity of the *plan existence problem*.
- It ranges from $\mathbb{P}$ up to undecidable:
  - In HTN planning, structural properties have a large impact on the computational complexity.
  - In TIHTN planning, they do not: Task insertion eliminates the need for recursion.
- Complexity results give raise to specialized algorithms, to heuristics, and to translations to other problem classes.