Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning

Pascal Bercher and Daniel Höller

Institute of Artificial Intelligence,
Ulm University, Germany

June 25th, ICAPS 2018 (Delft)

ulm university universität uulm

Everyone (beginner or expert) who is familiar with the basics of classical planning!

We assume prior knowledge about:

- standard problem definition and semantics of classical planning

Everyone (beginner or expert) who is familiar with the basics of classical planning!

We assume prior knowledge about:

- standard problem definition and semantics of classical planning
- heuristics, esp. delete relaxation

Everyone (beginner or expert) who is familiar with the basics of classical planning!

We assume prior knowledge about:

- standard problem definition and semantics of classical planning
- heuristics, esp. delete relaxation
- search strategies (A*, greedy, etc.)

Everyone (beginner or expert) who is familiar with the basics of classical planning!

We assume prior knowledge about:

- standard problem definition and semantics of classical planning
- heuristics, esp. delete relaxation
- search strategies (A*, greedy, etc.)
- basic complexity theory (Chomsky hierarchy, automata, etc.)

Obtain most relevant basics of the most-commonly known hierarchical planning formalization – HTN planning:

Obtain most relevant basics of the most-commonly known hierarchical planning formalization – HTN planning:

- understand the core differences to non-hierarchical (classical) planning: *HTN planning is **not** (just) a planning technique!*

Obtain most relevant basics of the most-commonly known hierarchical planning formalization – HTN planning:

- understand the core differences to non-hierarchical (classical) planning: *HTN planning is **not** (just) a planning technique!*
- learn basic theoretical properties of HTN planning: hardness of the problem(s), expressivity

Obtain most relevant basics of the most-commonly known hierarchical planning formalization – HTN planning:

- understand the core differences to non-hierarchical (classical) planning: *HTN planning is **not** (just) a planning technique!*
- learn basic theoretical properties of HTN planning: hardness of the problem(s), expressivity
- learn the most important solving techniques

Obtain most relevant basics of the most-commonly known hierarchical planning formalization – HTN planning:

- understand the core differences to non-hierarchical (classical) planning: *HTN planning is **not** (just) a planning technique!*
- learn basic theoretical properties of HTN planning: hardness of the problem(s), expressivity
- learn the most important solving techniques
- obtain some ideas on how to design heuristics (why is it more complicated than in non-hierarchical planning?)

Part I: Theoretical Foundations

- Problem Definition(s)
- Computational Complexity of Plan Existence Problem
- Expressivity Analysis

Part II: Practice

- Solution Techniques
- Heuristics
- Excursion: Further Hierarchical Planning Formalisms

Overview Part I

Theoretical Foundations

- **Introduction**
- Problem Definition
- Computational Complexity of the Plan Existence Problem
  - General HTN Planning
  - HTN Planning with Task Insertion
  - Totally Ordered HTN Planning
  - Restricting Recursion (Acyclic, Regular, Tail-recursive)
- Expressivity Analysis

Classical Planning (Recap)

Environment:

- Fully observable
- Discrete (no time or resources)
- Deterministic
- Single-agent
- *Just one kind of action!*

Classical Planning (Recap)

Environment:

- Fully observable
- Discrete (no time or resources)
- Deterministic
- Single-agent
- *Just one kind of action!*

Planning:

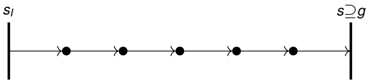
- Offline
- Usually ground and via progression search
- Solutions are action sequences

Classical Planning (Recap, cont'd)

Problem formalization, $\mathcal{P} = (V, s_I, A, g)$:

- Set of state variables V
- Initial state $s_I \in 2^V$
- Set of actions A, $a \in A$ has the form $(prec, add, del) \in (2^V)^3$
  An action $(prec, add, del)$ is executable in a state $s \in 2^V$ iff $prec \subseteq s$. Its application to s results into the state $(s \setminus del) \cup add$.
  Executability of task sequences defined analogously
- Goal description $g \subseteq V$

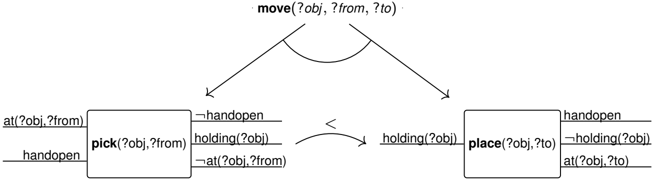
solution:



Informal Problem Definition

What is hierarchical planning, anyway?

Here: the model specifies a ***task hierarchy***: *compound* (or *complex*, *abstract*, *high-level*) tasks need to be decomposed into *primitive tasks*.



Problem given as a compound task (or a set of compound and/or primitive tasks).

Goal: Finding a (primitive) executable refinement.

Motivation

Why relying on a hierarchical model?

- More flexibility with regard to modeling approach: incorporate procedural expert knowledge (just as a modeling means, or to speed up search)

Motivation

Why relying on a hierarchical model?

- More flexibility with regard to modeling approach: incorporate procedural expert knowledge (just as a modeling means, or to speed up search)
- Describe more complex behavior (i.e., pose complex restrictions on the desired solutions)

Motivation

Why relying on a hierarchical model?

- More flexibility with regard to modeling approach: incorporate procedural expert knowledge (just as a modeling means, or to speed up search)
- Describe more complex behavior (i.e., pose complex restrictions on the desired solutions)
- Allow easier user integration in the plan generation process (mixed initiative planning; MIP)

Motivation

Why relying on a hierarchical model?

- More flexibility with regard to modeling approach: incorporate procedural expert knowledge (just as a modeling means, or to speed up search)
- Describe more complex behavior (i.e., pose complex restrictions on the desired solutions)
- Allow easier user integration in the plan generation process (mixed initiative planning; MIP)
- Communicate plans on different levels of abstraction

Motivation

Why relying on a hierarchical model?

- More flexibility with regard to modeling approach: incorporate procedural expert knowledge (just as a modeling means, or to speed up search)
- Describe more complex behavior (i.e., pose complex restrictions on the desired solutions)
- Allow easier user integration in the plan generation process (mixed initiative planning; MIP)
- Communicate plans on different levels of abstraction
- Incorporate task abstraction in plan explanations

Overview Part I

Theoretical Foundations

- Introduction
- **Problem Definition**
- Computational Complexity of the Plan Existence Problem
  - General HTN Planning
  - HTN Planning with Task Insertion
  - Totally Ordered HTN Planning
  - Restricting Recursion (Acyclic, Regular, Tail-recursive)
- Expressivity Analysis

HTN Planning vs. Classical Planning

"HTN planners differ from classical planners in what they plan for and how they plan for it. In an HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks."

(Ghallab, Nau, and Traverso; Automated Planning: Theory and Practice)

HTN Planning vs. Classical Planning

"HTN planners differ from classical planners in what they plan for and how they plan for it. In an HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks."

(Ghallab, Nau, and Traverso; Automated Planning: Theory and Practice)

Main differences to classical planning problems:

- It's not about generating some goal state! The goal is find a refinement of the initial task(s), not to satisfy some goal description

HTN Planning vs. Classical Planning

"HTN planners differ from classical planners in what they plan for and how they plan for it. In an HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks."

(Ghallab, Nau, and Traverso; Automated Planning: Theory and Practice)

Main differences to classical planning problems:

- It's not about generating some goal state! The goal is find a refinement of the initial task(s), not to satisfy some goal description
- There is no arbitrary task insertion: to alter task networks, we need to decompose compound tasks using their pre-defined methods

Problem Definition & Solution Criteria

$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables

Problem Definition & Solution Criteria

primitive tasks

compound tasks

$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \rightarrow (2^V)^3$ the task name mapping
- C a set of compound task names

Problem Definition & Solution Criteria

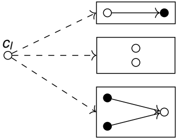
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \rightarrow (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task

A solution task network *tn* must:

- be a refinement of c_I,

Problem Definition & Solution Criteria



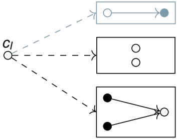
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \to (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and

Problem Definition & Solution Criteria



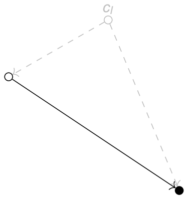
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \rightarrow (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and

Problem Definition & Solution Criteria



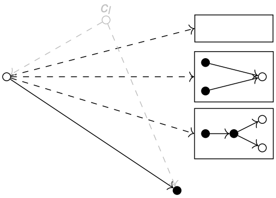
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \to (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and

Problem Definition & Solution Criteria



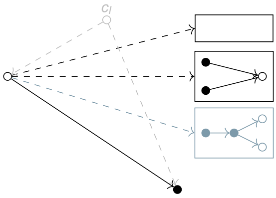
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \to (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and

Problem Definition & Solution Criteria



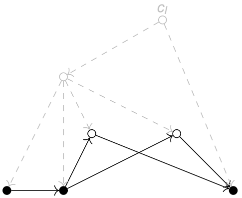
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \to (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and

Problem Definition & Solution Criteria



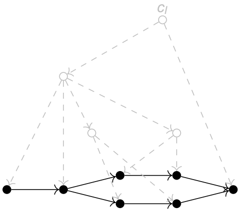
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \to (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and

Problem Definition & Solution Criteria



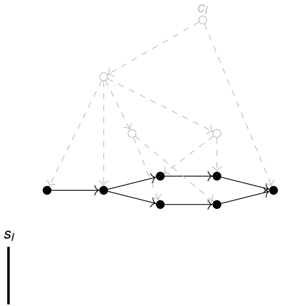
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \to (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and

Problem Definition & Solution Criteria



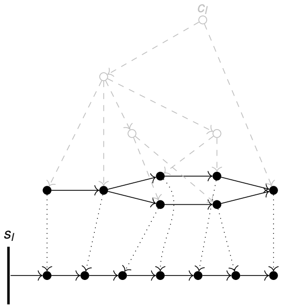
$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \rightarrow (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods
- $s_I \subseteq V$ the initial state

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and

Problem Definition & Solution Criteria



$$\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$$

- V a set of state variables
- P a set of primitive task names
- $\delta : P \to (2^V)^3$ the task name mapping
- C a set of compound task names
- $c_I \in C$ the initial task
- $M \subseteq C \times 2^{TN}$ the methods
- $s_I \subseteq V$ the initial state

A solution task network *tn* must:

- be a refinement of c_I,
- only contain primitive tasks, and
- have an executable linearization.

Problem Definition & Solution Criteria (Cont'd)

More formally:

- For the sake of simplicity, we present a *ground* formalism, but most results exist for lifted planning as well

Problem Definition & Solution Criteria (Cont'd)

More formally:

- For the sake of simplicity, we present a *ground* formalism, but most results exist for lifted planning as well
- Task network: $tn = (T, \prec, \alpha)$ consists of:
  - T, a possibly empty set of *tasks* or *task identifier symbols*
  - $\prec$, a partial order on the tasks
  - $\alpha : T \rightarrow P \dot{\cup} C$, the task mapping function

  Primitive task names are mapped to their tuples by the task name mapping $\delta : P \rightarrow (2^V)^3$

Problem Definition & Solution Criteria (Cont'd)

More formally:

- For the sake of simplicity, we present a *ground* formalism, but most results exist for lifted planning as well
- Task network: $tn = (T, \prec, \alpha)$ consists of:
  - T, a possibly empty set of *tasks* or *task identifier symbols*
  - $\prec$, a partial order on the tasks
  - $\alpha : T \to P \dot{\cup} C$, the task mapping function

  Primitive task names are mapped to their tuples by the task name mapping $\delta : P \to (2^V)^3$
- A task network is called *executable* if there exists an executable linearization of its tasks

Problem Definition & Solution Criteria (Cont'd)

More formally:

- For the sake of simplicity, we present a *ground* formalism, but most results exist for lifted planning as well
- Task network: $tn = (T, \prec, \alpha)$ consists of:
  - T, a possibly empty set of *tasks* or *task identifier symbols*
  - $\prec$, a partial order on the tasks
  - $\alpha : T \to P \dot{\cup} C$, the task mapping function

  Primitive task names are mapped to their tuples by the task name mapping $\delta : P \to (2^V)^3$
- A task network is called *executable* if there exists an executable linearization of its tasks
- Let p be a primitive task (name) and $\delta(p) = (prec, add, del)$. Then, p is called *executable* in state $s \in 2^V$ iff $prec \subseteq s$. Its application to s results into the state $(s \setminus del) \cup add$. Executability of task sequences defined analogously

Problem Definition & Solution Criteria (Cont'd)

More formally:

- A decomposition method $m \in M$ is a tuple $m = (c, tn_m)$ with a compound task c and task network $tn_m = (T_m, \prec_m, \alpha_m)$

Problem Definition & Solution Criteria (Cont'd)

More formally:

- A decomposition method $m \in M$ is a tuple $m = (c, tn_m)$ with a compound task c and task network $tn_m = (T_m, \prec_m, \alpha_m)$
- Let $tn = (T, \prec, \alpha)$ be a task network, $t \in T$ a task identifier, and $\alpha(t) = c$ a compound task to be decomposed by $m = (c, tn_m)$.
  We assume $T \cap T_m = \emptyset$.
  Then, the application of m to tn results into the task network $tn' = ((T \setminus \{t\}) \cup T_m, \prec \cup \prec_m \cup \prec_X, \alpha \cup \alpha_m)|_{(T\setminus\{t\})\cup T_m}$ with:

$$\begin{aligned}\prec_X :=& \{(t', t'') \mid (t', t) \in \prec, t'' \in T_m\} \cup \\ & \{(t'', t') \mid (t, t') \in \prec, t'' \in T_m\}\end{aligned}$$

  where $(X_1, \ldots, x_n)|_Y$ restricts the sets X_i to elements in Y

Problem Definition & Solution Criteria (Cont'd)

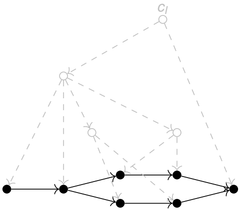
More formally:

- A task network *tn* is a solution if and only if:

Problem Definition & Solution Criteria (Cont'd)

More formally:

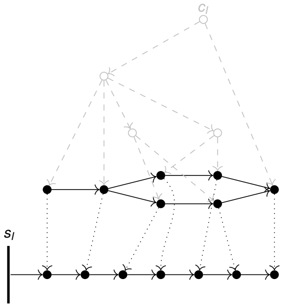
- A task network *tn* is a solution if and only if:
  - There is a sequence of decomposition methods $\overline{m}$ that transforms c_I into *tn*,
  - *tn* contains only primitive tasks, and



Problem Definition & Solution Criteria (Cont'd)

More formally:

- A task network *tn* is a solution if and only if:
  - There is a sequence of decomposition methods $\overline{m}$ that transforms c_I into *tn*,
  - *tn* contains only primitive tasks, and
  - the (still partially ordered) task network *tn* admits an executable linearization $\overline{t}$ of its tasks.



Overview

Which formalization choices do exist?
Which impact do they have?

Overview

Which formalization choices do exist?
Which impact do they have?

- Initial task network vs. a single initial task

Overview

Which formalization choices do exist?
Which impact do they have?

- Initial task network vs. a single initial task
- Adding a goal description

Overview

Which formalization choices do exist?
Which impact do they have?

- Initial task network vs. a single initial task
- Adding a goal description
- Adding state constraints

Impact of Initial Task Network

Recap: $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ describes an HTN planning problem as described before.

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, \boldsymbol{tn_I})$ be an **HTN planning problem with initial task network $\boldsymbol{tn_I}$**.

Impact of Initial Task Network

Recap: $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ describes an HTN planning problem as described before.

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, \boldsymbol{tn_I})$ be an **HTN planning problem with initial task network** $\boldsymbol{tn_I}$.

Then, a task network tn is a solution if and only if:

- There is a sequence of decomposition methods $\overline{m}$ **that transforms** $\boldsymbol{tn_I}$ **into** $\boldsymbol{tn}$,
- tn contains only primitive tasks, and
- the (still partially ordered) task network tn admits an executable linearization $\overline{t}$ of its tasks.

Impact of Initial Task Network (Cont'd)

Theorem: Initial task networks can be compiled away.

Impact of Initial Task Network (Cont'd)

Theorem: **Initial task networks can be compiled away.**

Proof:

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, \boldsymbol{tn_I})$ be an HTN planning problem with initial task network tn_I.

Then, there is an HTN planning problem $\mathcal{P}' = (V, P, \delta, \boldsymbol{C'}, \boldsymbol{M'}, s_I, \boldsymbol{c_I})$ with the same set of solutions:

Let $C' := C \mathbin{\dot\cup} \{c_I\}$ and $M' := M \cup \{(c_I, tn_I)\}$.

Identical solution set is obvious.

Impact of Goal Description

Recap: $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ describes an HTN planning problem as described before.

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, c_I, \boldsymbol{g})$ be an **HTN planning problem with goal description $\boldsymbol{g} \subseteq V$**.

Impact of Goal Description

Recap: $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ describes an HTN planning problem as described before.

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, c_I, \boldsymbol{g})$ be an **HTN planning problem with goal description $\boldsymbol{g} \subseteq V$.**

Then, a task network *tn* is a solution if and only if:

- There is a sequence of decomposition methods $\overline{m}$ that transforms c_I into *tn*,
- *tn* contains only primitive tasks,
- the (still partially ordered) task network *tn* admits an executable linearization $\overline{t}$ of its tasks, and
- **the task sequence $\overline{t}$ generates a goal state $\boldsymbol{s} \supseteq \boldsymbol{g}$.**

Impact of Goal Description (Cont'd)

Theorem: Goal descriptions can be compiled away.

Impact of Goal Description (Cont'd)

Theorem: Goal descriptions can be compiled away.

Proof:

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, c_I, \boldsymbol{g})$ be an HTN planning problem with goal description.

Then, there is an HTN planning problem $\mathcal{P}' = (V, \boldsymbol{P'}, \boldsymbol{\delta'}, C, M, s_I, \boldsymbol{tn_I})$ with the same set of solutions:

Here, tn_I contains two tasks: c_I followed by a new primitive task p with no effects and g as precondition, $\delta(p) = (g, \emptyset, \emptyset)$.

Impact of Goal Description (Cont'd)

Theorem: Goal descriptions can be compiled away.

Proof:

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, c_I, \boldsymbol{g})$ be an HTN planning problem with goal description.

Then, there is an HTN planning problem $\mathcal{P}' = (V, \boldsymbol{P'}, \boldsymbol{\delta'}, C, M, s_I, \boldsymbol{tn_I})$ with the same set of solutions:

Here, tn_I contains two tasks: c_I followed by a new primitive task p with no effects and g as precondition, $\delta(p) = (g, \emptyset, \emptyset)$.

Then, the initial task network in $\mathcal{P}'$ can be compiled away as before.

Identical solution set is obvious.

State Constraints in HTN Planning

State constraints have been introduced in the HTN formalization by Erol et al. (1994):

- (l, t), the literal l holds immediately before task t
- (t, l), the literal l holds immediately after task t
- (t, l, t'), the literal l holds in all states between t and t'

State Constraints in HTN Planning

State constraints have been introduced in the HTN formalization by Erol et al. (1994):

- (l, t), the literal l holds immediately before task t
- (t, l), the literal l holds immediately after task t
- (t, l, t'), the literal l holds in all states between t and t'

In case t, resp. t', are compound, a constraint (l, t) is, upon decomposition, translated to $(l, \mathit{first}[t_1, \ldots, t_n])$, where the t_i are all sub tasks of t. ((t, l) and (t, l, t') are handled analogously.)

State Constraints in HTN Planning

State constraints have been introduced in the HTN formalization by Erol et al. (1994):

- (l, t), the literal l holds immediately before task t
- (t, l), the literal l holds immediately after task t
- (t, l, t'), the literal l holds in all states between t and t'

In case t, resp. t', are compound, a constraint (l, t) is, upon decomposition, translated to $(l, \mathit{first}[t_1, \ldots, t_n])$, where the t_i are all sub tasks of t. ((t, l) and (t, l, t') are handled analogously.)

Notably: Erol et al.'s formalization specifies a boolean constraint formula, in which *state*, *variable*, and *ordering constraints* can be specified with negations and disjunctions.

State Constraints in HTN Planning

State constraints have been introduced in the HTN formalization by Erol et al. (1994):

- (l, t), the literal l holds immediately before task t
- (t, l), the literal l holds immediately after task t
- (t, l, t'), the literal l holds in all states between t and t'

In case t, resp. t', are compound, a constraint (l, t) is, upon decomposition, translated to $(l, \mathit{first}[t_1, \ldots, t_n])$, where the t_i are all sub tasks of t. ((t, l) and (t, l, t') are handled analogously.)

Notably: Erol et al.'s formalization specifies a boolean constraint formula, in which *state*, *variable*, and *ordering constraints* can be specified with negations and disjunctions.

No compilation known yet.

Overview Part I

Theoretical Foundations

- Introduction
- Problem Definition
- **Computational Complexity of the Plan Existence Problem**
  - General HTN Planning
  - HTN Planning with Task Insertion
  - Totally Ordered HTN Planning
  - Restricting Recursion (Acyclic, Regular, Tail-recursive)
- Expressivity Analysis

Definition & Motivation

The HTN plan existence problem is defined as follows:

Given an HTN planning problem $\mathcal{P}$, does $\mathcal{P}$ possess a solution?

The HTN plan existence problem is defined as follows:

Given an HTN planning problem $\mathcal{P}$, does $\mathcal{P}$ possess a solution?

Motivation for studying this problem

- Deeper problem understanding

Definition & Motivation

The HTN plan existence problem is defined as follows:

Given an HTN planning problem $\mathcal{P}$, does $\mathcal{P}$ possess a solution?

Motivation for studying this problem

- Deeper problem understanding
- Development of problem relaxations (heuristics) and specialized algorithms

Definition & Motivation

The HTN plan existence problem is defined as follows:

Given an HTN planning problem $\mathcal{P}$, does $\mathcal{P}$ possess a solution?

Motivation for studying this problem

- Deeper problem understanding
- Development of problem relaxations (heuristics) and specialized algorithms
- Development of problem compilations

Overview Part I

Theoretical Foundations

- Introduction
- Problem Definition
- **Computational Complexity of the Plan Existence Problem**
  - **General HTN Planning**
  - HTN Planning with Task Insertion
  - Totally Ordered HTN Planning
  - Restricting Recursion (Acyclic, Regular, Tail-recursive)
- Expressivity Analysis

Undecidability Proof

Theorem: HTN planning is undecidable.

Proof:

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

Undecidability Proof

Theorem: HTN planning is undecidable.

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*

Undecidability Proof

Theorem: HTN planning is undecidable.

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

Undecidability Proof

Theorem: HTN planning is undecidable.

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)

Theorem: HTN planning is undecidable.

Proof: (Cont'd, by example)

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

Production rules R:	$H \mapsto aQb$	$Q \mapsto aQ \mid bQ \mid a \mid b$
Production rules R':	$D \mapsto aFD \mid ab$	$F \mapsto a \mid b$

Undecidability Proof (Cont'd)

Theorem: HTN planning is undecidable.

Proof: (Cont'd, by example)

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

Undecidability Proof (Cont'd)

Theorem: HTN planning is undecidable.

Proof: (Cont'd, by example)

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

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

Undecidability Proof (Cont'd)

Theorem: HTN planning is undecidable.

Proof: (Cont'd, by example)

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

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

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

Undecidability Proof (Cont'd)

Theorem: HTN planning is undecidable.

Proof: (Cont'd, by example)

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

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

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

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

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

Overview

Which properties make the plan existence problem easier?

- Task insertion,

Overview

Which properties make the plan existence problem easier?

- Task insertion,
- Total order of all task networks,

Overview

Which properties make the plan existence problem easier?

- Task insertion,
- Total order of all task networks,
- Recursion. Methods are:

Overview

Which properties make the plan existence problem easier?

- Task insertion,
- Total order of all task networks,
- Recursion. Methods are:
  - *acyclic*: no recursion

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

Overview

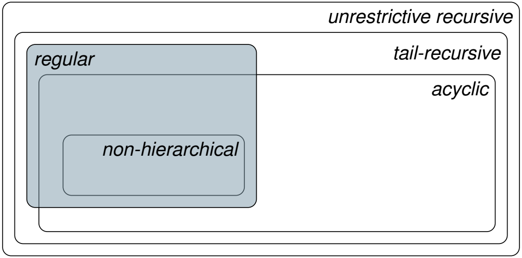
Which properties make the plan existence problem easier?

- 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

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



Overview Part I

Theoretical Foundations

- Introduction
- Problem Definition
- **Computational Complexity of the Plan Existence Problem**
  - General HTN Planning
  - **HTN Planning with Task Insertion**
  - Totally Ordered HTN Planning
  - Restricting Recursion (Acyclic, Regular, Tail-recursive)
- Expressivity Analysis

Problem Definition

In *HTN planning with task insertion*, *TIHTN planning*, tasks may be added arbitrarily to task networks (not just via decomposition):

Let $\mathcal{P}^{\star} = (V, P, \delta, C, M, s_I, c_I)$ be a **TIHTN planning problem**.

Problem Definition

In *HTN planning with task insertion*, *TIHTN planning*, tasks may be added arbitrarily to task networks (not just via decomposition):

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, c_I)$ be a **TIHTN planning problem**.

Then, a task network *tn* is a solution if and only if:

- There is a sequence of decomposition methods $\overline{m}$ **and task insertions** that transforms c_I into *tn*,
- *tn* contains only primitive tasks, and
- the (still partially ordered) task network *tn* admits an executable linearization $\overline{t}$ of its tasks.

Problem Definition

In *HTN planning with task insertion*, *TIHTN planning*, tasks may be added arbitrarily to task networks (not just via decomposition):

Let $\mathcal{P}^{\star} = (V, P, \delta, C, M, s_I, c_I)$ be a **TIHTN planning problem**.

Then, a task network *tn* is a solution if and only if:

- There is a sequence of decomposition methods $\overline{m}$ that transforms c_I into $\boldsymbol{tn'}$,
- $\boldsymbol{tn \supseteq tn'}$ **contains all tasks and orderings of** $\boldsymbol{tn'}$,
- *tn* contains only primitive tasks, and
- the (still partially ordered) task network *tn* admits an executable linearization $\overline{t}$ of its tasks.

Motivation

Benefits of allowing task insertion:

- Task insertion plus goal description fully subsumes classical planning (while allowing task hierarchies as well)

Motivation

Benefits of allowing task insertion:

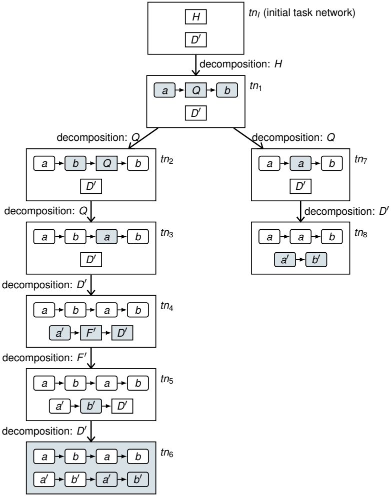
- Task insertion plus goal description fully subsumes classical planning (while allowing task hierarchies as well)
- Task insertion makes the modeling process easier: certain parts can be left to the planner

Motivation

Benefits of allowing task insertion:

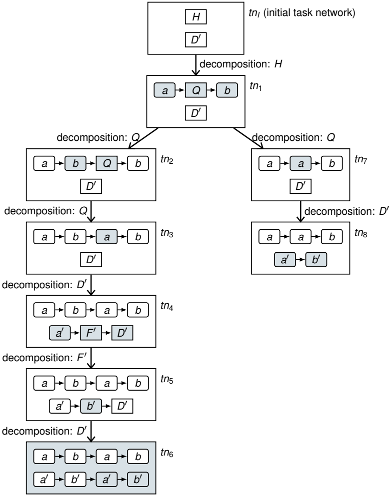
- Task insertion plus goal description fully subsumes classical planning (while allowing task hierarchies as well)
- Task insertion makes the modeling process easier: certain parts can be left to the planner
- Task insertion makes the problem computationally easier (can be exploited for heuristics)

Influence of Task Insertion



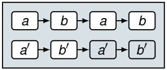
Recap: A task network is a solution if it contains the same word ω twice.

Influence of Task Insertion

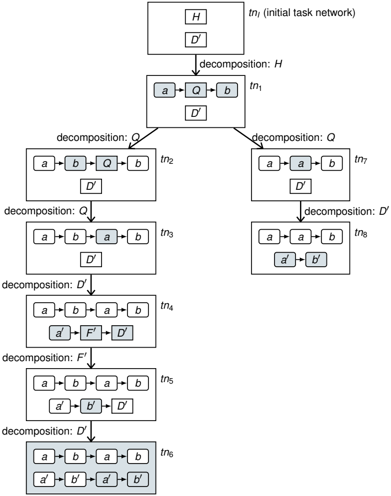


Recap: A task network is a solution if it contains the same word ω twice.

Task network tn_6 is a solution!

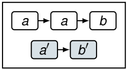


Influence of Task Insertion

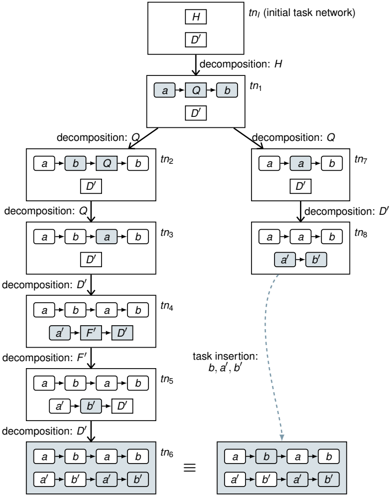


Recap: A task network is a solution if it contains the same word ω twice.

Task network tn_8 is no solution!

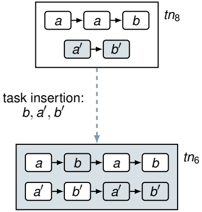


Influence of Task Insertion

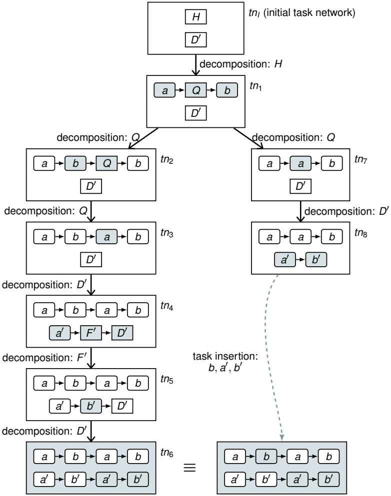


Recap: A task network is a solution if it contains the same word ω twice.

Influence of task insertion:



Influence of Task Insertion

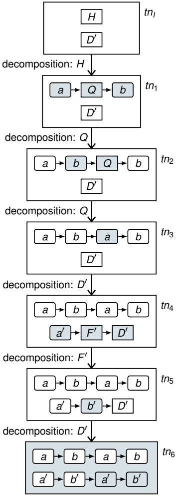


Recap: A task network is a solution if it contains the same word ω twice.

Observation:

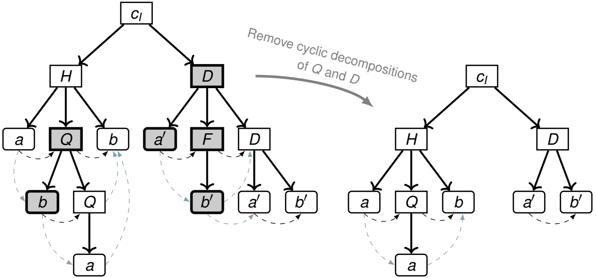
In TIHTN planning, recursion is not required.

Eliminating Recursion

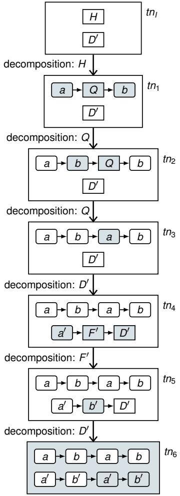


Theorem: TIHTN planning is in **NEXPTIME**

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



Eliminating Recursion



Theorem: TIHTN planning is in **NEXPTIME**

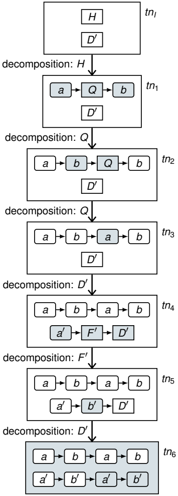
1. Step: Guess an acyclic decomposition:

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.

Eliminating Recursion



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)

Overview Part I

Theoretical Foundations

- Introduction
- Problem Definition
- **Computational Complexity of the Plan Existence Problem**
  - General HTN Planning
  - HTN Planning with Task Insertion
  - **Totally Ordered HTN Planning**
  - Restricting Recursion (Acyclic, Regular, Tail-recursive)
- Expressivity Analysis

Problem Definition

An HTN planning problem $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ 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.

Problem Definition

An HTN planning problem $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ 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.

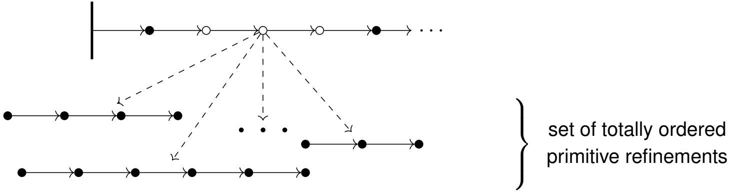
Computational Complexity

Theorem: Totally ordered HTN planning is in **EXPTIME**

Theorem: Totally ordered HTN planning is in **EXPTIME**

Intuition:

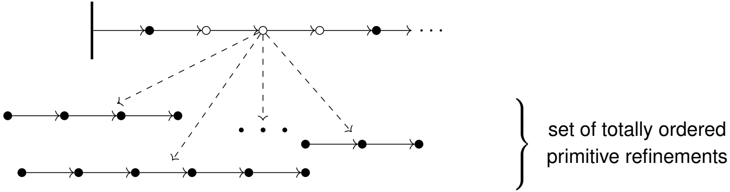
- Since plans are totally ordered, the only means of choosing the right refinement for a given compound task is to produce a suitable successor state



Theorem: Totally ordered HTN planning is in **EXPTIME**

Intuition:

- 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

Computational Complexity

Theorem: Totally ordered HTN planning is in **EXPTIME**

Proof:

- Create a table $2^V \times (C \cup P) \times 2^V \times \{\top, \bot, ?\}$ to store:

Computational Complexity

Theorem: Totally ordered HTN planning is in **EXPTIME**

Proof:

- 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'

Computational Complexity

Theorem: Totally ordered HTN planning is in **EXPTIME**

Proof:

- 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'

Computational Complexity

Theorem: Totally ordered HTN planning is in **EXPTIME**

Proof:

- 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 ?

Computational Complexity

Theorem: Totally ordered HTN planning is in **EXPTIME**

Proof:

- 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

Computational Complexity

Theorem: Totally ordered HTN planning is in **EXPTIME**

Proof:

- 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

Overview Part I

Theoretical Foundations

- Introduction
- Problem Definition
- **Computational Complexity of the Plan Existence Problem**
  - General HTN Planning
  - HTN Planning with Task Insertion
  - Totally Ordered HTN Planning
  - **Restricting Recursion (Acyclic**, Regular, Tail-recursive**)**
- Expressivity Analysis

Problem Definition

An HTN planning problem is called called *acyclic* if no compound task can reach itself via decomposition.

Computational Complexity

Theorem: Acyclic HTN planning is in **NEXPTIME**.

Computational Complexity

Theorem: Acyclic HTN planning is in **NEXPTIME**.

Proof:

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

Computational Complexity

Theorem: Acyclic HTN planning is in **NEXPTIME**.

Proof:

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)

Computational Complexity

Theorem: Acyclic HTN planning is in **NEXPTIME**.

Proof:

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

Computational Complexity

Theorem: Acyclic HTN planning is in **NEXPTIME**.

Proof:

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

Computational Complexity

Theorem: Acyclic HTN planning is in **NEXPTIME**.

Proof:

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 $O(b^{|C|+1})$

Overview Part I

Theoretical Foundations

- Introduction
- Problem Definition
- Computational Complexity of the Plan Existence Problem
  - General HTN Planning
  - HTN Planning with Task Insertion
  - Totally Ordered HTN Planning
  - **Restricting Recursion (**Acyclic, **Regular**, Tail-recursive**)**
- Expressivity Analysis

Problem Definition

- A task network $tn = (T, \prec, \alpha)$ is called *regular* if

Problem Definition

- A task network $tn = (T, \prec, \alpha)$ is called *regular* if
  - at most one task in T is compound and

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.

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.

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.

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

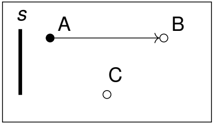
Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Proof:

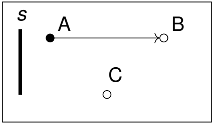
- Rely on progression search

Excursion: HTN Progression Search



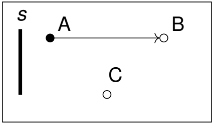
- Always progress tasks that are a possibly first task in the network

Excursion: HTN Progression Search



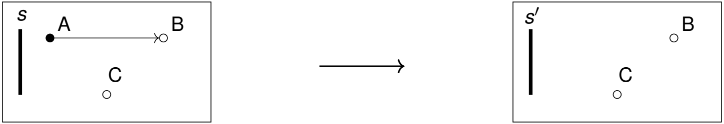
- Always progress tasks that are a possibly first task in the network
- Here, these are the tasks *A* and *C*.

Excursion: HTN Progression Search



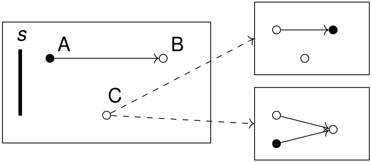
- Always progress tasks that are a possibly first task in the network
- Here, these are the tasks *A* and *C*.
- In case the chosen task to progress next is:

Excursion: HTN Progression Search



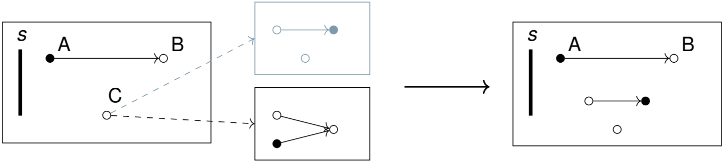
- Always progress tasks that are a possibly first task in the network
- Here, these are the tasks *A* and *C*.
- In case the chosen task to progress next is:
  - primitive: apply it and progress the state

Excursion: HTN Progression Search



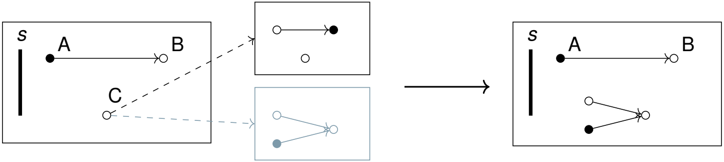
- Always progress tasks that are a possibly first task in the network
- Here, these are the tasks *A* and *C*.
- In case the chosen task to progress next is:
  - primitive: apply it and progress the state
  - compound: decompose it

Excursion: HTN Progression Search



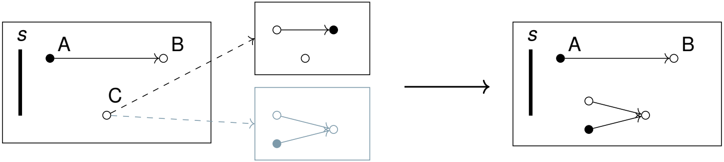
- Always progress tasks that are a possibly first task in the network
- Here, these are the tasks A and C.
- In case the chosen task to progress next is:
  - primitive: apply it and progress the state
  - compound: decompose it

Excursion: 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

Excursion: 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
- More details in Part II of this tutorial

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Proof:

- Rely on progression search

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Proof:

- Rely on progression search
- Until the compound task gets decomposed, all primitive tasks have been “progressed away”

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Proof:

- 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

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Note:

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

(This also shows the hardness of the problem.)

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Note:

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 $\mathcal{P}$

(This also shows the hardness of the problem.)

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Note:

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

(This also shows the hardness of the problem.)

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Note:

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

(This also shows the hardness of the problem.)

Computational Complexity

Theorem: Regular problems are in **PSPACE**.

Note:

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

(This also shows the hardness of the problem.)

Overview Part I

Theoretical Foundations

- Introduction
- Problem Definition
- Computational Complexity of the Plan Existence Problem
  - General HTN Planning
  - HTN Planning with Task Insertion
  - Totally Ordered HTN Planning
  - **Restricting Recursion (**Acyclic, Regular, **Tail-recursive)**
- Expressivity Analysis

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

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*.

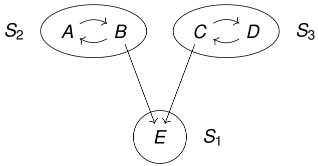
Stratifications

A stratification is defined as follows:

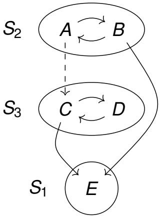
- A set $\leq \subseteq C \times C$ is called a *stratification* if it is a total preorder (i.e., reflexive, transitive, and *total*)

Stratifications: Example

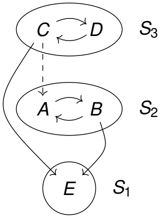
Stratifications: (Non-)Examples



(a) Relation $\leq_a$.



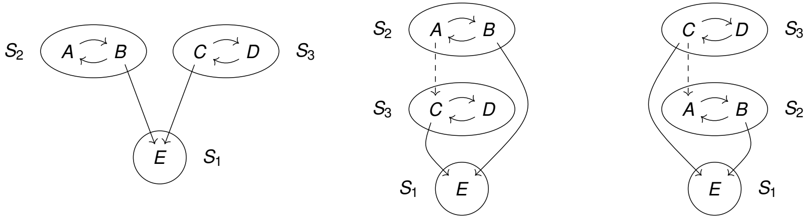
(b) Stratification $\leq_b$.



(c) Stratification $\leq_c$.

Stratifications: Example

Stratifications: (Non-)Examples

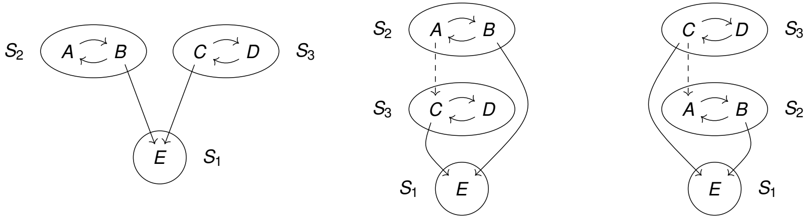


(a) Relation $\leq_a$. (b) Stratification $\leq_b$. (c) Stratification $\leq_c$.

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

Stratifications: Example

Stratifications: (Non-)Examples



(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*)

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.

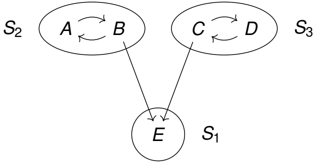
Stratifications

A stratification is defined as follows:

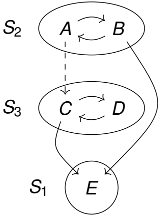
- A set $\leq \subseteq C \times C$ is called a *stratification* if it is a total preorder (i.e., reflexive, transitive, and *total*)
- We call any inclusion-maximal subset of C a *stratum* of $\leq$ if for all $x, y \in C$ both $(x, y) \in \leq$ and $(y, x) \in \leq$ hold.
- The *height of a stratification*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

Stratifications: Example

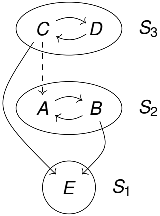
Stratifications: (Non-)Examples



(a) Relation $\leq_a$.



(b) Stratification $\leq_b$.

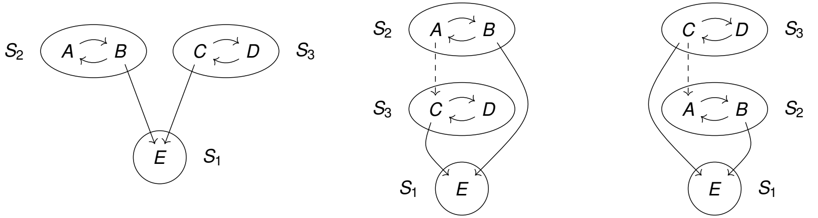


(c) Stratification $\leq_c$.

- $S_1 = \{E\}$, $S_2 = \{A, B\}$, and $S_3 = \{C, D\}$ are strata

Stratifications: Example

Stratifications: (Non-)Examples

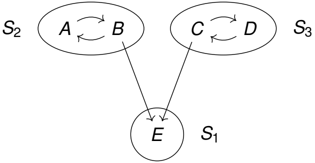


(a) Relation $\leq_a$. (b) Stratification $\leq_b$. (c) Stratification $\leq_c$.

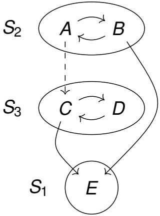
- $S_1 = \{E\}$, $S_2 = \{A, B\}$, and $S_3 = \{C, D\}$ are strata
- $\leq_b$ and $\leq_c$ have a height of 3.

Stratifications: Example

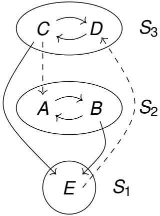
Stratifications: (Non-)Examples



(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.

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:

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 $(\alpha(t), c) \in \leq$

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

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 $(c, \alpha(t)) \notin \leq$

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

Computational Complexity

Theorem: Tail-recursive problems are in **EXPSPACE**.

Computational Complexity

Theorem: Tail-recursive problems are in **EXPSPACE**.

Proof:

- Rely on progression search (more details in Part II)

Computational Complexity

Theorem: Tail-recursive problems are in **EXPSPACE**.

Proof:

- Rely on progression search (more details in Part II)
- Until the last task gets decomposed, all tasks ordered before it have been “progressed away”

Computational Complexity

Theorem: Tail-recursive problems are in **EXPSPACE**.

Proof:

- Rely on progression search (more details in Part II)
- 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

Theorem: Tail-recursive problems are in **EXPSPACE**.

Proof:

- Rely on progression search (more details in Part II)
- 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

Computational Complexity

Theorem: Tail-recursive problems are in **EXPSPACE**.

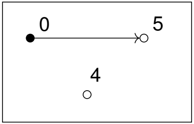
Proof:

- Rely on progression search (more details in Part II)
- 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

Computational Complexity: Example

Example:

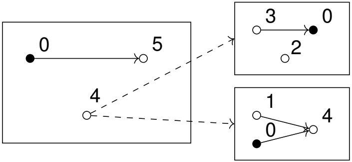
following initial task network of size 3:



Computational Complexity: Example

Example:

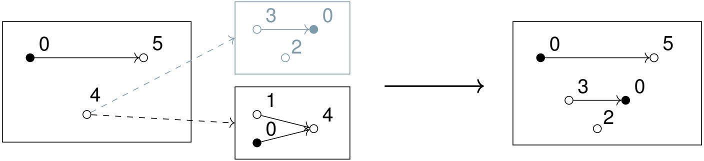
following initial task network of size 3:



Computational Complexity: Example

Example:

following initial task network of size 3:

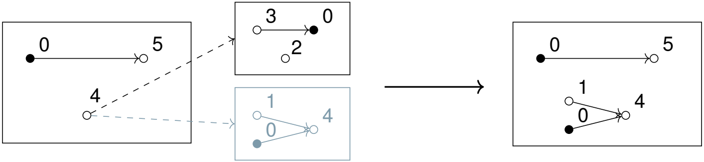


- Using a method *without* last task increases the size,
- but "such decompositions" can only finitely often (limited by the stratification height).

Computational Complexity: Example

Example:

following initial task network of size 3:

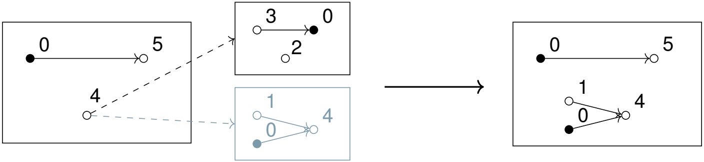


- Using a method *with* last task increases the size,

Computational Complexity: Example

Example:

following initial task network of size 3:

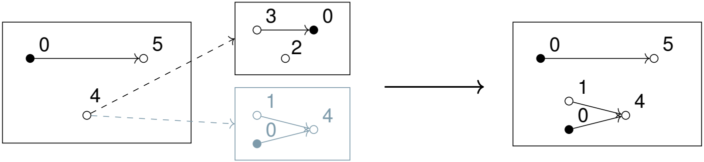


- Using a method *with* last task increases the size,
- and a task with the same stratification height remains(!),

Computational Complexity: Example

Example:

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.

Computational Complexity

Theorem: Tail-recursive problems are in **EXPSPACE**.

Proof:

- Rely on progression search (more details in Part II)
- 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 $k \cdot m^h$ as progression bound, where k is size of the initial task network, m is the size of the largest method, and h is the stratification height

Overview for Task Insertion

- When *task insertion* is allowed:
  - Recursion does not contribute to the hardness of the problem
  - Additional actions can be added by task insertion rather than by relying on recursive decomposition

Overview for Task Insertion

- When *task insertion* is allowed:
  - Recursion does not contribute to the hardness of the problem
  - Additional actions can be added by task insertion rather than by relying on recursive decomposition
- TIHTN Planning is **NEXPTIME**-complete (only membership was shown)

Overview for Standard HTN Planning

- HTN planning is in general undecidable

Overview for Standard HTN Planning

- HTN planning is in general undecidable
- HTN planning is also semi-decidable (not shown, but trivial)

Overview for Standard HTN Planning

- HTN planning is in general undecidable
- HTN planning is also semi-decidable (not shown, but trivial)
- Acyclic HTN problems are **NEXPTIME**-complete (only membership was shown)

Overview for Standard HTN Planning

- HTN planning is in general undecidable
- HTN planning is also semi-decidable (not shown, but trivial)
- Acyclic HTN problems are **NEXPTIME**-complete (only membership was shown)
- Regular HTN problems are **PSPACE**-complete

Overview for Standard HTN Planning

- HTN planning is in general undecidable
- HTN planning is also semi-decidable (not shown, but trivial)
- Acyclic HTN problems are **NEXPTIME**-complete (only membership was shown)
- Regular HTN problems are **PSPACE**-complete
- Tail-recursive HTN problems are **EXPSPACE**-complete (only membership was shown)

Overview Part I

Theoretical Foundations

- Introduction
- Problem Definition
- Computational Complexity of the Plan Existence Problem
  - General HTN Planning
  - HTN Planning with Task Insertion
  - Totally Ordered HTN Planning
  - Restricting Recursion (Acyclic, Regular, Tail-recursive)
- **Expressivity Analysis**

Expressivity of Planning Formalisms

Question:

- What can be *expressed* with the planning formalism at hand?

Expressivity of Planning Formalisms

Question:

- What can be *expressed* with the planning formalism at hand?
- How does behavior describable with a formalism look like?

Expressivity of Planning Formalisms

Question:

- What can be *expressed* with the planning formalism at hand?
- How does behavior describable with a formalism look like?

Answer so far:

- It is PSPACE-complete (undecidable) to find a STRIPS (HTN) plan

Expressivity of Planning Formalisms

Question:

- What can be *expressed* with the planning formalism at hand?
- How does behavior describable with a formalism look like?

Answer so far:

- It is PSPACE-complete (undecidable) to find a STRIPS (HTN) plan

Better answer:

- Formalism *A* can be compiled into formalism *B* (under several restrictions on compilation size and/or runtime)
- Gives an intuition on the relative expressivity
- Answer regarding STRIPS and HTN planning would be: (in general) impossible

Expressivity of Planning Formalisms

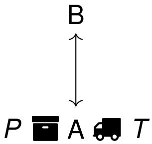
In the next slides:

- Provide a measure that allows more insights into the *structures* that can be represented

Expressivity of Planning Formalisms

In the next slides:

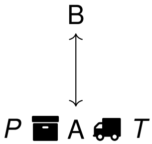
- Provide a measure that allows more insights into the *structures* that can be represented
- Consider the small STRIPS planning problem given at the right, *P* shall be delivered at *B*



Expressivity of Planning Formalisms

In the next slides:

- Provide a measure that allows more insights into the *structures* that can be represented
- Consider the small STRIPS planning problem given at the right, *P* shall be delivered at *B*

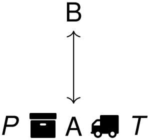


- Model is a compact representation for a space of states
- Actions define state transitions
- Initial state and goal definition specifies a set of (transition) sequences we are interested in

Expressivity of Planning Formalisms

In the next slides:

- Provide a measure that allows more insights into the *structures* that can be represented
- Consider the small STRIPS planning problem given at the right, P shall be delivered at B



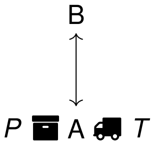
- Model is a compact representation for a space of states
- Actions define state transitions
- Initial state and goal definition specifies a set of (transition) sequences we are interested in

$$\{\langle \mathit{pickup}(T, P, A), \mathit{move}(T, A, B), \mathit{drop}(T, P, B)\rangle,$$
$$\langle \mathit{pickup}(T, P, A), \mathit{drop}(T, P, A), \mathit{pickup}(T, P, A), \mathit{move}(T, A, B), \mathit{drop}(T, P, B)\rangle,$$
$$\langle \mathit{pickup}(T, P, A), \mathit{move}(T, A, B), \mathit{move}(T, B, A), \mathit{move}(T, A, B), \mathit{drop}(T, P, B)\rangle,$$
$$\langle \mathit{move}(T, A, B), \mathit{move}(T, B, A), \mathit{pickup}(T, P, A), \mathit{move}(T, A, B), \mathit{drop}(T, P, B)\rangle, \ldots\}$$

Expressivity of Planning Formalisms

In the next slides:

- Provide a measure that allows more insights into the *structures* that can be represented
- Consider the small STRIPS planning problem given at the right, *P* shall be delivered at *B*



- Model is a compact representation for a space of states
- Actions define state transitions
- Initial state and goal definition specifies a set of (transition) sequences we are interested in

$\{\langle pickup(T, P, A), move(T, A, B), drop(T, P, B)\rangle,$

$\langle pickup(T, P, A), drop(T, P, A), pickup(T, P, A), move(T, A, B), drop(T, P, B)\rangle,$

$\langle pickup(T, P, A), move(T, A, B), move(T, B, A), move(T, A, B), drop(T, P, B)\rangle,$

$\langle move(T, A, B), move(T, B, A), pickup(T, P, A), move(T, A, B), drop(T, P, B)\rangle, \dots\}$

→ The planning problem is a compact representation for a (possibly infinite) set of sequences

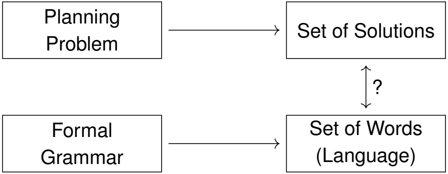
Comparison to Formal Languages

Planning Problem $\longrightarrow$ Set of Solutions

Formal Grammar $\longrightarrow$ Set of Words (Language)

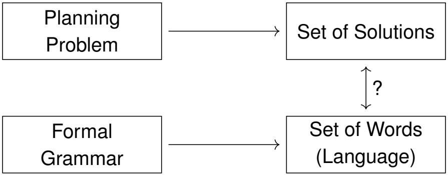
- Actions of a problem form the (terminal) symbols of a language
- Solution criteria define valid words
- Set of solutions forms the *language* of the problem

Comparison to Formal Languages



- Actions of a problem form the (terminal) symbols of a language
- Solution criteria define valid words
- Set of solutions forms the *language* of the problem

Comparison to Formal Languages



- Actions of a problem form the (terminal) symbols of a language
- Solution criteria define valid words
- Set of solutions forms the *language* of the problem

→ Which languages can be expressed using a certain formalism?

Comparison to Formal Languages

$$
\begin{tikzcd}
\boxed{\text{Planning Problem}} \arrow[r] & \boxed{\text{Set of Solutions}} \arrow[d, leftrightarrow, "?"] \\
\boxed{\text{Formal Grammar}} \arrow[r] & \boxed{\text{Set of Words (Language)}}
\end{tikzcd}
$$

- Actions of a problem form the (terminal) symbols of a language
- Solution criteria define valid words
- Set of solutions forms the *language* of the problem

→ Which languages can be expressed using a certain formalism?

Question (given before):

- What can be *expressed* with the planning formalism at hand?

Answer:

- STRIPS can represent (a subset of the) regular languages

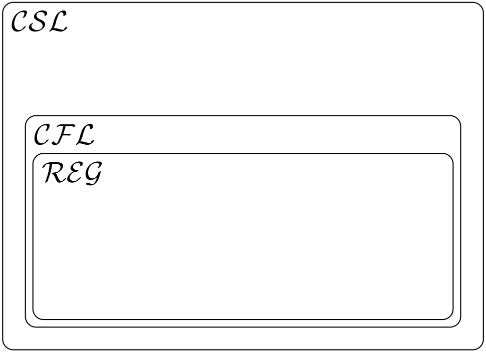
Expressivity via Comparison to Formal Languages

- Chomsky hierarchy as reference framework

Expressivity via Comparison to Formal Languages

- Chomsky hierarchy as reference framework
- Corresponding problems from planning and formal languages
  - Plan Existence and Emptiness Problem
  - Plan Verification and Word Problem
  - Plan Recognition and Prefix Problem

Expressivity via Comparison to Formal Languages



Language of STRIPS with Conditional Effects

Theorem: STRIPS with conditional effects (SCE) is equivalent to the regular languages

Language of STRIPS with Conditional Effects

Theorem: STRIPS with conditional effects (SCE) is equivalent to the regular languages

Proof: *(to show)*

- For every SCE planning problem, there is an equivalent regular language
- For every regular language, there is a SCE problem generating it

Language of STRIPS with Conditional Effects

- Let $P = (V, A, s_0, g)$ be a planning problem
- We define a Deterministic Finite Automaton (Σ, S, d, i, F) with
  - Σ is its input alphabet
  - S its set of states
  - $d : S \times \Sigma \to S$ its state-transition function
  - i its initial state
  - F its set of final states

Language of STRIPS with Conditional Effects

- Let $P = (V, A, s_0, g)$ be a planning problem
- We define a Deterministic Finite Automaton (Σ, S, d, i, F) with
  - Σ is its input alphabet
  - S its set of states
  - $d : S \times \Sigma \rightarrow S$ its state-transition function
  - i its initial state
  - F its set of final states
- We define
  - $\Sigma = A$
  - $S = 2^V$
  - i contains exactly the literals that hold in s_0
  - Every state including the literals in g is included in F

$$d(s, a) = \begin{cases} s', & \text{iff } (\tau(a, s) \wedge \gamma(a, s) = s') \\ \text{undefined}, & \text{else} \end{cases}$$

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$,

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
  - A equals the alphabet Σ

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
  - A equals the alphabet Σ

$$\forall a \in A : prec(a)$$
$$add(a)$$

$$del(a)$$

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
  - A equals the alphabet Σ

$$\forall a \in A : prec(a) = \emptyset$$

$$add(a)$$

$$del(a)$$

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
  - A equals the alphabet Σ

$$\forall a \in A : prec(a) = \emptyset$$

$$add(a) = \{(\{s\} \rightarrow \{s'\} \qquad) \mid d(s, a) = s'\}$$

$$del(a)$$

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
  - A equals the alphabet Σ

$$\forall a \in A : prec(a) = \emptyset$$
$$add(a) = \{(\{s\} \rightarrow \{s'\} \cup G') \mid d(s,a) = s'\}$$

$$del(a)$$

Language of STRIPS with Conditional Effects

- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
  - A equals the alphabet Σ

$$\forall a \in A : prec(a) = \emptyset$$

$$add(a) = \{(\{s\} \rightarrow \{s'\} \cup G') \mid d(s, a) = s'\}$$

$$\text{with } G' = \begin{cases} \{g\}, & \text{if } s' \in F \\ \emptyset, & \text{else} \end{cases}$$

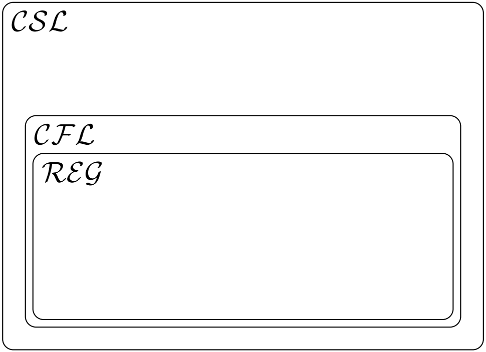
$$del(a)$$

Language of STRIPS with Conditional Effects

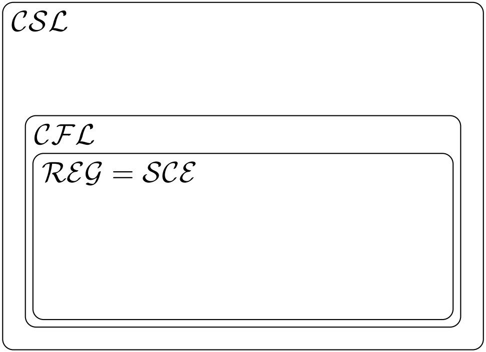
- Let (Σ, S, d, i, F) be a Deterministic Finite Automaton
- We define a planning problem $\mathcal{P} = (V, A, s_0, g)$
  - $V = S \cup \{g\}$ and $g \notin S$
  - $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
  - A equals the alphabet Σ

$$\forall a \in A : prec(a) = \emptyset$$
$$add(a) = \{(\{s\} \rightarrow \{s'\} \cup G') \mid d(s, a) = s'\}$$
$$\text{with } G' = \begin{cases} \{g\}, & \text{if } s' \in F \\ \emptyset, & \text{else} \end{cases}$$
$$del(a) = \{(\emptyset \rightarrow V)\}$$

Expressivity via Comparison to Formal Languages

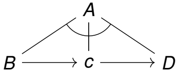


Expressivity via Comparison to Formal Languages



Totally Ordered HTN Planning Problems

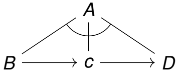
- Decomposition in totally ordered HTN planning problems is very similar to rule application in context-free grammars



$$A \mapsto BcD$$

Totally Ordered HTN Planning Problems

- Decomposition in totally ordered HTN planning problems is very similar to rule application in context-free grammars

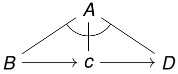


$$A \mapsto BcD$$

- The encoding of (totally ordered) HTN decomposition as (context-free) grammar rules and vice versa is straightforward
- Constraints introduced by preconditions and effects can be treated via intersection with a regular language

Totally Ordered HTN Planning Problems

- Decomposition in totally ordered HTN planning problems is very similar to rule application in context-free grammars

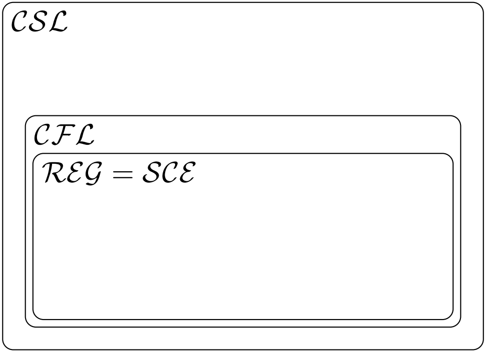


$$A \mapsto BcD$$

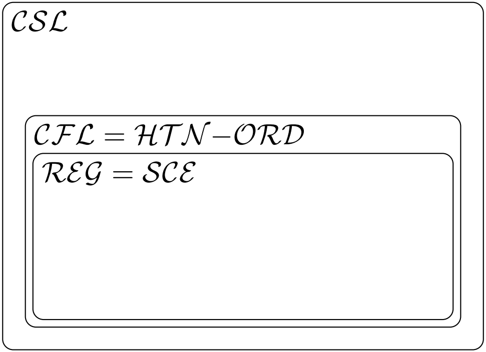
- The encoding of (totally ordered) HTN decomposition as (context-free) grammar rules and vice versa is straightforward
- Constraints introduced by preconditions and effects can be treated via intersection with a regular language

$\rightarrow$ $\mathcal{HTN}\text{–}\mathcal{ORD} = \mathcal{CFL}$

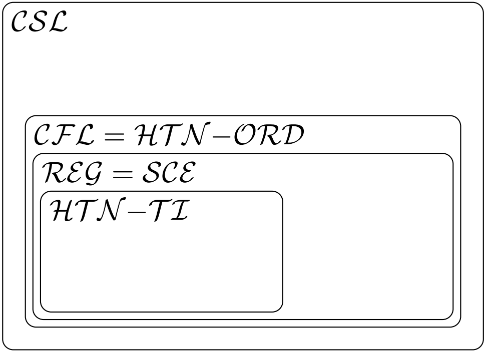
Expressivity via Comparison to Formal Languages



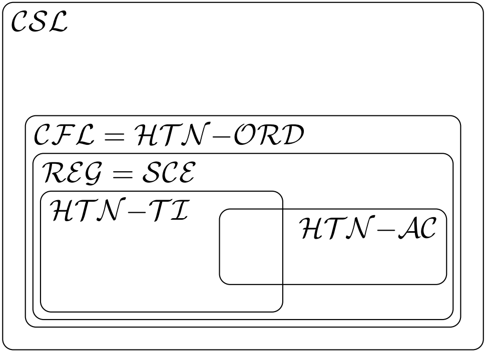
Expressivity via Comparison to Formal Languages



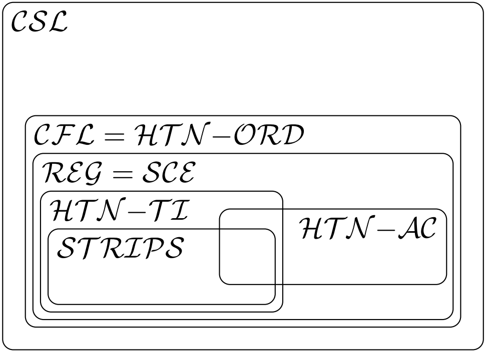
Expressivity via Comparison to Formal Languages



Expressivity via Comparison to Formal Languages



Expressivity via Comparison to Formal Languages



Noop HTN Planning Problems

- Subtasks of the problem's methods may be partially ordered

Noop HTN Planning Problems

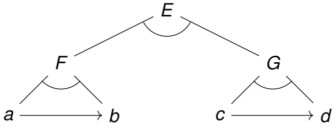
- Subtasks of the problem's methods may be partially ordered
- First class we look at:
  $\mathcal{HTN}\!-\!\mathcal{NOOP}$ – actions have no preconditions and effects

Noop HTN Planning Problems

- Subtasks of the problem's methods may be partially ordered
- First class we look at:
  $\mathcal{HTN}\!-\!\mathcal{NOOP}$ – actions have no preconditions and effects
- Can a partially ordered method be transfered to a set of totally ordered methods?

Noop HTN Planning Problems

HTN



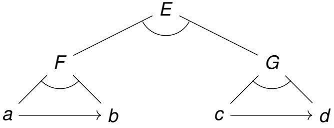
Grammar

$E \mapsto FG \quad F \mapsto ab$

$E \mapsto GF \quad G \mapsto cd$

Noop HTN Planning Problems

HTN



Grammar

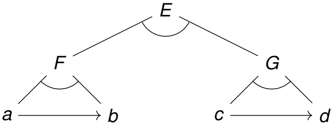
$E \mapsto FG \quad F \mapsto ab$

$E \mapsto GF \quad G \mapsto cd$

Word 1 *cdab*

Noop HTN Planning Problems

HTN



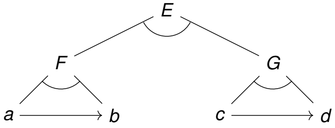
Grammar

$$E \mapsto FG \quad F \mapsto ab$$
$$E \mapsto GF \quad G \mapsto cd$$

Word 1 *cdab* ✓

Noop HTN Planning Problems

HTN



Grammar

$E \mapsto FG \quad F \mapsto ab$

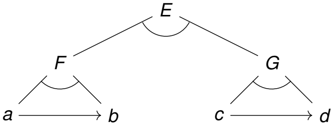
$E \mapsto GF \quad G \mapsto cd$

Word 1 *cdab* ✓

Word 2 *acbd*

Noop HTN Planning Problems

HTN



Grammar

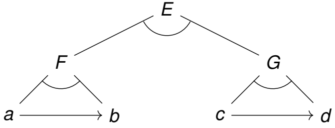
$E \mapsto FG \quad F \mapsto ab$

$E \mapsto GF \quad G \mapsto cd$

Word 1 *cdab* ✓ Word 2 *acbd* X

Noop HTN Planning Problems

HTN



Grammar

$E \mapsto FG \quad F \mapsto ab$

$E \mapsto GF \quad G \mapsto cd$

Word 1 *cdab* ✓

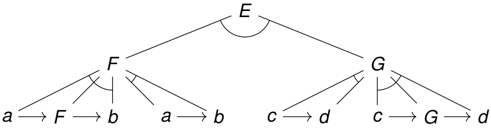
$ab||cd$

Word 2 *acbd* X

$\{abcd\} \cup \{cdab\}$

Noop HTN Planning Problems

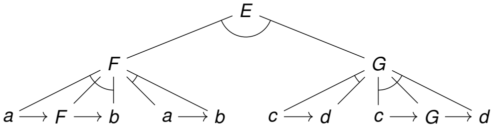
- The HTN depicted below generates the language $a^n b^n || c^m d^m$
- Using the *Pumping Lemma* for context-free languages, it can be shown that this language is not context-free



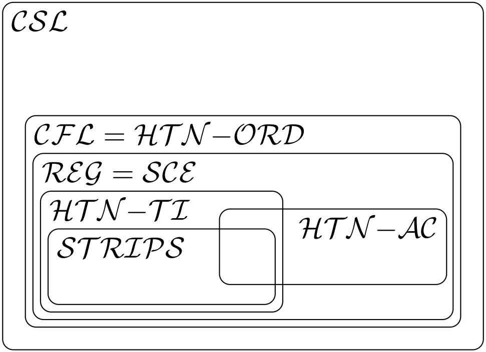
Noop HTN Planning Problems

- The HTN depicted below generates the language $a^n b^n || c^m d^m$
- Using the *Pumping Lemma* for context-free languages, it can be shown that this language is not context-free

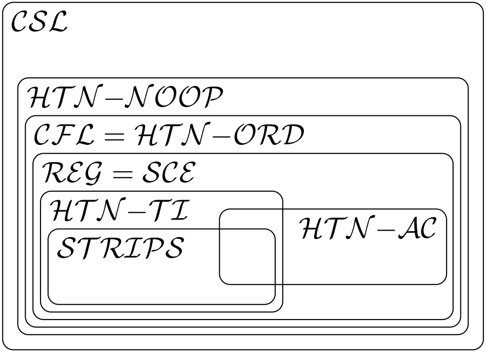
$\rightarrow$ $\mathcal{CFL} \subsetneq \mathcal{HTN}{-}\mathcal{NOOP}$



Expressivity via Comparison to Formal Languages



Expressivity via Comparison to Formal Languages



Full HTN Planning Problems

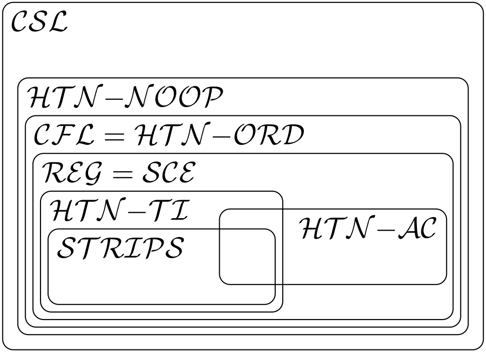
- For every HTN there is a linear space-bounded Turing machine that decides its word problem

Full HTN Planning Problems

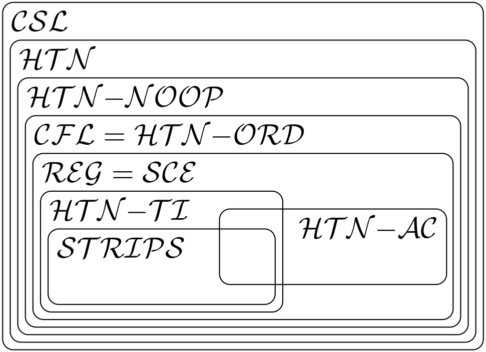
- For every HTN there is a linear space-bounded Turing machine that decides its word problem

$\rightarrow \mathcal{HTN} \subseteq \mathcal{CSL}$

Expressivity via Comparison to Formal Languages



Expressivity via Comparison to Formal Languages



Expressivity via Comparison to Formal Languages

- Chomsky hierarchy as reference framework
- Corresponding problems from planning and formal languages
  - Plan Existence and Emptiness Problem
  - Plan Verification and Word Problem
  - Plan Recognition and Prefix Problem

Expressivity via Comparison to Formal Languages

- Chomsky hierarchy as reference framework
- Corresponding problems from planning and formal languages
  - Plan Existence and Emptiness Problem
  - Plan Verification and Word Problem
  - Plan Recognition and Prefix Problem
- Representation blow-up is not considered
  - Theoretical approach to assess expressivity
  - Measures expressivity, not computational complexity

Thank You for Your Attention!

Thank you for your attention!

Are there questions?