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
Target Audience

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
Target Audience

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**
Environment:

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

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
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:
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.
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)
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)
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
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 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 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 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
Basic Problem Definition

Problem Definition & Solution Criteria

\[ P = (V, P, \delta, C, M, s_I, c_I) \]

- \( V \) a set of state variables
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
Basic Problem Definition

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

A solution task network \( tn \) must:

- be a refinement of \( c_I \),
Problem Definition & Solution Criteria

\[ P = (V, P, \delta, C, M, s_I, c_I) \]

- \( V \) 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
Basic Problem Definition

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
Basic Problem Definition

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
Basic Problem Definition

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
Basic Problem Definition

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
Basic Problem Definition

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
Basic Problem Definition

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
Basic Problem Definition

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
Basic Problem Definition

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.
More formally:

- For the sake of simplicity, we present a ground formalism, but most results exist for lifted planning as well.
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
  - $\alpha : T \rightarrow P \cup C$, the task mapping function

Primitive task names are mapped to their tuples by the task name mapping $\delta : P \rightarrow (2^V)^3$
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
  - $\alpha : T \rightarrow P \cup C$, the task mapping function

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

- A task network is called *executable* if there exists an executable linearization of its tasks
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
  - $\alpha : T \rightarrow P \cup C$, the task mapping function

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

- A task network is called *executable* if there exists an executable linearization of its tasks.

- Let $p$ be a primitive task (name) and $\delta(p) = (\text{prec}, \text{add}, \text{del})$. Then, $p$ is called *executable* in state $s \in 2^V$ iff $\text{prec} \subseteq s$.

  Its application to $s$ results into the state $(s \setminus \text{del}) \cup \text{add}$.

  Executability of task sequences defined analogously.
More formally:

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

\[
\prec_X := \{(t', t'') \mid (t', t) \in \prec, t'' \in T_m\} \cup \{(t'', t') \mid (t, t') \in \prec, t'' \in T_m\}
\]

where \( (X_1, \ldots, x_n)|_Y \) restricts the sets \( X_i \) to elements in \( Y \).
A task network $tn$ is a solution if and only if:

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

- A task network $tn$ is a solution if and only if:
  - There is a sequence of decomposition methods $\vec{m}$ that transforms $c_i$ into $tn$,
  - $tn$ contains only primitive tasks, and
  - the (still partially ordered) task network $tn$ admits an executable linearization $\vec{t}$ 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
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
Recap: $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ describes an HTN planning problem as described before.

Let $\mathcal{P}^* = (V, P, \delta, C, M, s_I, tn_I)$ be an HTN planning problem with initial task network $tn_I$. 
Recap: $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ describes an HTN planning problem as described before.

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

Then, a task network $t_n$ is a solution if and only if:

- There is a sequence of decomposition methods $\bar{m}$ that transforms $t_{n_I}$ into $t_n$,
- $t_n$ contains only primitive tasks, and
- the (still partially ordered) task network $t_n$ admits an executable linearization $\bar{f}$ of its tasks.
Theorem: Initial task networks can be compiled away.
Theorem: Initial task networks can be compiled away.

Proof:

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

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

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

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

Let $\mathcal{P^*} = (V, P, \delta, C, M, s_I, c_I, g)$ be an HTN planning problem with goal description $g \subseteq V$. 
Recap: $\mathcal{P} = (V, P, \delta, C, M, s_I, c_I)$ describes an HTN planning problem as described before.

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

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 $s \supseteq g$. 
Theorem: Goal descriptions can be compiled away.
Theorem: Goal descriptions can be compiled away.

Proof:

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

Then, there is an HTN planning problem $\mathcal{P}' = (V, P', \delta', C, M, s_I, tn_I)$ 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, $\delta(p) = (g, \emptyset, \emptyset)$. 
Theorem: Goal descriptions can be compiled away.

Proof:

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

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

Here, $t_{n_I}$ 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 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 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 \((l, first[t_1, \ldots, t_n])\), where the \(t_i\) are all sub tasks of \(t\). \((t, l)\) and \((t, l, t')\) are handled analogously.
State constraints 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 \((l, \text{first}[t_1, \ldots, t_n])\), 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 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 \((l, \text{first}[t_1, \ldots, t_n])\), 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
The HTN plan existence problem is defined as follows:

*Given an HTN planning problem $P$, does $P$ 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
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
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
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
**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 $\omega$ in both languages $L(G) \cap L(G')$?
**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 $\omega$ 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.
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 $\omega$ 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.
Theorem: HTN planning is undeducible.

Proof:

Reduction from the language intersection problem of two context-free grammars: given $G$ and $G'$, is there a word $\omega$ 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 $\omega$ – 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.
Theorem: HTN planning is undecidable.

Proof: (Cont’d, by example)

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

Production rules $R$: $H \rightarrow aQb$

Production rules $R'$: $D \rightarrow aFD \mid ab$

$Q \rightarrow aQ \mid bQ \mid a \mid b$

$F \rightarrow a \mid b$
**Theorem:** HTN planning is undecidable.

**Proof:** (Cont’d, by example)

\[ \mathcal{P} = (V, \{H, Q, D, F\}, \{a, b, a', b'\}, \delta, M, \{v_{\text{turn}:G}\}, t_{\text{tn}}, \{v_{\text{turn}:G}\}) \]
Theorem: HTN planning is undecidable.

Proof: (Cont’d, by example)

\[ \mathcal{P} = (V, \{H, Q, D, F\}, \{a, b, a', b'\}, \delta, M, \{v_{\text{turn}:G}\}, \text{tn}_I, \{v_{\text{turn}:G}\}) \]

\[ V = \{v_{\text{turn}:G}, v_{\text{turn}:G'}\} \cup \{v_a, v_b\} \]
Theorem: HTN planning is undecidable.

Proof: (Cont’d, by example)

$$\mathcal{P} = (V, \{H, Q, D, F\}, \{a, b, a', b'\}, \delta, M, \{v_{\text{turn}:G}\}, \text{tn}_I, \{v_{\text{turn}:G}\})$$

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

$$\delta = \{(a, (\{v_{\text{turn}:G}\}, \{v_{\text{turn}:G'}, v_a\}, \{v_{\text{turn}:G}\})),$$

$$\quad (b, (\{v_{\text{turn}:G}\}, \{v_{\text{turn}:G'}, v_b\}, \{v_{\text{turn}:G}\})),$$

$$\quad (a', (\{v_{\text{turn}:G'}, v_a\}, \{v_{\text{turn}:G}\}, \{v_{\text{turn}:G'}, v_a\})),$$

$$\quad (b', (\{v_{\text{turn}:G'}, v_b\}, \{v_{\text{turn}:G}\}, \{v_{\text{turn}:G'}, v_b\}))) \}$$
Theorem: HTN planning is undecidable.

Proof: (Cont'd, by example)

\[
\mathcal{P} = (V, \{ H, Q, D, F \}, \{ a, b, a', b' \}, \delta, M, \{ v_{turn}:G \}, t_{nI}, \{ v_{turn}:G' \})
\]

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

\[
\delta = \{(a, (\{ v_{turn}:G \}, \{ v_{turn}:G', v_a \}, \{ v_{turn}:G \})),
\]

\[
(b, (\{ v_{turn}:G \}, \{ v_{turn}:G', v_b \}, \{ v_{turn}:G \})),
\]

\[
(a', (\{ v_{turn}:G' \}, v_a, \{ v_{turn}:G \}, \{ v_{turn}:G', v_a \})),
\]

\[
(b', (\{ v_{turn}:G' \}, v_b, \{ v_{turn}:G \}, \{ v_{turn}:G', v_b \})))
\]

\[
M = M(G) \cup M(G') \ (\text{translated production rules of } G' \text{ and } G')
\]

\[
t_{nI} = (\{ t, t' \}, \emptyset, \{(t, H), (t', D)\})
\]
Which properties make the plan existence problem easier?

- Task insertion,
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:
Which properties make the plan existence problem easier?

- Task insertion,
- Total order of all task networks,
- Recursion. Methods are:
  - *acyclic*: no recursion
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
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
Which properties make the plan existence problem easier?

- Task insertion,
- Total order of all task networks,
- Recursion. Methods are:
  - \textit{acyclic}: no recursion
  - \textit{regular}: only one compound task, which is the last one
  - \textit{tail-recursive}: arbitrary many compound tasks, only the last one is recursive
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
In *HTN planning with task insertion, TIHTN planning*, tasks may be added arbitrarily to task networks (not just via decomposition):

Let $\mathcal{P}^* = (V, P, \delta, C, M, s_I, c_I)$ be a *TIHTN planning problem.*
In **HTN planning with task insertion, TIHTN planning**, tasks may be added arbitrarily to task networks (not just via decomposition):

Let $\mathcal{P}^* = (V, P, \delta, C, M, s_i, c_i)$ 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.
Plan Existence Problem of TIHTN Planning

Problem Definition

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

Let $P^* = (V, P, \delta, C, M, s_I, c_I)$ 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 $tn'$,
- $tn \supseteq tn'$ contains all tasks and orderings of $tn'$,
- $tn$ contains only primitive tasks, and
- the (still partially ordered) task network $tn$ admits an executable linearization $\overline{t}$ of its tasks.
Benefits of allowing task insertion:

- Task insertion plus goal description fully subsumes classical planning (while allowing task hierarchies as well)
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
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)
Recap: A task network is a solution if it contains the same word $\omega$ twice.
Recap: A task network is a solution if it contains the same word $\omega$ twice.

Task network $tn_6$ is a solution!
Recap: A task network is a solution if it contains the same word $\omega$ twice.

Task network $tn_8$ is no solution!
Plan Existence Problem of TIHTN Planning

Influence of Task Insertion

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

Influence of task insertion:

$tn_1$ (initial task network)

$tn_2$

$tn_3$

$tn_4$

$tn_5$

$tn_6$

$tn_7$

$tn_8$
Plan Existence Problem of TIHTN Planning

Influence of Task Insertion

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

Observation:

In TIHTN planning, recursion is not required.
Theorem: TIHTN planning is in \textit{NEXPTIME}

\textbf{Idea:} Restrict to \emph{acyclic} decompositions, fill the rest with task insertion, and verify.

Remove cyclic decompositions of $Q$ and $D$
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.

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

Verify in \( O(b|C| + 1) \) whether the tree describes a correct sequence of decompositions.
**Theorem:** TIHTN planning is in \textbf{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)
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
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.
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.
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.

![Diagram showing the set of totally ordered primitive refinements](image)
**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.
**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:
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 $s'$
Theorem: Totally ordered HTN planning is in \textbf{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 $s'$
**Theorem:** Totally ordered HTN planning is in \textbf{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 ?
**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.
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
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
An HTN planning problem is called called *acyclic* if no compound task can reach itself via decomposition.
Theorem: Acyclic HTN planning is in \textbf{NEXPTIME}.
**Theorem:** Acyclic HTN planning is in $\text{NEXPTIME}$.

**Proof:**

Do the same as for TIHTN problems, but without the task insertion part:
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)
Theorem: Acyclic HTN planning is in \textbf{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.
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.
  $(C = \text{set of compound tasks})$
  $(b = \text{size of largest task network in the model})$
- Verify in $O(b^{|C|+1})$ whether the decompositions can be applied in sequence
- Guess a linearization of the resulting task network
Theorem: Acyclic HTN planning is in \textbf{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 $O(b^{|C|+1})$
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
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 task network $tn = (T, \prec, \alpha)$ is called *regular* if

- at most one task in $T$ is compound and
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 $t$.  

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

June 25th, ICAPS 2018 (Delft)
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.
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.
**Theorem:** Regular problems are in \textbf{PSPACE}.
**Theorem:** Regular problems are in PSPACE.

**Proof:**

- Rely on progression search
Regular Problems

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 $C$. 

```latex
\begin{figure}
\centering
\begin{tikzpicture}
\node [draw] (A) at (0,0) {$A$};
\node [draw] (B) at (1,0) {$B$};
\node [draw] (C) at (0,-1) {$C$};
\draw (A) -- (B);
\end{tikzpicture}
\end{figure}
```
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:
Regular Problems

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
Regular Problems

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

**Proof:**

- Rely on progression search
Theorem: Regular problems are in \textbf{PSPACE}.

Proof:

- Rely on progression search
- Until the compound task gets decomposed, all primitive tasks have been “progressed away”
**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
Theorem: Regular problems are in \textbf{PSPACE}.

Note:

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

(This also shows the hardness of the problem.)
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.)
Theorem: Regular problems are in $\text{PSPACE}$.

Note:

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

- The actions in $P_{\text{STRIPS}}$ are primitive tasks in $P$
- 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.)
Theorem: Regular problems are in PSPACE.

Note:

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

- The actions in $P_{STRIPS}$ are primitive tasks in $P$
- 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.)
**Theorem:** Regular problems are in \textbf{PSPACE}.

**Note:**

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

- The actions in \( \mathcal{P}_{\text{STRIPS}} \) are primitive tasks in \( \mathcal{P} \).
- There is just one compound task \( X \) generating all possible action sequences: for all \( p \in \mathcal{P} \), 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.)
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
Informally, *tail-recursive* problems look as follows:

- limited recursion for all tasks in all methods
- non-last tasks have a more restricted recursion
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*. 
A stratification is defined as follows:

- A set \( \leq \subseteq C \times C \) is called a \textit{stratification} if it is a total preorder (i.e., reflexive, transitive, and \textit{total})
Stratifications: (Non-)Examples

(a) Relation $\leq_a$.

(b) Stratification $\leq_b$.

(c) Stratification $\leq_c$. 
Stratifications: (Non-)Examples

(a) Relation $\leq_a$.

\begin{itemize}
  \item $\leq_a = \{(A, B), (B, A), (C, D), (D, C), (E, B), (E, C)\}$
  \item $\leq_a$ is not a stratification, as it is not total
\end{itemize}
Stratifications: (Non-)Examples

(a) Relation $\leq_a$.

$$\leq_a = \{(A, B), (B, A), (C, D), (D, C), (E, B), (E, C)\}$$

(b) Stratification $\leq_b$.

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

(c) Stratification $\leq_c$.

$$\leq_c = \{(A, B), (B, A), (C, D), (D, C), (E, B), (E, C), (A, C)\}^*$$
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*)
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.
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: (Non-)Examples

#### (a) Relation $\leq_a$.

- $S_1 = \{E\}$, $S_2 = \{A, B\}$, and $S_3 = \{C, D\}$ are strata
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: (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.
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:
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 $(\alpha(t), c) \in \leq$
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 $c$*
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 $(c, \alpha(t)) \notin \leq$
An HTN problem $\mathcal{P}$ is called \textit{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 \textit{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 $(\alpha(t), c) \in \leq$

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

  \textit{This means: any non-last task is easier (on a lower stratum) than the decomposed task $c$}
**Theorem:** Tail-recursive problems are in **EXPSPACE**.
Theorem: Tail-recursive problems are in $\text{EXPSPACE}$.

Proof:
- Rely on progression search (more details in Part II)
Theorem: Tail-recursive problems are in \textbf{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”
Theorem: Tail-recursive problems are in \( \text{EXPSPACE} \).

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
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
Example:

following initial task network of size 3:

```
0  5

  4
```

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.
Example:

following initial task network of size 3:

```
  0  5

  4

  3  0

  2

  3

  2

  1  4

  1  0

  4

  4

  0
```
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).
Example:

following initial task network of size 3:

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

following initial task network of size 3:

- Using a method \textit{with} last task increases the size,
- and a task with the same stratification height remains(!),
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.
**Theorem:** Tail-recursive problems are in \textit{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 \textit{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
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)
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)
HTN planning is in general undecidable

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

Acyclic HTN problems are $\text{NEXPTIME}$-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 $\text{NEXPTIME}$-complete (only membership was shown)
- Regular HTN problems are $\text{PSPACE}$-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 $\text{NEXPTIME}$-complete (only membership was shown)
- Regular HTN problems are $\text{PSPACE}$-complete
- Tail-recursive HTN problems are $\text{EXPSPACE}$-complete (only membership was shown)
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
Question:

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

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

Expressivity of Planning Formalisms

Question:

- What can be \textit{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.
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
In the next slides:

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

\[
\{ \langle \text{pickup}(T, P, A), \text{move}(T, A, B), \text{drop}(T, P, B) \rangle, \\
\langle \text{pickup}(T, P, A), \text{drop}(T, P, A), \text{pickup}(T, P, A), \text{move}(T, A, B), \text{drop}(T, P, B) \rangle, \\
\langle \text{pickup}(T, P, A), \text{move}(T, A, B), \text{move}(T, B, A), \text{move}(T, A, B), \text{drop}(T, P, B) \rangle, \\
\langle \text{move}(T, A, B), \text{move}(T, B, A), \text{pickup}(T, P, A), \text{move}(T, A, B), \text{drop}(T, P, B) \rangle, \ldots \} 
\]
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

\[
\{ \langle \text{pickup}(T, P, A), \text{move}(T, A, B), \text{drop}(T, P, B) \rangle, \\
\langle \text{pickup}(T, P, A), \text{drop}(T, P, A), \text{pickup}(T, P, A), \text{move}(T, A, B), \text{drop}(T, P, B) \rangle, \\
\langle \text{pickup}(T, P, A), \text{move}(T, A, B), \text{move}(T, B, A), \text{move}(T, A, B), \text{drop}(T, P, B) \rangle, \\
\langle \text{move}(T, A, B), \text{move}(T, B, A), \text{pickup}(T, P, A), \text{move}(T, A, B), \text{drop}(T, P, B) \rangle, \ldots \} 
\]

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

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
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?
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
Motivation

Expressivity via Comparison to Formal Languages

- Chomsky hierarchy as reference framework
Motivation

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
Motivation

Expressivity via Comparison to Formal Languages

$CSL$

$CFL$

$REG$
**Theorem:** STRIPS with conditional effects (SCE) is equivalent to the regular languages
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
Let $P = (V, A, s_0, g)$ be a planning problem.

We define a Deterministic Finite Automaton $(\Sigma, S, d, i, F)$ with

- $\Sigma$ 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
Let \( P = (V, A, s_0, g) \) be a planning problem.

We define a Deterministic Finite Automaton \((\Sigma, S, d, i, F)\) with:

- \( \Sigma \) is its input alphabet
- \( S \) is its set of states
- \( d : S \times \Sigma \rightarrow S \) is its state-transition function
- \( i \) is its initial state
- \( F \) is 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 \( F \)

\[
d(s, a) = \begin{cases} 
  s', & \text{iff } (\tau(a, s) \land \gamma(a, s) = s') \\
  \text{undefined}, & \text{else}
\end{cases}
\]
Let \((\Sigma, S, d, i, F)\) be a Deterministic Finite Automaton.
Let \((\Sigma, S, d, i, F)\) be a Deterministic Finite Automaton

We define a planning problem \(\mathcal{P} = (V, A, s_0, g)\).
Language of STRIPS with Conditional Effects

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

- We define a planning problem \(P = (V, A, s_0, g)\)
  - \(V = S \cup \{ g \}\) and \(g \not\in S\)
  - \(s_0 = \{ i \}\)
Analysis of Common Planning Formalisms

Language of STRIPS with Conditional Effects

- Let \((\Sigma, S, d, i, F)\) 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 \(i \in F\)
Let \((\Sigma, S, d, i, F)\) be a Deterministic Finite Automaton

We define a planning problem \(\mathcal{P} = (V, A, s_0, g)\)

- \(V = S \cup \{g\}\) and \(g \not\in S\)
- \(s_0 = \{i\}\), \(g \in s_0\) iff \(i \in F\)
- \(A\) equals the alphabet \(\Sigma\)
Let $\langle \Sigma, S, d, i, F \rangle$ be a Deterministic Finite Automaton

We define a planning problem $P = (V, A, s_0, g)$

1. $V = S \cup \{g\}$ and $g \not\in S$
2. $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
3. $A$ equals the alphabet $\Sigma$

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

$add(a)$

$del(a)$
Let \((\Sigma, S, d, i, F)\) 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 \(\Sigma\)

\[
\forall a \in A : prec(a) = \emptyset
\]

\[
add(a)
\]

\[
del(a)
\]
Let $(\Sigma, S, d, i, F)$ be a Deterministic Finite Automaton.

We define a planning problem $P = (V, A, s_0, g)$

- $V = S \cup \{g\}$ and $g \not\in S$
- $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
- $A$ equals the alphabet $\Sigma$

\[
\forall a \in A : prec(a) = \emptyset
\]

\[
add(a) = \{ ([\{s\} \rightarrow \{s'\}] \mid d(s, a) = s') \}
\]

\[
del(a)
\]
Let $(\Sigma, S, d, i, F)$ be a Deterministic Finite Automaton.

We define a planning problem $P = (V, A, s_0, g)$:

- $V = S \cup \{g\}$ and $g \notin S$
- $s_0 = \{i\}$, $g \in s_0$ iff $i \in F$
- $A$ equals the alphabet $\Sigma$

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

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

$$del(a)$$
About the Tutorial
Introduction

Problem Formalization

Plan Existence Problem

Expressivity Analysis

Analysis of Common Planning Formalisms

Language of STRIPS with Conditional Effects

- Let \((\Sigma, S, d, i, F)\) 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 \(\Sigma\)

\[
\forall a \in A : \text{prec}(a) = \emptyset
\]

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

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

\[
\text{del}(a)
\]
**Language of STRIPS with Conditional Effects**

- Let $(\Sigma, S, d, i, F)$ 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 $\Sigma$.

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

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

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

$$\text{del}(a) = \{(\emptyset \to V)\}$$
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[
\begin{align*}
CSL &= CFL \\
CFL &= REG
\end{align*}
\]
Expressivity via Comparison to Formal Languages

\[
\begin{align*}
\mathcal{CSL} \\
\mathcal{CFL} \\
\mathcal{REG} = \mathcal{SCE}
\end{align*}
\]
Decomposition in totally ordered HTN planning problems is very similar to rule application in context-free grammars.

\[ A \rightarrow BcD \]
Decomposition in totally ordered HTN planning problems is very similar to rule application in context-free grammars

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

\[
A \rightarrow B c D
\]

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.

\[\text{HTN} - \text{ORD} = \text{CFL}\]
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[\text{CSL} \supset \text{CFL} \]
\[\text{REG} = \text{SCE} \]
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[ \text{CSL} \]

\[ \text{CFL} = \text{HTN-ORD} \]

\[ \text{REG} = \text{SCE} \]
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[
\begin{align*}
\text{CSL} & \\
\text{CFL} & = \text{HTN-ORD} \\
\text{REG} & = \text{SCE} \\
\text{HTN-TI} & 
\end{align*}
\]
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[
\begin{align*}
\text{CSL} \\
\text{CFL} &= \text{HTN−ORD} \\
\text{REG} &= \text{SCE} \\
\text{HTN−TI} &\subseteq \text{HTN−AC}
\end{align*}
\]
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[
\begin{align*}
CSL & \\
CFL &= HTN-ORD \\
REG & = SCE \\
HTN-TI & \\
STRIPS & \\
HTN-AC & \\
\end{align*}
\]
Subtasks of the problem’s methods may be partially ordered
Subtasks of the problem’s methods may be partially ordered.

First class we look at:

\(\textit{HTN−NOOP}\) – actions have no preconditions and effects.
Subtasks of the problem’s methods may be partially ordered

First class we look at: $\textit{HTN-NOOP}$ – actions have no preconditions and effects

Can a partially ordered method be transferred to a set of totally ordered methods?
Analysis of Common Planning Formalisms

Noop HTN Planning Problems

```
HTN
  E
  / \   /
 F   G  \
  a   c  d
  \   /
   b  d

Grammar

E ⇔ FG  F ⇔ ab
E ⇔ GF  G ⇔ cd
```
Noop HTN Planning Problems

**HTN**

- **E**
  - **F**
    - **a** → **b**
  - **G**
    - **c** → **d**

**Grammar**

- **E** ⇔ **FG**
- **F** ⇔ **ab**
- **E** ⇔ **GF**
- **G** ⇔ **cd**

Word 1: cdab
Noop HTN Planning Problems

HTN

Grammar

\[ E \rightarrow FG \quad F \rightarrow ab \]
\[ E \rightarrow GF \quad G \rightarrow cd \]

Word 1: cdab  ✓
Analysis of Common Planning Formalisms

Noop HTN Planning Problems

![Diagram of HTN and Grammar with variables E, F, G, a, b, c, d, and rules E \rightarrow FG, F \rightarrow ab, E \rightarrow GF, G \rightarrow cd.]

Word 1 \ cdab \ ✓ \ Word 2 \ acbd
Noop HTN Planning Problems

```
<table>
<thead>
<tr>
<th>HTN</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E \rightarrow FG$</td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow ab$</td>
</tr>
<tr>
<td>$G$</td>
<td>$E \rightarrow GF$</td>
</tr>
<tr>
<td></td>
<td>$G \rightarrow cd$</td>
</tr>
</tbody>
</table>

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

\[ \{abcd\} \cup \{cdab\} \]

Diagram:

```
  E
 / \  /
F   G
|   |
|   |
|   |
a b c d
```

Analysis of Common Planning Formalisms

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
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

$\Rightarrow \mathcal{CFL} \subsetneq \mathcal{HTN-NOOP}$
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[ CSL \]

\[ CFL = HTN − ORD \]

\[ REG = SCE \]

\[ HTN − TI \]

\[ STRIPS \]

\[ HTN − AC \]
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[ \text{CSL} \]

\[ \text{HTN-NOOP} \]

\[ \text{CFL} = \text{HTN-ORD} \]

\[ \text{REG} = \text{SCE} \]

\[ \text{HTN-TI} \]

\[ \text{STRIPS} \quad \text{HTN-AC} \]
For every HTN there is a linear space-bounded Turing machine that decides its word problem
For every HTN there is a linear space-bounded Turing machine that decides its word problem

\[ \mathcal{HTN} \subseteq \mathcal{CSL} \]
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

\[
\begin{align*}
\text{CSL} \\
\text{HTN-NOOP} \\
\text{CFL} = \text{HTN-ORD} \\
\text{REG} = \text{SCE} \\
\text{HTN-TI} \\
\text{STRIPS} & \quad \text{HTN-AC}
\end{align*}
\]
Analysis of Common Planning Formalisms

Expressivity via Comparison to Formal Languages

- CSL
- HTN
- HTN-NOOP
- $\text{CFL} = \text{HTN-ORD}$
- $\text{REG} = \text{SCE}$
- HTN-TI
- STRIPS
- HTN-AC
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
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!

Are there questions?