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)
Solving HTN Planning Problems

- **Search-based Approaches**
  - Plan Space Search
  - Progression Search

- **Compilation-based Approaches**
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- **Heuristics for Heuristic Search**
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

**Excursion**

- Further Hierarchical Planning Formalisms
Example Domain

deliver(p, l2)

get-to(v, l1)  pick-up(v, l1, p)  get-to(v, l2)  drop(v, l2, p)

m-deliver(p, l1, l2, v)

get-to(v, l2)

l1 ≠ l2

drive(v, l1, l2)

m-direct(v, l1, l2)

get-to(v, l2)

get-to(v, l1)  drive(v, l1, l2)

m-via(v, l1, l2)

get-to(v, l)

l1 ≠ l2

get-to(v, l1)  drive(v, l1, l2)

m-via(v, l1, l2)

no-op()

m-noop(v, l)
Solving HTN Planning Problems

- **Search-based Approaches**
  - Plan Space Search
  - Progression Search

- **Compilation-based Approaches**
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- **Heuristics for Heuristic Search**
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- **Further Hierarchical Planning Formalisms**
Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
- Elements of the partial plan preventing it from being a solution are represented as so-called flaws:
  
  - compound task flaw \( t \): the task \( t \) is compound, i.e., not decomposed yet
  - open precondition flaw \( (t, oc) \): the precondition \( oc \) of the task \( t \) is still open or unprotected, i.e., no causal link protects it yet
  - causal treat flaw \( t \otimes (t', c, t'') \): there is a causal link between \( t' \) and \( t'' \) protecting the condition \( c \) and the ordering constraints allow \( t \) to be ordered between \( t' \) and \( t'' \), i.e., \( t' < t < t'' \) – and \( c \) is a delete effect of \( t \).
HTN Plan Space Search

Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
- Elements of the partial plan preventing it from being a solution are represented as so-called flaws:
  - **compound task flaw** \( t \): the task \( t \) is compound, i.e., not decomposed yet
Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
- Elements of the partial plan preventing it from being a solution are represented as so-called flaws:
  - **compound task flaw** of the task $t$: the task $t$ is compound, i.e., not decomposed yet
  - **open precondition flaw** ($t, oc$): the precondition $oc$ of the task $t$ is still open or unprotected, i.e., no causal link protects it yet
HTN Plan Space Search

Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
- Elements of the partial plan preventing it from being a solution are represented as so-called flaws:
  - **compound task flaw** $t$: the task $t$ is compound, i.e., not decomposed yet
  - **open precondition flaw** $(t, oc)$: the precondition $oc$ of the task $t$ is still open or unprotected, i.e., no causal link protects it yet
  - **causal treat flaw** $t' \leq c \leq t''$: there is a causal link between $t'$ and $t''$ protecting the condition $c$ and the ordering constraints allow $t$ to be ordered between $t'$ and $t''$, i.e., $t' < t < t''$ – and $c$ is a delete effect of $t$. 
Modifications for *compound task flaws*:

- Decompose the compound task (one modification for each method)

\[
\text{get-to}(T_1, B)
\]
Modifications for *open precondition flaws*:
HTN Plan Space Search

Flaws in Partial Plans

Modifications for *open precondition flaws*:

- Insert a causal link from existing plan step (one modification for each possible producer)
HTN Plan Space Search

Flaws in Partial Plans

Modifications for open precondition flaws:

- Insert a causal link from existing plan step (one modification for each possible producer)
- Decompose a compound task if it has a sub task with a compatible effect (one modification for each method that has a compatible sub task)
HTN Plan Space Search

Flaws in Partial Plans

Modifications for *open precondition flaws*:

- Insert a causal link from existing plan step (one modification for each possible producer)
- Decompose a compound task if it has a sub task with a compatible effect (one modification for each method that has a compatible sub task)
- Insert a causal link from a newly inserted task (one modification for each possible producer) – only if task insertion is allowed!
Modifications for *causal threat flaws*: 

- Move the threatening task before the producer of the threatened link, called *demotion* (not possible here).
- Move the threatening task behind the consumer of the threatened link, called *promotion*.
Modifications for *causal threat flaws*:

- Move the threatening task before the producer of the threatened link, called *demotion* (not possible here)
Modifications for *causal threat flaws*:

- Move the threatening task before the producer of the threatened link, called *demotion* (not possible here)
- Move the threatening task behind the consumer of the threatened link, called *promotion*
Partial plans (as well as solutions) are only partially ordered, thus compactly representing many linearizations.
Partial plans (as well as solutions) are only partially ordered, thus compactly representing many linearizations.

Search works both top-down (decomposition of compound tasks) as well as backwards (goal-directed causal link establishment).
Partial plans (as well as solutions) are only partially ordered, thus compactly representing many linearizations.

Search works both top-down (decomposition of compound tasks) as well as backwards (goal-directed causal link establishment).

Search works in a two-step way:
- Select a most-promising plan (via standard search strategies)
Partial plans (as well as solutions) are only partially ordered, thus compactly representing many linearizations.

Search works both top-down (decomposition of compound tasks) as well as backwards (goal-directed causal link establishment).

Search works in a two-step way:

- Select a most-promising plan (via standard search strategies).
- Then, select a flaw (this is not(!) a backtrack point) and branch over all possibilities to resolve it.

Follows the principle of least commitment.
Solving Techniques

Heuristics

Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

**Input**: fringe = \{P_{\text{init}}\}

**Output**: A solution plan or fail.

1. while fringe \neq \emptyset do
2. \hspace{1em} P := \text{PlanSel}(fringe)
3. \hspace{1em} F := \text{FlawDet}(P)
4. \hspace{1em} if F = \emptyset then return P
5. \hspace{1em} f := \text{FlawSel}(F)
6. \hspace{1em} fringe := (fringe \setminus \{P\}) \cup \text{Successors}(P, f)
7. return fail

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

Standard Plan Space-based Algorithm

**Input**: fringe = \{ P_{\text{init}} \}

**Output**: A solution plan or fail.

1. while fringe ≠ ∅ do
2. \( P := \text{PlanSel}(\text{fringe}) \)
3. \( F := \text{FlawDet}(P) \)
4. if \( F = \emptyset \) then return \( P \)
5. \( f := \text{FlawSel}(F) \)
6. \( \text{fringe} := (\text{fringe} \setminus \{ P \}) \cup \text{Successors}(P, f) \)
7. return fail

**Initial partial plan** \( P_{\text{init}} \) equals the initial task network preceded by an artificial task encoding the initial state.
Solving Techniques

HTN Plan Space Search

Standard Plan Space-based Algorithm

**Input:** fringe = \{ P_{init} \}

**Output:** A solution plan or fail.

1. while fringe ≠ ∅ do
2. \( P := \text{PlanSel}(\text{fringe}) \)
3. \( F := \text{FlawDet}(P) \)
4. if \( F = ∅ \) then return \( P \)
5. \( f := \text{FlawSel}(F) \)
6. fringe := (fringe \&\& \{ P \}) \cup \text{Successors}(P, f)
7. return fail

- **Initial partial plan** \( P_{init} \) equals the initial task network preceded by an artificial task encoding the initial state

- **Search nodes** contain partial plans of the form \(( T, \prec, \alpha, CL)\)
HTN Plan Space Search

Standard Plan Space-based Algorithm

Input: $fringe = \{P_{init}\}$

Output: A solution plan or fail.

1. while $fringe \neq \emptyset$ do
2.     $P := PlanSel(fringe)$
3.     $F := FlawDet(P)$
4.     if $F = \emptyset$ then return $P$
5.     $f := FlawSel(F)$
6.     $fringe := (fringe \setminus \{P\}) \cup Successors(P, f)$
7. return fail

- **Initial partial plan** $P_{init}$ equals the initial task network preceded by an artificial task encoding the initial state.

- **Search nodes** contain partial plans of the form $(T, \prec, \alpha, CL)$.

- **Fringe** is sorted according to some heuristic.
HTN Plan Space Search

Standard Plan Space-based Algorithm

**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

1. while fringe ≠ ∅ do
2. \hspace{1em} P := PlanSel(fringe)
3. \hspace{1em} F := FlawDet(P)
4. \hspace{1em} if F = ∅ then return P
5. \hspace{1em} f := FlawSel(F)
6. \hspace{1em} fringe := (fringe \{P\})
7. \hspace{1em} fringe := fringe ∪ Successors(P, f)
8. return fail

- **Initial partial plan** $P_{init}$ equals the initial task network preceded by an artificial task encoding the initial state.

- **Search nodes** contain partial plans of the form $(T, \prec, \alpha, CL)$.

- **Fringe** is sorted according to some heuristic.

- **F** is the set of all flaws of the current partial plan.
HTN Plan Space Search

Standard Plan Space-based Algorithm

Input : fringe = \{P_{init}\}
Output : A solution plan or fail.

1 while fringe ≠ ∅ do
   2 P := PlanSel(fringe)
   3 F := FlawDet(P)
   4 if F = ∅ then return P
   5 f := FlawSel(F)
   6 fringe := (fringe \ {P}) ∪ Successors(P, f)
   7 return fail

- **Initial partial plan** $P_{init}$ equals the initial task network preceded by an artificial task encoding the initial state.

- **Search nodes** contain partial plans of the form $(T, \prec, \alpha, CL)$.

- **Fringe** is sorted according to some heuristic.

- **F** is the set of all flaws of the current partial plan.

- **FlawSel** selects (not a backtrack point!) a flaw according to a flaw selection strategy.
Solving Techniques

HTN Plan Space Search

Standard Plan Space-based Algorithm

Input: fringe = \{P_{\text{init}}\}

Output: A solution plan or fail.

```
1 while fringe ≠ ∅ do
2   P := PlanSel(fringe)
3   F := FlawDet(P)
4   if F = ∅ then return P
5       f := FlawSel(F)
6       fringe := (fringe \ {P}) ∪ Successors(P, f)
7 return fail
```
Input: fringe = \{P_{init}\}

Output: A solution plan or fail.

while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P}) ∪ Successors(P, f)
return fail
**Solving Techniques**

**Heuristics**

**Excursion**

### HTN Plan Space Search

#### Standard Plan Space-based Algorithm

**Input**: fringe $= \{ P_{init} \}$

**Output**: A solution plan or **fail**.

1. while fringe $\neq \emptyset$ do
   2. $P := \text{PlanSel}(\text{fringe})$
   3. $F := \text{FlawDet}(P)$
   4. if $F = \emptyset$ then return $P$
   5. $f := \text{FlawSel}(F)$
   6. fringe := (fringe \ {P}) $\cup$ Successors($P$, $f$)
   7. return **fail**

---

**Flaws**

- Compound task: deliver($P_1, B$)
  - Decompose with $m$-deliver($P_1, A, B, T_1$)

- Compound task: deliver($P_2, D$)
  - Decompose with $m$-deliver($P_2, C, D, T_2$)

---

**Diagrams**

- HTN plan space search diagram
- Example HTN plan space search for a delivery task

---

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

June 25th, ICAPS 2018 (Delft)
Solving Techniques
Heuristics
Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

1. while fringe ≠ ∅ do
2. \( P := \text{PlanSel}(\text{fringe}) \)
3. \( F := \text{FlawDet}(P) \)
4. if \( F = \emptyset \) then return \( P \)
5. \( f := \text{FlawSel}(F) \)
6. fringe := (fringe \ \{P\}) ∪ Successors(\( P, f \))
7. return fail

<table>
<thead>
<tr>
<th>Flaws</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>compound task</strong>: deliver(( P_{2}, D ))</td>
<td>decompose with m-deliver(( P_{2}, C, D, T_{2} ))</td>
</tr>
<tr>
<td><strong>compound task</strong>: get-to(( T_{1}, A ))</td>
<td>decompose with m-direct(( T_{1}, B, A )) decompose with m-via(( T_{1}, B, A )) decompose with m-noop(( T_{1}, A ))</td>
</tr>
</tbody>
</table>

**open prec.**: at(\( T_{1}, A \)) of pick-up(\( T_{1}, A, P_{1} \))
- insert causal link from \( \text{init} \)
- decompose get-to(\( T_{1}, A \)) with m-direct(\( T_{1}, B, A \))
- decompose get-to(\( T_{1}, A \)) with m-via(\( T_{1}, B, A \))

**open prec.**: at(\( P_{1}, A \)) of pick-up(\( T_{1}, A, P_{1} \))
- insert causal link from \( \text{init} \)

**compound task**: get-to(\( T_{1}, B \))
- decompose with m-direct(\( T_{1}, A, B \))
- decompose with m-via(\( T_{1}, A, B \))
- decompose with m-noop(\( T_{1}, B \))

**open prec.**: at(\( T_{1}, B \)) of drop(\( T_{1}, B, P_{1} \))
- decompose get-to(\( T_{1}, B \)) with m-direct(\( T_{1}, A, B \))
- decompose get-to(\( T_{1}, B \)) with m-via(\( T_{1}, A, B \))

**open prec.**: in(\( P_{1}, T_{1} \)) of drop(\( T_{1}, B, P_{1} \))
- insert causal link from pick-up(\( T_{1}, A, P_{1} \))
### HTN Plan Space Search

#### Standard Plan Space-based Algorithm

**Input:** fringe = \{P_{\text{init}}\}

**Output:** A solution plan or fail.

1. while fringe ≠ ∅ do
2. \( P := \text{PlanSel}(\text{fringe}) \)
3. \( F := \text{FlawDet}(P) \)
4. if \( F = ∅ \) then return \( P \)
5. \( f := \text{FlawSel}(F) \)
6. fringe := (fringe \ \{P\}) ∪ Successors(P, f)
7. return fail

---

**Flaws**

- **compound task:** deliver(P₂, D)
  - **Modifications:**
  - decompose with m-deliver(P₂, C, D, T₂)

- **compound task:** get-to(T₁, A)
  - **Modifications:**
  - decompose with m-direct(T₁, B, A)
  - decompose with m-via(T₁, B, A)
  - decompose with m-noop(T₁, A)

- **open prec.:** at(T₁, A) of pick-up(T₁, A, P₁)
  - **Modifications:**
  - insert causal link from init
decompose get-to(T₁, A) with m-direct(T₁, B, A)
decompose get-to(T₁, A) with m-via(T₁, B, A)

- **open prec.:** at(P₁, A) of pick-up(T₁, A, P₁)
  - **Modifications:**
  - insert causal link from init

- **compound task:** get-to(T₁, B)
  - **Modifications:**
  - decompose with m-direct(T₁, A, B)
  - decompose with m-via(T₁, A, B)
  - decompose with m-noop(T₁, B)

- **open prec.:** at(T₁, B) of drop(T₁, B, P₁)
  - **Modifications:**
  - decompose get-to(T₁, B) with m-direct(T₁, A, B)
decompose get-to(T₁, B) with m-via(T₁, A, B)

- **open prec.:** in(P₁, T₁) of drop(T₁, B, P₁)
  - **Modifications:**
  - insert causal link from pickup(T1, A, P1)
**Solving Techniques**

**Heuristics**

**Excursion**

---

**HTN Plan Space Search**

**Standard Plan Space-based Algorithm**

Input: fringe = \{P_{init}\}

Output: A solution plan or fail.

1. while fringe ≠ ∅ do
   2. P := PlanSel(fringe)
   3. F := FlawDet(P)
   4. if F = ∅ then return P
   5. f := FlawSel(F)
   6. fringe := (fringe \ \{P\}) ∪ Successors(P, f)
   7. return fail

---

**Flaws**

- compound task: deliver(P_2, D)
  - decompose with m-deliver(P_2, C, D, T_2)

- compound task: get-to(T_1, A)
  - decompose with m-direct(T_1, B, A)
  - decompose with m-via(T_1, B, A)
  - decompose with m-noop(T_1, A)

- open prec.: at(T_1, A) of pick-up(T_1, A, P_1)
  - insert causal link from init
  - decompose get-to(T_1, A) with m-direct(T_1, B, A)
  - decompose get-to(T_1, A) with m-via(T_1, B, A)

- compound task: get-to(T_1, B)
  - decompose with m-direct(T_1, A, B)
  - decompose with m-via(T_1, A, B)
  - decompose with m-noop(T_1, B)

- open prec.: at(T_1, B) of drop(T_1, B, P_1)
  - decompose get-to(T_1, B) with m-direct(T_1, A, B)
  - decompose get-to(T_1, B) with m-via(T_1, A, B)

- open prec.: in(P_1, T_1) of drop(T_1, B, P_1)
  - insert causal link from pickup(T_1, A, P_1)
Solving Techniques

Heuristics

Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

Input : fringe = \{P_{init}\}
Output : A solution plan or fail.

1 while fringe \neq \emptyset do
2 \quad P := \text{PlanSel}(fringe)
3 \quad F := \text{FlawDet}(P)
4 \quad if F = \emptyset then return P
5 \quad f := \text{FlawSel}(F)
6 \quad fringe := (fringe \setminus \{P\}) \cup \text{Successors}(P, f)
7 return fail

Flaws

- compound task: deliver(P_2, D)
- compound task: get-to(T_1, A)

Modifications

- decompose with m-deliver(P_2, C, D, T_2)
- decompose with m-direct(T_1, B, A)
- decompose with m-via(T_1, B, A)
- decompose with m-noop(T_1, A)

open prec.: at(T_1, A) of pick-up(T_1, A, P_1)

insert causal link from init
de-compose get-to(T_1, A) with m-direct(T_1, B, A)
de-compose get-to(T_1, A) with m-via(T_1, B, A)

compound task: get-to(T_1, B)

- decompose with m-direct(T_1, A, B)
- decompose with m-via(T_1, A, B)
- decompose with m-noop(T_1, B)

open prec.: at(T_1, B) of drop(T_1, B, P_1)

- decompose get-to(T_1, B) with m-direct(T_1, A, B)
- decompose get-to(T_1, B) with m-via(T_1, A, B)

open prec.: in(P_1, T_1) of drop(T_1, B, P_1)

insert causal link from pickup(T_1, A, P_1)
**Solving Techniques**

**Heuristics**

**Excursion**

---

**HTN Plan Space Search**

**Standard Plan Space-based Algorithm**

---

**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

1. while fringe ≠ ∅ do
2. \( P := \text{PlanSel}(\text{fringe}) \)
3. \( F := \text{FlawDet}(P) \)
4. if \( F = \emptyset \) then return \( P \)
5. \( f := \text{FlawSel}(F) \)
6. fringe := (fringe \ \{P\}) \cup \text{Successors}(P, f)
7. return fail

---

**Flaws**

- **compound task**: deliver\( (P_2, D) \)
  - decompose with \( m\text{-deliver}(P_2, C, D, T_2) \)

- **compound task**: get-to\( (T_1, A) \)
  - decompose with \( m\text{-direct}(T_1, B, A) \)
  - decompose with \( m\text{-via}(T_1, B, A) \)
  - decompose with \( m\text{-noop}(T_1, A) \)

- **open prec.**: at\( (T_1, A) \) of pick-up\( (T_1, A, P_1) \)
  - insert causal link from \( \text{init} \)
  - decompose get-to\( (T_1, A) \) with \( m\text{-direct}(T_1, B, A) \)
  - decompose get-to\( (T_1, A) \) with \( m\text{-via}(T_1, B, A) \)

- **compound task**: get-to\( (T_1, B) \)
  - decompose with \( m\text{-direct}(T_1, A, B) \)
  - decompose with \( m\text{-via}(T_1, A, B) \)
  - decompose with \( m\text{-noop}(T_1, B) \)

- **open prec.**: at\( (T_1, B) \) of drop\( (T_1, B, P_1) \)
  - decompose get-to\( (T_1, B) \) with \( m\text{-direct}(T_1, A, B) \)
  - decompose get-to\( (T_1, B) \) with \( m\text{-via}(T_1, A, B) \)
Solving Techniques
Heuristics
Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

Input : fringe = \{P_{\text{init}}\}

Output : A solution plan or fail.

1. while fringe ≠ ∅ do
2. \[ P := \text{PlanSel}(\text{fringe}) \]
3. \[ F := \text{FlawDet}(P) \]
4. if \( F = ∅ \) then return \( P \)
5. \[ f := \text{FlawSel}(F) \]
6. fringe := (fringe \ − \ \{P\}) \cup \text{Successors}(P, f)
7. return fail

Flaws

**compound task: deliver\( (P_2, D) \)**
- decompose with \( m\text{-deliver}(P_2, C, D, T_2) \)

**compound task: get-to\( (T_1, A) \)**
- decompose with \( m\text{-direct}(T_1, B, A) \)
- decompose with \( m\text{-via}(T_1, B, A) \)
- decompose with \( m\text{-noop}(T_1, A) \)

**open prec.: at\( (T_1, A) \) of pick-up\( (T_1, A, P_1) \)**
- insert causal link from \( \text{init} \)
- decompose get-to\( (T_1, A) \) with \( m\text{-direct}(T_1, B, A) \)
- decompose get-to\( (T_1, A) \) with \( m\text{-via}(T_1, B, A) \)

**compound task: get-to\( (T_1, B) \)**
- decompose with \( m\text{-direct}(T_1, A, B) \)
- decompose with \( m\text{-via}(T_1, A, B) \)
- decompose with \( m\text{-noop}(T_1, B) \)

**open prec.: at\( (T_1, B) \) of drop\( (T_1, B, P_1) \)**
- decompose get-to\( (T_1, B) \) with \( m\text{-direct}(T_1, A, B) \)
- decompose get-to\( (T_1, B) \) with \( m\text{-via}(T_1, A, B) \)
### HTN Plan Space Search

#### Standard Plan Space-based Algorithm

**Input:** fringe = \( \{ P_{\text{init}} \} \)

**Output:** A solution plan or fail.

1. \textbf{while} fringe \( \neq \emptyset \) \textbf{do}
2. \hspace{1em} \( P := \text{PlanSel}(\text{fringe}) \)
3. \hspace{1em} \( F := \text{FlawDet}(P) \)
4. \hspace{1em} \textbf{if} \( F = \emptyset \) \textbf{then} return \( P \)
5. \hspace{1em} \( f := \text{FlawSel}(F) \)
6. \hspace{1em} fringe := (fringe \setminus \{ P \}) \cup \text{Successors}(P, f)
7. \hspace{1em} \textbf{return} fail

### Flaws

**compound task:** get-to\((T_1, A)\)
- decompose with \( m\text{-direct}(T_1, B, A) \)
- decompose with \( m\text{-via}(T_1, B, A) \)
- decompose with \( m\text{-noop}(T_1, A) \)

**open prec.:** at\((T_1, A)\) of pick-up\((T_1, A, P_1)\)
- insert causal link from init
- decompose get-to\((T_1, A)\) with \( m\text{-direct}(T_1, B, A) \)
- decompose get-to\((T_1, A)\) with \( m\text{-via}(T_1, B, A) \)

**compound task:** get-to\((T_1, B)\)
- decompose with \( m\text{-direct}(T_1, A, B) \)
- decompose with \( m\text{-via}(T_1, A, B) \)
- decompose with \( m\text{-noop}(T_1, B) \)

**open prec.:** at\((T_1, B)\) of drop\((T_1, B, P_1)\)
- decompose get-to\((T_1, B)\) with \( m\text{-direct}(T_1, A, B) \)
- decompose get-to\((T_1, B)\) with \( m\text{-via}(T_1, A, B) \)

\[ ... \]
HTN Plan Space Search

Standard Plan Space-based Algorithm

Input : fringe = \{P_{\text{init}}\}
Output : A solution plan or fail.

while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P}) ∪ Successors(P, f)
return fail

Flaws

- compound task: get-to(T_1, A)
- open prec.: at(T_1, A) of pick-up(T_1, A, P_1)

 Modifications

- decompose with m-direct(T_1, B, A)
- decompose with m-via(T_1, B, A)
- decompose with m-noop(T_1, A)
- insert causal link from init
decompose get-to(T_1, A) with m-direct(T_1, B, A)
decompose get-to(T_1, A) with m-via(T_1, B, A)

compound task: get-to(T_1, B)
open prec.: at(T_1, B) of drop(T_1, B, P_1)
decompose get-to(T_1, B) with m-direct(T_1, A, B)
decompose get-to(T_1, B) with m-via(T_1, A, B)
decompose get-to(T_1, B) with m-noop(T_1, B)
decompose get-to(T_1, B) with m-direct(T_1, A, B)
decompose get-to(T_1, B) with m-via(T_1, A, B)
...
Input : fringe = \{P_{\text{init}}\}
Output: A solution plan or fail.

while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P}) ∪ Successors(P, f)
return fail

Flaws

<table>
<thead>
<tr>
<th>compound task: get-to(T_1, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifications</td>
</tr>
<tr>
<td>decompose with (m)-direct(T_1, B, A)</td>
</tr>
<tr>
<td>decompose with (m)-via(T_1, B, A)</td>
</tr>
<tr>
<td>decompose with (m)-noop(T_1, A)</td>
</tr>
</tbody>
</table>

open prec.: at(T_1, A) of pick-up(T_1, A, P_1)
insert causal link from \(\text{init}\)
decompose get-to(T_1, A) with \(m\)-direct(T_1, B, A)
de-compose get-to(T_1, A) with \(m\)-via(T_1, B, A)

compound task: get-to(T_1, B)
de-compose get-to(T_1, A) with \(m\)-direct(T_1, B, A)
de-compose get-to(T_1, A) with \(m\)-via(T_1, B, A)
de-compose get-to(T_1, A) with \(m\)-noop(T_1, B)

open prec.: at(T_1, B) of drop(T_1, B, P_1)
de-compose get-to(T_1, B) with \(m\)-direct(T_1, A, B)
de-compose get-to(T_1, B) with \(m\)-via(T_1, A, B)
de-compose get-to(T_1, B) with \(m\)-noop(T_1, B)

...
**Input**: fringe = \{P_{\text{init}}\}

**Output**: A solution plan or fail.

1. while fringe ≠ ∅ do
2. \[ P := \text{PlanSel}(\text{fringe}) \]
3. \[ F := \text{FlawDet}(P) \]
4. if F = ∅ then return P
5. \[ f := \text{FlawSel}(F) \]
6. fringe := (fringe \setminus \{P\}) \cup \text{Successors}(P, f)
7. return fail
Solving Techniques

Heuristics

Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

```
1 while fringe \neq \emptyset do
2     P := PlanSel(fringe)
3     F := FlawDet(P)
4     if F = \emptyset then return P
5     f := FlawSel(F)
6     fringe := (fringe \setminus \{P\}) \cup Successors(P, f)
7 8 return fail
```

**Flaws**

- compound task: get-to(T₁, A)
- open prec.: at(T₁, A) of pick-up(T₁, A, P₁)
- open prec.: at(T₁, A) of drive(T₁, A, B)
- open prec.: road(A, B) of drive(T₁, A, B)

**Modifications**

- decompose with m-direct(T₁, B, A)
- decompose with m-via(T₁, B, A)
- decompose with m-noop(T₁, A)
- insert causal link from init
decompose get-to(T₁, A) with m-direct(T₁, B, A)
decompose get-to(T₁, A) with m-via(T₁, B, A)
decompose get-to(T₁, A) with m-via(T₁, B, A)
insert causal link from init
insert causal link from init

...
Solving Techniques

Heuristics

Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

Input : fringe = \{P_{init}\}

Output : A solution plan or fail.

while fringe ≠ ∅ do
  P := PlanSel(fringe)
  F := FlawDet(P)
  if F = ∅ then return P
  f := FlawSel(F)
  fringe := (fringe \ {P}) ∪ Successors(P, f)

return fail
Solving Techniques

Heuristics

Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

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

June 25th, ICAPS 2018 (Delft) 9 / 62

Input: fringe = \{P_{\text{init}}\}

Output: A solution plan or \text{fail}.

1. while fringe $\neq \emptyset$ do
2. \hspace{1em} $P := \text{PlanSel}(\text{fringe})$
3. \hspace{1em} $F := \text{FlawDet}(P)$
4. \hspace{1em} if $F = \emptyset$ then return $P$
5. \hspace{2em} $f := \text{FlawSel}(F)$
6. \hspace{2em} fringe := (fringe \ \{P\})
7. \hspace{2.5em} $\cup \text{Successors}(P, f)$
8. return fail

Flaws

compound task: get-to(T₁, A)

modifications

decompose with \text{m-direct}(T₁, B, A)
decompose with \text{m-via}(T₁, B, A)
decompose with \text{m-noop}(T₁, A)

open prec.: at(T₁, A) of pick-up(T₁, A, P₁)

insert causal link from \text{init}
decompose get-to(T₁, A) with \text{m-direct}(T₁, B, A)
decompose get-to(T₁, A) with \text{m-via}(T₁, B, A)

open prec.: at(T₁, A) of drive(T₁, A, B)

insert causal link from \text{init}
decompose get-to(T₁, A) with \text{m-direct}(T₁, B, A)
decompose get-to(T₁, A) with \text{m-via}(T₁, B, A)

...
**Solving Techniques**

**Heuristics**

**Excursion**

---

**HTN Plan Space Search**

**Standard Plan Space-based Algorithm**

---

**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

1. while fringe ≠ ∅ do
2.   \(P := \text{PlanSel}(\text{fringe})\)
3.   \(F := \text{FlawDet}(P)\)
4.   if \(F = ∅\) then return \(P\)
5.   \(f := \text{FlawSel}(F)\)
6.   fringe := (fringe \ {P}) \cup \text{Successors}(P, f)
7. return fail

---

**Flaws**

- **compound task**: get-to\((T_1, A)\)
- **open prec.**: at\((T_1, A)\) of pick-up\((T_1, A, P_1)\)
- **open prec.**: at\((T_1, A)\) of drive\((T_1, A, B)\)

**Modifications**

- decompose with m-direct\((T_1, B, A)\)
- decompose with m-via\((T_1, B, A)\)
- decompose with m-noop\((T_1, A)\)
- insert causal link from init
- decompose get-to\((T_1, A)\) with m-direct\((T_1, B, A)\)
- decompose get-to\((T_1, A)\) with m-via\((T_1, B, A)\)
- insert causal link from init
- decompose get-to\((T_1, A)\) with m-direct\((T_1, B, A)\)
- decompose get-to\((T_1, A)\) with m-via\((T_1, B, A)\)

---

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

June 25th, ICAPS 2018 (Delft)
### HTN Plan Space Search

#### Standard Plan Space-based Algorithm

**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

```plaintext
while fringe ≠ ∅ do
  P := PlanSel(fringe)
  F := FlawDet(P)
  if F = ∅ then return P
  f := FlawSel(F)
  fringe := (fringe \ {P}) ∪ Successors(P,f)
return fail
```

#### Flaws

- **compound task**: get-to(T₁, A)
  - Decompose with m-direct(T₁, B, A)
  - Decompose with m-via(T₁, B, A)
  - Decompose with m-noop(T₁, A)

- **open prec.**: at(T₁, A) of pick-up(T₁, A, P₁)
  - Insert causal link from init
    - Decompose get-to(T₁, A) with m-direct(T₁, B, A)
    - Decompose get-to(T₁, A) with m-via(T₁, B, A)

- **open prec.**: at(T₁, A) of drive(T₁, A, B)
  - Insert causal link from init
    - Decompose get-to(T₁, A) with m-direct(T₁, B, A)
    - Decompose get-to(T₁, A) with m-via(T₁, B, A)
Solving Techniques

Heuristics

Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

Input : fringe = \{P_{init}\}

Output : A solution plan or fail.

1. while fringe ≠ ∅ do
2.     P := PlanSel(fringe)
3.     F := FlawDet(P)
4.     if F = ∅ then return P
5.     f := FlawSel(F)
6.     fringe := (fringe \ \{P\})
7.     \cup Successors(P, f)
8. return fail

Flaws

compound task: get-to(T_1, A)

decompose with m-direct(T_1, B, A)
decompose with m-via(T_1, B, A)
decompose with m-noop(T_1, A)

open prec.: at(T_1, A) of pick-up(T_1, A, P_1)

insert causal link from init
decompose get-to(T_1, A) with m-direct(T_1, B, A)
decompose get-to(T_1, A) with m-via(T_1, B, A)

open prec.: at(T_1, A) of drive(T_1, A, B)

insert causal link from init
decompose get-to(T_1, A) with m-direct(T_1, B, A)
decompose get-to(T_1, A) with m-via(T_1, B, A)

...
**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

```python
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P}) ∪ Successors(P, f)
return fail
```

**Flaws**

- **open prec:** at(T₁, B) of drive(T₁, B, A)
- **open prec.:** road(B, A) of drive(T₁, B, A)
- **open prec.:** at(T₁, A) of pick-up(T₁, A, P₁)

**Modifications**

- insert causal link from init
- insert causal link from drive(T₁, B, A)
- insert causal link from init
- insert causal link from drive(T₁, B, A)
HTN Plan Space Search

Standard Plan Space-based Algorithm

Input: fringe = \{ P_{\text{init}} \}

Output: A solution plan or fail.

1 while fringe \neq \emptyset do
2 \hspace{1em} P := \text{PlanSel}(fringe)
3 \hspace{1em} F := \text{FlawDet}(P)
4 \hspace{1em} if F = \emptyset then return P
5 \hspace{1em} f := \text{FlawSel}(F)
6 \hspace{1em} fringe := (fringe \setminus \{ P \}) \cup \text{Successors}(P, f)
7 \hspace{1em} return fail

Flaws

- \textbf{open prec: at(T_1, B) of drive(T_1, B, A)}
- \textbf{open prec.: road(B, A) of drive(T_1, B, A)}
- \textbf{open prec.: at(T_1, A) of pick-up(T_1, A, P_1)}
- \textbf{open prec.: at(T_1, A) of drive(T_1, A, B)}

Modifications

- insert causal link from init
- insert causal link from drive(T_1, B, A)
- insert causal link from init
- insert causal link from drive(T_1, B, A)

This partial plan can be discarded, because it has a flaw without modifications.
Solving Techniques

Heuristics

Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

Flaws

compound task: get-to($T_1$, $A$)

Modifications

decompose with $m$-direct($T_1$, $B$, $A$)
decompose with $m$-via($T_1$, $B$, $A$)
decompose with $m$-noop($T_1$, $A$)

open prec.: at($T_1$, $A$) of pick-up($T_1$, $A$, $P_1$)

insert causal link from init
decompose get-to($T_1$, $A$) with $m$-direct($T_1$, $B$, $A$)
decompose get-to($T_1$, $A$) with $m$-via($T_1$, $B$, $A$)

open prec.: at($T_1$, $A$) of drive($T_1$, $A$, $B$)

insert causal link from init
decompose get-to($T_1$, $A$) with $m$-direct($T_1$, $B$, $A$)
decompose get-to($T_1$, $A$) with $m$-via($T_1$, $B$, $A$)

... ...

Input : fringe = {$P_{\text{init}}$}
Output : A solution plan or fail.

while fringe $\neq \emptyset$ do

$P := \text{PlanSel}(\text{fringe})$

$F := \text{FlawDet}(P)$

if $F = \emptyset$ then return $P$

$f := \text{FlawSel}(F)$

$\text{fringe} := (\text{fringe} \setminus \{P\}) \cup \text{Successors}(P, f)$

return fail

... ...
**Solving Techniques**

**Heuristics**

**Excursion**

---

**HTN Plan Space Search**

**Standard Plan Space-based Algorithm**

---

**Input**: fringe = \{ P_{init} \}

**Output**: A solution plan or fail.

1. while fringe ≠ ∅ do
2. 
   \[ P := \text{PlanSel}(\text{fringe}) \]
3. 
   \[ F := \text{FlawDet}(P) \]
4. 
   if F = ∅ then return P
5. 
   \[ f := \text{FlawSel}(F) \]
6. 
   fringe := (fringe \ {P}) ∪ Successors(P, f)
7. return fail

---

**Flaws**

**Modifications**

<table>
<thead>
<tr>
<th>Flaws</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>open prec.: ( at(T_1, A) ) of ( \text{pick-up}(T_1, A, P_1) )</td>
<td>insert causal link from \text{init}</td>
</tr>
<tr>
<td>open prec.: ( at(T_1, A) ) of ( \text{drive}(T_1, A, B) )</td>
<td>insert causal link from \text{init}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

---

Diagram showing the plan space search and flaws with modifications.
Input: fringe = \{P_{init}\}
Output: A solution plan or fail.

1 while fringe ≠ ∅ do
2 P := PlanSel(fringe)
3 F := FlawDet(P)
4 if F = ∅ then return P
5 f := FlawSel(F)
6 fringe := (fringe \ {P}) ∪ Successors(P, f)
7 return fail

Flaws Modifications
open prec.: at(T1, A) of pick-up(T1, A, P1) insert causal link from init
open prec.: at(T1, A) of drive(T1, A, B) insert causal link from init
... ...
... ...
Input : fringe = \{P_{init}\}

Output : A solution plan or fail.

1 while fringe ≠ ∅ do
2 \hspace{1em} P := PlanSel(fringe)
3 \hspace{1em} F := FlawDet(P)
4 \hspace{1em} if F = ∅ then return P
5 \hspace{1em} f := FlawSel(F)
6 \hspace{1em} fringe := (fringe \{P\})
7 \hspace{2.5em} \cup Successors(P, f)
8 return fail
Input : fringe = \{P_{\text{init}}\}

Output : A solution plan or fail.

1 \textbf{while} fringe \neq \emptyset \textbf{do}
2 \quad P := \text{PlanSel}(\text{fringe})
3 \quad F := \text{FlawDet}(P)
4 \quad \textbf{if} F = \emptyset \textbf{then return} P
5 \quad f := \text{FlawSel}(F)
6 \quad \text{fringe} := (\text{fringe} \setminus \{P\}) \cup \text{Successors}(P, f)
7 \textbf{return fail}
**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

1. **while** fringe ≠ ∅ **do**
   2. \( P := \text{PlanSel}(\text{fringe}) \)
   3. \( F := \text{FlawDet}(P) \)
   4. **if** \( F = \emptyset \) **then** return \( P \)
   5. \( f := \text{FlawSel}(F) \)
   6. \( \text{fringe} := (\text{fringe} \setminus \{P\}) \cup \text{Successors}(P, f) \)
   7. **return** fail

**Flaws**

<table>
<thead>
<tr>
<th>Flaws</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Definition**

HTN Plan Space Search

**Standard Plan Space-based Algorithm**

- **Heuristics**
- **Excursion**

---

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

June 25th, ICAPS 2018 (Delft)
**Input**: fringe = \{P_{init}\}

**Output**: A solution plan or fail.

1. \textbf{while} fringe \neq \emptyset \textbf{do}
2. \hspace{1em} P := \text{PlanSel}(fringe)
3. \hspace{1em} F := FlawDet(P)
4. \hspace{1em} \textbf{if} F = \emptyset \textbf{then} return P
5. \hspace{2em} f := FlawSel(F)
6. \hspace{1em} fringe := (fringe \setminus \{P\})
7. \hspace{2em} \cup \text{Successors}(P, f)
8. return fail
Input : fringe = \{P_{init}\}
Output : A solution plan or fail.

1 while fringe ≠ ∅ do
2 \hspace{1cm} P := PlanSel(fringe)
3 \hspace{1cm} F := FlawDet(P)
4 \hspace{1cm} if F = ∅ then return P
5 \hspace{1cm} f := FlawSel(F)
6 \hspace{1cm} fringe := (fringe \setminus \{P\}) \cup Successors(P, f)
7 \hspace{1cm} return fail
Input : \( \text{fringe} = \{P_{\text{init}}\} \)

Output : A solution plan or fail.

```plaintext
while fringe ≠ ∅ do
  P := PlanSel(fringe)
  F := FlawDet(P)
  if F = ∅ then return P
  f := FlawSel(F)
  fringe := (fringe \ {P}) ∪ Successors(P, f)
return fail
```

This partial plan has no flaws, so it is a solution and returned.
Plan Space-based Search – Properties

- Plan Space-based search is **sound**
Plan Space-based Search – Properties

- Plan Space-based search is **sound**
- ... and **complete** (completeness only depends on the plan selection function (fringe sorting), but not on the flaw selection function)
Plan Space-based search is sound

... and complete (completeness only depends on the plan selection function (fringe sorting), but not on the flaw selection function)

There is no current state during search (the initial state is never changed)
Plan Space-based Search – Properties

- Plan Space-based search is **sound**
- ... and **complete** (completeness only depends on the plan selection function (fringe sorting), but not on the flaw selection function)
- There is **no current state** during search (the initial state is never changed)
- Tasks are partially ordered and can be inserted anywhere in a partial plan
Solving HTN Planning Problems

- **Search-based Approaches**
  - Plan Space Search
  - **Progression Search**

- **Compilation-based Approaches**
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- **Heuristics for Heuristic Search**
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

**Excursion**

- **Further Hierarchical Planning Formalisms**
Only those (primitive or compound) tasks in a task network that have **no predecessor** in the ordering relations are processed.

Actions that are processed are **removed** from the network and cause **state transition**.
Only those (primitive or compound) tasks in a task network that have **no predecessor** in the ordering relations are processed.

Actions that are processed are **removed** from the network and cause **state transition**.

→ Search nodes contain the current task network and state.
Only those (primitive or compound) tasks in a task network that have **no predecessor** in the ordering relations are processed.

- Actions that are processed are **removed** from the network and cause **state transition**

→ Search nodes contain the current task network **and** state

→ **Commitment** to the prefix of the solution during search
Only those (primitive or compound) tasks in a task network that have no predecessor in the ordering relations are processed.

Actions that are processed are removed from the network and cause state transition.

- Search nodes contain the current task network and state.
- Commitment to the prefix of the solution during search.
- We are searching for an empty task network.
**HTN Progression Search**

**Standard Progression Algorithm**

```plaintext
1 fringe ← \{(s_I, tn_I, ())\}
2 while fringe ≠ ∅ do
3 n ← fringe.poll()
4 if n.isgoal then return n
5 U ← n.unconstrainedNodes
6 for t ∈ U do
7   if isPrimitive(t) then
8     n' ← n.apply(t)
9     fringe.add(n')
10 else
11    for m ∈ t.methods do
12      n' ← n.decompose(t, m)
13      fringe.add(n')
```
### Standard Progression Algorithm

1. `fringe ← \{(s_I, tn_I, ())\}
2. while \(fringe \neq \emptyset\) do
3. \(n ← fringe.poll()\)
4. if `\(n.isgoal\)` then return `\(n\)`
5. `\(U ← n.unconstrainedNodes\)`
6. for \(t ∈ U\) do
7. if `\(isPrimitive(t)\)` then
8. \(n' ← n.apply(t)\)
9. `fringe.add(n')`
10. else
11. for \(m ∈ t.methods\) do
12. \(n' ← n.decompose(t, m)\)
13. `fringe.add(n')`

- **Search nodes** contain task network, state, and solution prefix
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

1. fringe ← \{(s_I, t_n, ())\}
2. while fringe ≠ ∅ do
3.   n ← fringe.poll()
4.   if n.isgoal then return n
5.   U ← n.unconstrainedNodes
6.   for t ∈ U do
7.     if isPrimitive(t) then
8.       n' ← n.apply(t)
9.       fringe.add(n')
10.    else
11.      for m ∈ t.methods do
12.        n' ← n.decompose(t, m)
13.        fringe.add(n')

- **Search nodes** contain task network, state, and solution prefix
- **Fringe** is sorted according to some heuristic
HTN Progression Search

Standard Progression Algorithm

1. \( fringe \leftarrow \{(s_1, tn_1, ())\} \)
2. \( \text{while } fringe \neq \emptyset \text{ do } \)
3. \( n \leftarrow fringe.poll() \)
4. \( \text{if } n.isgoal \text{ then return } n \)
5. \( U \leftarrow n.unconstrainedNodes \)
6. \( \text{for } t \in U \text{ do } \)
7. \( \text{if isPrimitive}(t) \text{ then } \)
8. \( n' \leftarrow n.apply(t) \)
9. \( fringe.add(n') \)
10. \( \text{else } \)
11. \( \text{for } m \in t.methods \text{ do } \)
12. \( n' \leftarrow n.decompose(t, m) \)
13. \( fringe.add(n') \)

- **Search nodes** contain task network, state, and solution prefix
- **Fringe** is sorted according to some heuristic
- **Goal test** checks for empty task network (maybe for a goal state)
**HTN Progression Search**

**Standard Progression Algorithm**

1. $\text{fringe} \leftarrow \{(s_I, tn_I, (\))\}$
2. while $\text{fringe} \neq \emptyset$ do
   3. $n \leftarrow \text{fringe.poll()}$
   4. if $n.\text{isgoal}$ then return $n$
   5. $U \leftarrow n.\text{unconstrainedNodes}$
   6. for $t \in U$ do
      7. if $\text{isPrimitive}(t)$ then
         8. $n' \leftarrow n.\text{apply}(t)$
         9. $\text{fringe.add}(n')$
      else
         10. for $m \in t.\text{methods}$ do
             11. $n' \leftarrow n.\text{decompose}(t, m)$
             12. $\text{fringe.add}(n')$

- **Search nodes** contain task network, state, and solution prefix
- **Fringe** is sorted according to some heuristic
- **Goal test** checks for empty task network (maybe for a goal state)
- **Unconstrained tasks** have no predecessor
```
1 fringe ← \{(s_I, tn_I, ())\}
2 while fringe ≠ ∅ do
3     n ← fringe.poll()
4     if n.isgoal then return n
5     U ← n.unconstrainedNodes
6     for t ∈ U do
7         if isPrimitive(t) then
8             n' ← n.apply(t)
9             fringe.add(n')
10        else
11            for m ∈ t.methods do
12                n' ← n.decompose(t, m)
13                fringe.add(n')
```
Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning

June 25th, ICAPS 2018 (Delft)

14 / 62

1. fringe ← \{(s_i, t_n, (\))\}
2. while fringe ≠ ∅ do
3.   n ← fringe.poll()
4. if n.isgoal then return n
5.   U ← n.unconstrainedNodes
6. for t ∈ U do
7.     if isPrimitive(t) then
8.       n' ← n.apply(t)
9.       fringe.add(n')
10.   else
11.     for m ∈ t.methods do
12.         n' ← n.decompose(t, m)
13.         fringe.add(n')

π = ()

deliver(P_1, B)

deliver(P_2, D)
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

\[ \text{fringe} \leftarrow \{ (s_i, t_i, () ) \} \]

\[ \text{while} \ \text{fringe} \neq \emptyset \ \text{do} \]

\[ n \leftarrow \text{fringe.poll()} \]

\[ \text{if} \ n \text{.isgoal} \ \text{then} \ \text{return} \ n \]

\[ U \leftarrow n \text{.unconstrainedNodes} \]

\[ \text{for} \ t \in U \ \text{do} \]

\[ \text{if} \ \text{isPrimitive}(t) \ \text{then} \]

\[ n' \leftarrow n \text{.apply}(t) \]

\[ \text{fringe.add}(n') \]

\[ \text{else} \]

\[ \text{for} \ m \in t \text{.methods} \ \text{do} \]

\[ n' \leftarrow n \text{.decompose}(t, m) \]

\[ \text{fringe.add}(n') \]

\[ \pi = () \]

\[ \text{deliver}(P_1, B) \]

\[ m \text{-deliver}(P_1, A, B, T_1) \]

\[ \text{get-to}(T_1, A) \rightarrow \text{pick-up}(T_1, A, P_1) \rightarrow \text{get-to}(T_1, B) \rightarrow \text{drop}(T_1, B, P_1) \]

\[ \text{deliver}(P_2, D) \]
HTN Progression Search

**Standard Progression Algorithm**

\[
\begin{align*}
&\text{fringe} \leftarrow \{(s_i, t_n, ())\} \\
&\text{while fringe} \neq \emptyset \text{ do} \\
&\quad n \leftarrow \text{fringe.poll}() \\
&\quad \text{if } n.\text{isgoal} \text{ then return } n \\
&\quad U \leftarrow n.\text{unconstrainedNodes} \\
&\quad \text{for } t \in U \text{ do} \\
&\quad\quad \text{if isPrimitive}(t) \text{ then} \\
&\quad\quad\quad n' \leftarrow n.\text{apply}(t) \\
&\quad\quad\quad \text{fringe.add}(n') \\
&\quad\text{else} \\
&\quad\quad \text{for } m \in t.\text{methods} \text{ do} \\
&\quad\quad\quad n' \leftarrow n.\text{decompose}(t, m) \\
&\quad\quad\quad \text{fringe.add}(n') \\
&\end{align*}
\]

\[\pi = ()\]
1. \( \text{fringe} \leftarrow \{(s, t, n, ())\} \)

2. \textbf{while} \( \text{fringe} \neq \emptyset \) \textbf{do}

3. \( n \leftarrow \text{fringe.poll()} \)

4. \textbf{if} \( n \text{.isgoal} \) \textbf{then} \textbf{return} \( n \)

5. \( U \leftarrow n \text{.unconstrainedNodes} \)

6. \textbf{for} \( t \in U \) \textbf{do}

7. \textbf{if} \( \text{isPrimitive}(t) \) \textbf{then}

8. \( n' \leftarrow n \text{.apply}(t) \)

9. \( \text{fringe.add}(n') \)

10. \textbf{else}

11. \textbf{for} \( m \in t \text{.methods} \) \textbf{do}

12. \( n' \leftarrow n \text{.decompose}(t, m) \)

13. \( \text{fringe.add}(n') \)
1. fringe ← \{(s, tn, (\))\}
2. while fringe ≠ ∅ do
   3. n ← fringe.poll()
   4. if n.isgoal then return n
   5. U ← n.unconstrainedNodes
   6. for t ∈ U do
      7. if isPrimitive(t) then
         8. n' ← n.apply(t)
         9. fringe.add(n')
      10. else
          11. for m ∈ t.methods do
              12. n' ← n.decompose(t, m)
              13. fringe.add(n')

π = ( )

deliver(P_1, B)

goto(T_2, C) ≺ \{pick-up(T_2, C, P_2)\} ≺ goto(T_2, D) ≺ \{drop(T_2, D, P_2)\}
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

1. fringe ← \{(s_i, t_{n_i}, ())\}
2. while fringe ≠ ∅ do
   3. n ← fringe.poll()
   4. if n.isgoal then return n
   5. U ← n.unconstrainedNodes
   6. for t ∈ U do
      7. if isPrimitive(t) then
         8. n' ← n.apply(t)
         fringe.add(n')
      else
         9. for m ∈ t.methods do
            10. n' ← n.decompose(t, m)
            fringe.add(n')
   
\[ \pi = () \]

get-to(T_1, A) \rhd \ldots \rhd pick-up(T_1, A, P_1) \rhd \ldots \rhd get-to(T_1, B) \rhd \ldots \rhd drop(T_1, B, P_1)

deliver(P_2, D)
Standard Progression Algorithm

1. $\text{fringe} \leftarrow \{(s_1, t_{n_1}, (\))\}
2. while $\text{fringe} \neq \emptyset$ do
3.     $n \leftarrow \text{fringe.poll}()$
4.     if $n$ is goal then return $n$
5.     $U \leftarrow n.unconstrainedNodes$
6.     for $t \in U$ do
7.         if $\text{isPrimitive}(t)$ then
8.             $n' \leftarrow n.apply(t)$
9.             $\text{fringe}.add(n')$
10.        else
11.            for $m \in t.methods$ do
12.                $n' \leftarrow n.decompose(t, m)$
13.                $\text{fringe}.add(n')$
14.     $\pi = ()$
HTN Progression Search

Standard Progression Algorithm

1. fringe ← \{(s, tn, ()): fringe.poll()
2. while fringe ≠ ∅ do
3.   n ← fringe.poll()
4.   if n.isgoal then return n
5.   U ← n.unconstrainedNodes
6.   for t ∈ U do
7.     if isPrimitive(t) then
8.       n′ ← n.apply(t)
9.       fringe.add(n′)
10.    else
11.      for m ∈ t.methods do
12.         n′ ← n.decompose(t, m)
13.         fringe.add(n′)

\[ \pi = () \]

Example HTN:

- **pick-up** \((T_1, A, P_1)\)
- **get-to** \((T_1, B)\)
- **drop** \((T_1, B, P_1)\)
- **deliver** \((P_2, D)\)
- **no-op** ()

Diagram:

```
  B
  /|
  / |
P_2 C T_2
  |
  |
  P_1 A T_1
  |
  |
  D
```
1 fringe ← \{(s, tn, ())\}
2 while fringe ≠ ∅ do
3 n ← fringe.poll()
4 if n.isgoal then return n
5 U ← n.unconstrainedNodes
6 for t ∈ U do
7   if isPrimitive(t) then
8     n' ← n.apply(t)
9     fringe.add(n')
10 else
11   for m ∈ t.methods do
12     n' ← n.decompose(t, m)
13     fringe.add(n')

π = ()

get-to(T_1, A) ... \text{pick-up}(T_1, A, P_1) ... get-to(T_1, B) ... \text{drop}(T_1, B, P_1)

deliver(P_2, D)
m-deliver(P_2, C, D, T_2)

get-to(T_2, C) ... \text{pick-up}(T_2, C, P_2) ... get-to(T_2, D) ... \text{drop}(T_2, D, P_2)
**HTN Progression Search**

**Standard Progression Algorithm**

1. \( \text{fringe} \leftarrow \{(s, t, n, ())\} \)
2. \( \text{while} \ \text{fringe} \neq \emptyset \) \( \text{do} \)
   3. \( n \leftarrow \text{fringe.poll()} \)
   4. \( \text{if} \ n.\text{isgoal} \ \text{then} \ \text{return} \ n \)
   5. \( U \leftarrow n.\text{unconstrainedNodes} \)
   6. \( \text{for} \ t \in U \ \text{do} \)
      7. \( \text{if} \ \text{isPrimitive}(t) \ \text{then} \)
         8. \( n' \leftarrow n.\text{apply}(t) \)
         9. \( \text{fringe.add}(n') \)
      10. \( \text{else} \)
          11. \( \text{for} \ m \in t.\text{methods} \ \text{do} \)
              12. \( n' \leftarrow n.\text{decompose}(t, m) \)
              13. \( \text{fringe.add}(n') \)

\( \pi = () \)

get-to\((T_1, A)\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) pick-up\((T_1, A, P_1)\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) get-to\((T_1, B)\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) drop\((T_1, B, P_1)\)

get-to\((T_2, C)\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) pick-up\((T_2, C, P_2)\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) get-to\((T_2, D)\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) \(\prec\) drop\((T_2, D, P_2)\)
```
1 fringe ← \{(s, tn, (\))\}
2 while fringe ≠ ∅ do
3    n ← fringe.poll()
4    if n.isgoal then return n
5    U ← n.unconstrainedNodes
6    for t ∈ U do
7       if isPrimitive(t) then
8          n' ← n.apply(t)
9          fringe.add(n')
10      else
11         for m ∈ t.methods do
12            n' ← n.decompose(t, m)
13            fringe.add(n')
```

\[ \pi = () \]
**Standard Progression Algorithm**

1. $\text{fringe} \leftarrow \{(s, t, n, ())\}$
2. while $\text{fringe} \neq \emptyset$
   3. $n \leftarrow \text{fringe.poll()}$
   4. if $n.isGoal$ then return $n$
   5. $U \leftarrow n.unconstrainedNodes$
   6. for $t \in U$
      7. if $isPrimitive(t)$ then
         8. $n' \leftarrow n.apply(t)$
         9. $\text{fringe.add}(n')$
      else
         10. for $m \in t.methods$
             11. $n' \leftarrow n.decompose(t, m)$
             12. $\text{fringe.add}(n')$

\[
\pi = ()
\]
1. fringe ← {(s₁, tn₁, ())}
2. while fringe ≠ ∅ do
3.     n ← fringe.poll()
4.     if n.isgoal then return n
5.     U ← n.unconstrainedNodes
6.     for t ∈ U do
7.         if isPrimitive(t) then
8.             n' ← n.apply(t)
9.             fringe.add(n')
10.     else
11.         for m ∈ t.methods do
12.             n' ← n.decompose(t, m)
13.             fringe.add(n')

π = ()
**HTN Progression Search**

**Standard Progression Algorithm**

1. \( fringe \leftarrow \{(s, tn, ()): fringe.poll()\} \)
2. \( \text{while } fringe \neq \emptyset \text{ do} \)
3. \( n \leftarrow fringe.poll() \)
4. \( \text{if } n\.isgoal \text{ then return } n \)
5. \( U \leftarrow n\.unconstrainedNodes \)
6. \( \text{for } t \in U \text{ do} \)
7. \( \text{if } isPrimitive(t) \text{ then} \)
8. \( n' \leftarrow n\.apply(t) \)
9. \( fringe\.add(n') \)
10. \( \text{else} \)
11. \( \text{for } m \in t\.methods \text{ do} \)
12. \( n' \leftarrow n\.decompose(t, m) \)
13. \( fringe\.add(n') \)

\[ \pi = () \]

```
fringe ← {(s, tn, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```

![Diagram of HTN Progression Search](image.png)
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

```
1 fringe ← \{(s, tn, ( ))\}
2 while fringe \neq \emptyset do
3     n ← fringe.poll()
4     if n.isgoal then return n
5     U ← n.unconstrainedNodes
6     for t ∈ U do
7         if isPrimitive(t) then
8             n' ← n.apply(t)
9             fringe.add(n')
10        else
11            for m ∈ t.methods do
12                n' ← n.decompose(t, m)
13                fringe.add(n')
14 get-to(T1, A) ≺ pick-up(T1, A, P1) ≺ ... get-to(T1, B) ≺ drop(T1, B, P1)
15 get-to(T2, C) ≺ pick-up(T2, C, P2) ≺ ... get-to(T2, D) ≺ drop(T2, D, P2)
16 π = ()
```
**HTN Progression Search**

**Standard Progression Algorithm**

1. Initialize the fringe with the initial state.
2. While the fringe is not empty, do:
   - Remove the next node from the fringe.
   - If the node is a goal, return it.
   - Otherwise, for each unconstrained node:
     - If it is primitive, apply it and add the new node to the fringe.
     - Otherwise, for each method:
       - Decompose the node and add the new node to the fringe.

```plaintext
 fringe ← {(s, tn, ())}
 while fringe ≠ ∅ do
   n ← fringe.poll()
   if n.isgoal then return n
   U ← n.unconstrainedNodes
   for t ∈ U do
     if isPrimitive(t) then
       n' ← n.apply(t)
       fringe.add(n')
     else
       for m ∈ t.methods do
         n' ← n.decompose(t, m)
         fringe.add(n')
```

π = ()

Example:

- get-to(T1, A) \( \prec \) pick-up(T1, A, P1) \( \prec \) get-to(T1, B) \( \prec \) drop(T1, B, P1)
- get-to(T2, C) \( \prec \) pick-up(T2, C, P2) \( \prec \) get-to(T2, D) \( \prec \) drop(T2, D, P2)

Example problem:

```
get-to(T1, A) \( \prec \) pick-up(T1, A, P1) \( \prec \) get-to(T1, B) \( \prec \) drop(T1, B, P1)
get-to(T2, C) \( \prec \) pick-up(T2, C, P2) \( \prec \) get-to(T2, D) \( \prec \) drop(T2, D, P2)
```
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

\[ \text{fringe} \leftarrow \{ (s, \text{tn}, (\)) \} \]

\[ \text{while } \text{fringe} \neq \emptyset \text{ do} \]

\[ \quad n \leftarrow \text{fringe.poll()} \]

\[ \quad \text{if } n.\text{isgoal} \text{ then return } n \]

\[ \quad U \leftarrow n.\text{unconstrainedNodes} \]

\[ \quad \text{for } t \in U \text{ do} \]

\[ \quad \quad \text{if } \text{isPrimitive}(t) \text{ then} \]

\[ \quad \quad \quad n' \leftarrow n.\text{apply}(t) \]

\[ \quad \quad \quad \text{fringe.add}(n') \]

\[ \quad \text{else} \]

\[ \quad \quad \text{for } m \in t.\text{methods} \text{ do} \]

\[ \quad \quad \quad n' \leftarrow n.\text{decompose}(t, m) \]

\[ \quad \quad \quad \text{fringe.add}(n') \]

\[ \pi = () \]

\[ \text{get-to}(T_1, A) \searrow \text{pick-up}(T_1, A, P_1) \searrow \text{get-to}(T_1, B) \searrow \text{drop}(T_1, B, P_1) \]

\[ \text{get-to}(T_2, C) \searrow \text{pick-up}(T_2, C, P_2) \searrow \text{get-to}(T_2, D) \searrow \text{drop}(T_2, D, P_2) \]

\[ \text{B} \quad P_2 \quad \square \quad C \quad T_2 \quad P_1 \quad \square \quad A \quad T_1 \quad D \]
**HTN Progression Search**

**Standard Progression Algorithm**

1. \( \text{fringe} \leftarrow \{(s_1, t(n_1, ()))\} \)
2. \( \text{while fringe} \neq \emptyset \) do
   3. \( n \leftarrow \text{fringe.poll()} \)
   4. if \( n.\text{isgoal} \) then return \( n \)
   5. \( U \leftarrow n.\text{unconstrainedNodes} \)
   6. for \( t \in U \) do
      7. if \( \text{isPrimitive}(t) \) then
         8. \( n' \leftarrow n.\text{apply}(t) \)
         9. fringe.add(\( n' \))
      10. else
          11. for \( m \in t.\text{methods} \) do
              12. \( n' \leftarrow n.\text{decompose}(t, m) \)
              13. fringe.add(\( n' \))

\[ \pi = () \]

\[ \begin{align*}
\text{get-to}(T_1, A) & \prec \text{pick-up}(T_1, A, P_1) \prec \text{get-to}(T_1, B) \prec \text{drop}(T_1, B, P_1) \\
\text{get-to}(T_2, C) & \prec \text{m-noop}(T_2, C) \prec \text{pick-up}(T_2, C, P_2) \prec \text{get-to}(T_2, D) \prec \text{drop}(T_2, D, P_2) \\
\text{no-op}() & \\
\end{align*} \]
Standard Progression Algorithm

1. \( fringe \leftarrow \{(s, t(n_1, ()))\} \)
2. \( \text{while } fringe \neq \emptyset \) do
3. \( n \leftarrow fringe.\text{poll}() \)
4. \( \text{if } n.\text{isgoal} \text{ then return } n \)
5. \( U \leftarrow n.\text{unconstrainedNodes} \)
6. \( \text{for } t \in U \text{ do} \)
7. \( \text{if } \text{isPrimitive}(t) \text{ then} \)
8. \( n' \leftarrow n.\text{apply}(t) \)
9. \( fringe.\text{add}(n') \)
10. \( \text{else} \)
11. \( \text{for } m \in t.\text{methods} \text{ do} \)
12. \( n' \leftarrow n.\text{decompose}(t, m) \)
13. \( fringe.\text{add}(n') \)

\( \pi = () \)

get-to(\( T_1, A \)) \揠
pick-up(\( T_1, A, P_1 \)) \揠
get-to(\( T_1, B \)) \揠
\( \text{drop}(T_1, B, P_1) \)

pick-up(\( T_2, C, P_2 \)) \揠
get-to(\( T_2, D \)) \揠
\( \text{drop}(T_2, D, P_2) \)

no-op()
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

```
fringe ← \{(s_i, t_n, ())\}

while fringe ≠ ∅ do

    n ← fringe.poll()

    if n.isgoal then return n

    U ← n.unconstrainedNodes

    for t ∈ U do

        if isPrimitive(t) then

            n' ← n.apply(t)

            fringe.add(n')

        else

            for m ∈ t.methods do

                n' ← n.decompose(t, m)

                fringe.add(n')
```

\[\pi = (\text{no-op}())\]
HTN Progression Search

Standard Progression Algorithm

1. fringe $\leftarrow \{(s, tn((),))\}$
2. while fringe $\neq \emptyset$ do
3.   $n \leftarrow$ fringe.poll()
4.   if $n$.isgoal then return $n$
5.   $U \leftarrow n$.unconstrainedNodes
6.   for $t \in U$ do
7.     if isPrimitive($t$) then
8.       $n' \leftarrow n$.apply($t$)
9.       fringe.add($n'$)
10.    else
11.       for $m \in t$.methods do
12.         $n' \leftarrow n$.decompose($t, m$)
13.         fringe.add($n'$)

$\pi = (\text{no-op}())$

get-to($T_1, A$), m-noop($T_1, A$), pick-up($T_1, A, P_1$), get-to($T_1, B$), drop($T_1, B, P_1$)
Standard Progression Algorithm

\[ fringe \leftarrow \{(s_i, t_n, \lambda)\} \]

\[ \text{while } fringe \neq \emptyset \text{ do} \]

\[ n \leftarrow fringe.poll() \]

\[ \text{if } n.\text{isgoal} \text{ then return } n \]

\[ U \leftarrow n.\text{unconstrainedNodes} \]

\[ \text{for } t \in U \text{ do} \]

\[ \text{if } isPrimitive(t) \text{ then} \]

\[ n' \leftarrow n.\text{apply}(t) \]

\[ fringe.\text{add}(n') \]

\[ \text{else} \]

\[ \text{for } m \in t.\text{methods} \text{ do} \]

\[ n' \leftarrow n.\text{decompose}(t, m) \]

\[ fringe.\text{add}(n') \]

\[ \pi = (\text{no-op}()) \]

\[ \text{pick-up}(T_1, A, P_1) \ldots \text{get-to}(T_1, B) \ldots \text{drop}(T_1, B, P_1) \]

\[ \text{no-op}() \]

\[ \text{pick-up}(T_2, C, P_2) \ldots \text{get-to}(T_2, D) \ldots \text{drop}(T_2, D, P_2) \]
1. \(\text{fringe} \leftarrow \{(s_i, t_n, ())\}\)

2. \(\text{while fringe} \neq \emptyset\) do

3. \(n \leftarrow \text{fringe.poll()}\)

4. if \(n.\text{isgoal}\) then return \(n\)

5. \(U \leftarrow n.\text{unconstrainedNodes}\)

6. for \(t \in U\) do

7. if \(\text{isPrimitive}(t)\) then

8. \(n' \leftarrow n.\text{apply}(t)\)

9. \(\text{fringe.add}(n')\)

10. else

11. for \(m \in t.\text{methods}\) do

12. \(n' \leftarrow n.\text{decompose}(t, m)\)

13. \(\text{fringe.add}(n')\)

\[\pi = (\text{no-op()}, \text{no-op()})\]
**Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning**

**June 25th, ICAPS 2018 (Delft)**

---

**HTN Progression Search**

**Standard Progression Algorithm**

```plaintext
fringe ← {((s, tn, ()))}

while fringe ≠ ∅ do
  n ← fringe.poll()
  if n.isgoal then return n
  U ← n.unconstrainedNodes
  for t ∈ U do
    if isPrimitive(t) then
      n' ← n.apply(t)
      fringe.add(n')
    else
      for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')

π = (no-op(), no-op(), pick-up(T1, A, P1))

get-to(T1, B) ≺ get-to(T2, D) ≺ drop(T1, B, P1) ≺ drop(T2, D, P2)
```

---

**Diagram:**

A truck is located at T1, and an archive is located at T2. The task is to pick up the archive from T1 and drop it at T2, and then to drive from T1 to T2 and finally to D.
**HTN Progression Search**

**Standard Progression Algorithm**

1. $\text{fringe} \leftarrow \{(s, t_n, ())\}$
2. **while** $\text{fringe} \neq \emptyset$ **do**
   3. $n \leftarrow \text{fringe.poll()}$
   4. **if** $n$.isgoal **then** **return** $n$
   5. $U \leftarrow n$.unconstrainedNodes
   6. **for** $t \in U$ **do**
   7. **if** isPrimitive($t$) **then**
      8. $n' \leftarrow n$.apply($t$)
      9. fringe.add($n'$)
   10. **else**
      11. **for** $m \in t$.methods **do**
      12. $n' \leftarrow n$.decompose($t, m$)
      13. fringe.add($n'$)

$$\pi = (\text{no-op()}, \text{no-op()}, \text{pick-up}(T_1, A, P_1))$$
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

```
fringe ← \{(s_i, tn_i, ())\}
while fringe ≠ ∅ do
  n ← fringe.poll()
  if n.isgoal then return n
  U ← n.unconstrainedNodes
  for t ∈ U do
    if isPrimitive(t) then
      n' ← n.apply(t)
      fringe.add(n')
    else
      for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')

π = (no-op(), no-op(), pick-up(T_1, A, P_1))
```
1 fringe ← \{(s, t, n, ())\}
2 while fringe ≠ ∅ do
3 n ← fringe.poll()
4 if n.isgoal then return n
5 U ← n.unconstrainedNodes
6 for t ∈ U do
7     if isPrimitive(t) then
8         n' ← n.apply(t)
9         fringe.add(n')
10    else
11       for m ∈ t.methods do
12           n' ← n.decompose(t, m)
13           fringe.add(n')

\[ \pi = (\text{no-op()}, \text{no-op()}, \text{pick-up}(T_1, A, P_1), \text{drive}(T_1, A, B)) \]
Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning

June 25th, ICAPS 2018 (Delft)

14 / 62

Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

\[ \pi = (\text{no-op}(), \text{no-op}(), \text{pick-up}(T_1, A, P_1), \text{drive}(T_1, A, B), \text{pick-up}(T_2, C, P_2)) \]
HTN Progression Search

Standard Progression Algorithm

\begin{algorithm}
\begin{algorithmic}
\State \texttt{fringe} $\leftarrow \{(s_1, t_{n_1}, ())\}$
\While{\texttt{fringe} $\neq \emptyset$}
\State \texttt{n} $\leftarrow$ \texttt{fringe.poll()}\;
\If{\texttt{n.isgoal}} \textbf{return} \texttt{n}\; \EndIf
\State \texttt{U} $\leftarrow$ \texttt{n.unconstrainedNodes}$\texttt{U}$
\For{\texttt{t} $\in$ \texttt{U}}
\If{\texttt{isPrimitive($t$)}}
\State \texttt{n'} $\leftarrow$ \texttt{n.apply($t$)}
\State \texttt{fringe.add($n'$)}
\Else
\For{\texttt{m} $\in$ \texttt{t.methods}}
\State \texttt{n'} $\leftarrow$ \texttt{n.decompose($t$, m)}
\State \texttt{fringe.add($n'$)}
\EndFor
\EndIf
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

$\pi = (\text{no-op()}, \text{no-op()}, \text{pick-up}(T_1, A, P_1), \text{drive}(T_1, A, B), \text{pick-up}(T_2, C, P_2))$
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

\[ \pi = (\text{no-op}(), \text{no-op}(), \text{pick-up}(T_1, A, P_1), \text{drive}(T_1, A, B), \text{pick-up}(T_2, C, P_2)) \]
**Standard Progression Algorithm**

1. $\text{fringe} \leftarrow \{(s_i, t_n, ())\}$
2. while $\text{fringe} \neq \emptyset$ do
   3. $n \leftarrow \text{fringe.poll}(())$
   4. if $n$.isgoal then return $n$
   5. $U \leftarrow n$.unconstrainedNodes
   6. for $t \in U$ do
      7. if $\text{isPrimitive}(t)$ then
         8. $n' \leftarrow n$.apply($t$)
         9. $\text{fringe}.\text{add}(n')$
      10. else
           11. for $m \in t$.methods do
               12. $n' \leftarrow n$.decompose($t, m$)
               13. $\text{fringe}.\text{add}(n')$

\[\pi = (\text{no-op}(), \text{no-op}(), \text{pick-up}(T_1, A, P_1), \text{drive}(T_1, A, B), \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D))\]
Solving Techniques

HTN Progression Search

Standard Progression Algorithm

\[ \begin{align*}
1 & \text{ fringe } \leftarrow \{(s, t_n, ())\} \\
2 & \text{ while fringe } \neq \emptyset \text{ do} \\
3 & \quad n \leftarrow \text{fringe.poll}() \\
4 & \quad \text{if } n.\text{isgoal} \text{ then return } n \\
5 & \quad U \leftarrow n.\text{unconstrainedNodes} \\
6 & \quad \text{for } t \in U \text{ do} \\
7 & \quad \quad \text{if isPrimitive}(t) \text{ then} \\
8 & \quad \quad \quad n' \leftarrow n.\text{apply}(t) \\
9 & \quad \quad \quad \text{fringe.add}(n') \\
10 & \quad \quad \text{else} \\
11 & \quad \quad \quad \text{for } m \in t.\text{methods} \text{ do} \\
12 & \quad \quad \quad \quad n' \leftarrow n.\text{decompose}(t, m) \\
13 & \quad \quad \quad \quad \text{fringe.add}(n')
\end{align*} \]

\[ \pi = (\text{no-op}(), \text{no-op}(), \text{pick-up}(T_1, A, P_1), \text{drive}(T_1, A, B), \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D), \text{drop}(T_2, D, P_2)) \]
1. fringe $\leftarrow \{(s, tn, (\))\}$
2. while fringe $\neq \emptyset$ do
3.     $n \leftarrow$ fringe.poll()
4.     if $n$.isgoal then return $n$
5.     $U \leftarrow n$.unconstrainedNodes
6.     for $t \in U$ do
7.         if isPrimitive($t$) then
8.             $n' \leftarrow n$.apply($t$)
9.             fringe.add($n'$)
10.    else
11.        for $m \in t$.methods do
12.            $n' \leftarrow n$.decompose($t$, $m$)
13.            fringe.add($n'$)

$\pi = (\text{no-op(), no-op(), pick-up}(T_1, A, P_1), \text{drive}(T_1, A, B), \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D), \text{drop}(T_2, D, P_2), \text{drop}(T_1, B, P_1))$
Progression Search – Properties

- Progression Search is **sound** . . .
- . . . and **complete**
Progression Search – Properties

- Progression Search is **sound** . . .
- . . . and **complete**
- It maintains the **current state** during search
- This has been used to control search via state-based **preconditions** for **methods**
Progression Search – Properties

- Progression Search is **sound** . . .
- . . . and **complete**
- It maintains the **current state** during search
- This has been used to control search via state-based **preconditions** for **methods**
- It is also useful for calculating **heuristics**
Observation: In partially ordered models, standard progression search searches parts of the search space more than once.
Observation: In partially ordered models, standard progression search searches parts of the search space more than once.

This is due to branching (i.e. a non-deterministic choice) over unconstrained compound tasks.

When processing actions, the algorithm commits to an ordering in the solution.

The decision which task is decomposed implies no commitment to the solution.

The decision which method is used implies commitment to the solution.
Observation: In partially ordered models, standard progression search searches parts of the search space more than once.

This is due to branching (i.e. a non-deterministic choice) over unconstrained \textit{compound} tasks.

When processing actions, the algorithm \textit{commits} to an ordering in the solution.

The decision which \textit{task is decomposed} implies \textit{no} commitment to the solution.

The decision which \textit{method is used} implies commitment to the solution.

$\rightarrow$ For selection of the compound task, no branching is needed, we can simply “pick” one and decompose it.

$\rightarrow$ The decision which method is used must be made via branching.
Improved Progression Algorithm

1. $\text{fringe} \leftarrow \{(s_0, t_{n_1}, ())\}$
2. while $\text{fringe} \neq \emptyset$ do
   3. $n \leftarrow \text{fringe.poll}()$
   4. if $n$ is goal then return $n$
   5. $(U_C, U_P) \leftarrow n.unconstrainedNodes$
   6. for $t \in U_P$ do
      7. $n' \leftarrow n.apply(t)$
      8. fringe.add($n'$)
   9. $t \leftarrow \text{selectAbstractTask}(U_C)$
   10. for $m \in t$ methods do
      11. $n' \leftarrow n.decompose(t, m)$
      12. fringe.add($n'$)

Unconstrained tasks are split into compound and primitive tasks.
Improved Progression Algorithm

1. \( fringe \leftarrow \{(s_0, t_{n_1}, ())\} \)
2. \textbf{while} \( fringe \neq \emptyset \) \textbf{do}
3. \( n \leftarrow fringe.\text{poll}() \)
4. \textbf{if} \( n.\text{isgoal} \) \textbf{then} \textbf{return} \( n \)
5. \( (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \)
6. \textbf{for} \( t \in U_P \) \textbf{do}
7. \( n' \leftarrow n.\text{apply}(t) \)
8. \( fringe.\text{add}(n') \)
9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
10. \textbf{for} \( m \in t.\text{methods} \) \textbf{do}
11. \( n' \leftarrow n.\text{decompose}(t, m) \)
12. \( fringe.\text{add}(n') \)

- **Unconstrained tasks** are split into compound and primitive tasks.
- **Action application** is done via branching.
Improved Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s_0, t_0, ())\} \)
2. while \( \text{fringe} \neq \emptyset \) do
   3. \( n \leftarrow \text{fringe.poll()} \)
   4. if \( n \text{.isgoal} \) then return \( n \)
   5. \((U_C, U_P) \leftarrow n \text{.unconstrainedNodes} \)
   6. for \( t \in U_P \) do
      7. \( n' \leftarrow n \text{.apply}(t) \)
      8. \( \text{fringe.add}(n') \)
   9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
  10. for \( m \in t \text{.methods} \) do
     11. \( n' \leftarrow n \text{.decompose}(t, m) \)
     12. \( \text{fringe.add}(n') \)

- **Unconstrained tasks** are split into compound and primitive tasks
- **Action application** is done via branching
- Only **one compound task** is processed
Improved Progression Algorithm

1. \( fringe \leftarrow \{(s_0, tn_I, ())\}\)
2. \textbf{while} \( fringe \neq \emptyset \) \textbf{do}
3. \( n \leftarrow fringe\text{.poll}()\)
4. \textbf{if} \( n\text{.isgoal} \) \textbf{then} \textbf{return} \( n \)
5. \( (U_C, U_P) \leftarrow n\text{.unconstrainedNodes} \)
6. \textbf{for} \( t \in U_P \) \textbf{do}
7. \( n' \leftarrow n\text{.apply}(t) \)
8. \textbf{if} \( n' \) \textbf{then} \( fringe\text{.add}(n') \)
9. \( t \leftarrow selectAbstractTask(U_C) \)
10. \textbf{for} \( m \in t\text{.methods} \) \textbf{do}
11. \( n' \leftarrow n\text{.decompose}(t, m) \)
12. \textbf{if} \( n' \) \textbf{then} \( fringe\text{.add}(n') \)

- **Unconstrained tasks** are split into compound and primitive tasks
- **Action application** is done via branching
- Only **one compound task** is processed
- **Method application** is done via branching
Improved Progression Algorithm

1. \( fringe \leftarrow \{(s_0, tn_1, (\))\} \)
2. \( \text{while } fringe \neq \emptyset \text{ do} \)
3. \( n \leftarrow fringe.\text{poll}() \)
4. \( \text{if } n.\text{isgoal} \text{ then return } n \)
5. \( (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \)
6. \( \text{for } t \in U_P \text{ do} \)
7. \( n' \leftarrow n.\text{apply}(t) \)
8. \( \text{fringe.add}(n') \)
9. \( t \leftarrow selectAbstractTask(U_C) \)
10. \( \text{for } m \in t.\text{methods} \text{ do} \)
11. \( n' \leftarrow n.\text{decompose}(t, m) \)
12. \( \text{fringe.add}(n') \)

\[ \pi = () \]
**HTN Progression Search**

**Improved Progression Algorithm**

```
1 fringe ← {((s₀, tn₁, )),}  
2 while fringe ≠ ∅ do  
3     n ← fringe.poll()  
4     if n.isgoal then return n  
5     (UC, UP) ← n.unconstrainedNodes  
6     for t ∈ UP do  
7         n′ ← n.apply(t)  
8         fringe.add(n′)  
9     t ← selectAbstractTask(UC)  
10    for m ∈ t.methods do  
11        n′ ← n.decompose(t, m)  
12        fringe.add(n′)  
π = ()
```
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Improved Progression Algorithm

\[
\text{fringe} \leftarrow \{(s_0, t_1, ())\}
\]

\[
\text{while fringe} \neq \emptyset \text{ do}
\]

\[
\pi = ()
\]

\[
\text{n} \leftarrow \text{fringe.poll()}
\]

\[
\text{if n.isgoal then return n}
\]

\[
(U_C, U_P) \leftarrow n.\text{unconstrainedNodes}
\]

\[
\text{for } t \in U_P \text{ do}
\]

\[
\text{n'} \leftarrow n.\text{apply}(t)
\]

\[
\text{fringe.add(n')}
\]

\[
t \leftarrow \text{selectAbstractTask}(U_C)
\]

\[
\text{for } m \in t.\text{methods do}
\]

\[
\text{n'} \leftarrow n.\text{decompose}(t, m)
\]

\[
\text{fringe.add(n')}
\]

\[
\text{get-to}(T_1, A) \leadsto \text{pick-up}(T_1, A, P_1) \leadsto \text{get-to}(T_1, B) \leadsto \text{drop}(T_1, B, P_1)
\]

\[
\text{deliver}(P_2, D)
\]
Improved Progression Algorithm

```
1  fringe ← {{s₀, tn₁, ()}}
2  while fringe ≠ ∅ do
3      n ← fringe.poll()
4      if n.isgoal then return n
5      (U₀, U₁) ← n.unconstrainedNodes
6      for t ∈ U₁ do
7          n' ← n.apply(t)
8          fringe.add(n')
9      t ← selectAbstractTask(U₀)
10     for m ∈ t.methods do
11        n' ← n.decompose(t, m)
12        fringe.add(n')
```

$$\pi = ()$$
Solving Techniques

HTN Progression Search

Improved Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s_0, t_{n_1}, ())\} \)
2. while \( \text{fringe} \neq \emptyset \) do
3. \( n \leftarrow \text{fringe.poll()} \)
4. if \( n.\text{isgoal} \) then return \( n \)
5. \( (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \)
6. for \( t \in U_P \) do
7. \( n' \leftarrow n.\text{apply}(t) \)
8. \( \text{fringe.add}(n') \)
9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
10. for \( m \in t.\text{methods} \) do
11. \( n' \leftarrow n.\text{decompose}(t, m) \)
12. \( \text{fringe.add}(n') \)

\[ \pi = () \]

\( \text{pick-up}(T_1, A, P_1) \ldots \text{get-to}(T_1, B) \ldots \text{drop}(T_1, B, P_1) \)

\( \text{deliver}(P_2, D) \)

\( \text{no-op}() \)
HTN Progression Search

**Improved Progression Algorithm**

1. $\text{fringe} \leftarrow \{(s_0, \text{tn}_1, ())\}$
2. while $\text{fringe} \neq \emptyset$ do
   3. $n \leftarrow \text{fringe.poll}()$
   4. if $n\text{.isgoal}$ then return $n$
   5. $(U_C, U_P) \leftarrow n\text{.unconstrainedNodes}$
   6. for $t \in U_P$ do
      7. $n' \leftarrow n\text{.apply}(t)$
      8. fringe.add($n'$)
   9. $t \leftarrow \text{selectAbstractTask}(U_C)$
   10. for $m \in t\text{.methods}$ do
        11. $n' \leftarrow n\text{.decompose}(t, m)$
        12. fringe.add($n'$)

$\pi = (\text{no-op}())$

---

**Diagram**

```
pick-up(T_1, A, P_1) \prec \ldots \ldots \text{get-to}(T_1, B) \prec \ldots \ldots \text{drop}(T_1, B, P_1)
```

```
deliver(P_2, D)
```
1 \( fringe \leftarrow \{ (s_0, t_{n_1}, ()) \} \)
2 while \( fringe \neq \emptyset \) do 
3 \( n \leftarrow fringe.poll() \)
4 if \( n.isgoal \) then return \( n \)
5 \( (U_C, U_P) \leftarrow n.unconstrainedNodes \)
6 for \( t \in U_P \) do 
7 \( n' \leftarrow n.apply(t) \)
8 \( fringe.add(n') \)
9 \( t \leftarrow selectAbstractTask(U_C) \)
10 for \( m \in t.methods \) do 
11 \( n' \leftarrow n.decompose(t, m) \)
12 \( fringe.add(n') \)

\[ \pi = () \]
Improved Progression Algorithm

```
1 fringe ← \{(s₀, tn, ( ))\}
2 while fringe ≠ ∅ do
3     n ← fringe.poll()
4     if n.isgoal then return n
5     (U_C, U_P) ← n.unconstrainedNodes
6     for t ∈ U_P do
7         n' ← n.apply(t)
8         fringe.add(n')
9     t ← selectAbstractTask(U_C)
10    for m ∈ t.methods do
11       n' ← n.decompose(t, m)
12       fringe.add(n')

π = ( )
```

---

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

June 25th, ICAPS 2018 (Delft)
Improved Progression Algorithm

\[
\begin{aligned}
\text{fringe} & \leftarrow \{(s_0, t_1, ())\} \\
\text{while fring}e \neq \emptyset \text{ do} \\
\quad n & \leftarrow \text{fringe.poll()} \\
\quad \text{if } n.\text{isgoal} \text{ then return } n \\
\quad (U_C, U_P) & \leftarrow n.\text{unconstrainedNodes} \\
\text{for } t & \in U_P \text{ do} \\
\quad n' & \leftarrow n.\text{apply}(t) \\
\quad \text{fringe.add}(n') \\
\quad t & \leftarrow \text{selectAbstractTask}(U_C) \\
\text{for } m & \in t.\text{methods} \text{ do} \\
\quad n' & \leftarrow n.\text{decompose}(t, m) \\
\quad \text{fringe.add}(n') \\
\end{aligned}
\]

\[
\pi = (\text{no-op}())
\]
Improved Progression Algorithm

```
fringe ← {(s₀, tn₁, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (UC, UP) ← n.unconstrainedNodes
    for t ∈ UP do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(UC)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```

Here is a flowchart illustrating the algorithm:

- **Pick-up**: `pick-up(T₁, A, P₁)`
- **Get-to**: `get-to(T₁, B)`
- **Drop**: `drop(T₁, B, P₁)`
- **No-op**: `no-op()`

The progression search algorithm iteratively applies tasks and sub-tasks to achieve the goal state.
1. \( \text{fringe} \leftarrow \{(s_0, t_{n_1}, ())\} \)
2. \( \text{while } \text{fringe} \neq \emptyset \) do
3. \( n \leftarrow \text{fringe.poll()} \)
4. \( \text{if } n.\text{isgoal} \) then \( \text{return } n \)
5. \( (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \)
6. \( \text{for } t \in U_P \) do
7. \( n' \leftarrow n.\text{apply}(t) \)
8. \( \text{fringe.add}(n') \)
9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
10. \( \text{for } m \in t.\text{methods} \) do
11. \( n' \leftarrow n.\text{decompose}(t, m) \)
12. \( \text{fringe.add}(n') \)

\( \pi = () \)
Improved Progression Algorithm

1. fringe ← {(s₀, tn₁, ())}
2. while fringe ≠ ∅ do
   3. n ← fringe.poll()
   4. if n.isgoal then return n
   5. (U_C, U_P) ← n.unconstrainedNodes
   6. for t ∈ U_P do
      7. n' ← n.apply(t)
      8. fringe.add(n')
   9. t ← selectAbstractTask(U_C)
 10. for m ∈ t.methods do
     11. n' ← n.decompose(t, m)
     12. fringe.add(n')

π = (no-op())
1 \textit{fringe} \leftarrow \{(s_0, t_{n_1}, ())\}

2 \textbf{while} \textit{fringe} \neq \emptyset \textbf{do}

3 \quad n \leftarrow \textit{fringe.poll}()

4 \quad \textbf{if} \ n.\textit{isgoal} \ \textbf{then} \ \textbf{return} \ n

5 \quad (U_C, U_P) \leftarrow n.\textit{unconstrainedNodes}

6 \quad \textbf{for} \ t \in U_P \ \textbf{do}

7 \quad \quad n' \leftarrow n.\textit{apply}(t)

8 \quad \quad \textit{fringe.add}(n')

9 \quad t \leftarrow \textit{selectAbstractTask}(U_C)

10 \quad \textbf{for} \ m \in t.\textit{methods} \ \textbf{do}

11 \quad \quad n' \leftarrow n.\textit{decompose}(t, m)

12 \quad \quad \textit{fringe.add}(n')

\[ \pi = (\textit{no-op}()) \]
Improved Progression Algorithm

1. fringe ← \( \{(s_0, tn_1, ())\}\)
2. while fringe ≠ ∅ do
3. \( n \leftarrow \text{fringe.poll()} \)
4. if n.isgoal then return n
5. \((U_C, U_P) \leftarrow n\text{.unconstrainedNodes}\)
6. for \( t \in U_P \) do
7. \( n' \leftarrow n\text{.apply}(t) \)
8. \( \text{fringe.add}(n') \)
9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
10. for \( m \in t\text{.methods} \) do
11. \( n' \leftarrow n\text{.decompose}(t, m) \)
12. \( \text{fringe.add}(n') \)

\[ \pi = (\text{no-op}(), \text{no-op}()) \]
Improved Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s_0, t_1, ())\} \)
2. \( \text{while} \ \text{fringe} \neq \emptyset \ \text{do} \)
3. \( n \leftarrow \text{fringe.poll()} \)
4. \( \text{if} \ n.\text{isgoal} \ \text{then} \ \text{return} \ n \)
5. \( (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \)
6. \( \text{for} \ t \in U_P \ \text{do} \)
7. \( n' \leftarrow n.\text{apply}(t) \)
8. \( \text{fringe.add}(n') \)
9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
10. \( \text{for} \ m \in t.\text{methods} \ \text{do} \)
11. \( n' \leftarrow n.\text{decompose}(t, m) \)
12. \( \text{fringe.add}(n') \)

\( \pi = (\text{no-op()}, \text{pick-up}(T_2, C, P_2)) \)
Improved Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s_0, t_{n_1}, (\))\} \)
2. \( \textbf{while fringe} \neq \emptyset \) \( \textbf{do} \)
3. \( n \leftarrow \text{fringe.poll()} \)
4. \( \textbf{if} \ n \text{.isgoal} \textbf{then return} \ n \)
5. \( (U_C, U_P) \leftarrow n\text{.unconstrainedNodes} \)
6. \( \textbf{for} \ t \in U_P \textbf{ do} \)
7. \( n' \leftarrow n\text{.apply}(t) \)
8. \( \text{fringe}.\text{add}(n') \)
9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
10. \( \textbf{for} \ m \in t\text{.methods} \textbf{ do} \)
11. \( n' \leftarrow n\text{.decompose}(t, m) \)
12. \( \text{fringe}.\text{add}(n') \)

\[ \pi = (\text{no-op()}, \text{pick-up}(T_2, C, P_2)) \]
1 \textit{fringe} \leftarrow \{(s_0, t_0, (\))\}
2 \textbf{while} \textit{fringe} \neq \emptyset \textbf{ do}
3 \hspace{1em} n \leftarrow \textit{fringe.poll}()
4 \hspace{1em} \textbf{if} \ n.\textit{isgoal} \textbf{ then return} n
5 \hspace{1em} \hspace{1em} (U_C, U_P) \leftarrow n.\textit{unconstrainedNodes}
6 \hspace{1em} \textbf{for} \ t \in U_P \textbf{ do}
7 \hspace{2em} n' \leftarrow n.\textit{apply}(t)
8 \hspace{2em} \textit{fringe.add}(n')
9 \hspace{1em} \hspace{1em} t \leftarrow \textit{selectAbstractTask}(U_C)
10 \hspace{1em} \textbf{for} \ m \in t.\textit{methods} \textbf{ do}
11 \hspace{2em} n' \leftarrow n.\textit{decompose}(t, m)
12 \hspace{2em} \textit{fringe.add}(n')

\[ \pi = (\textit{no-op()}, \textit{pick-up}(T_2, C, P_2)) \]
1. fringe ← \{(s₀, tn₁, ())\}
2. while fringe ≠ ∅ do
   3. n ← fringe.poll()
   4. if n.isgoal then return n
   5. (U_C, U_P) ← n.unconstrainedNodes
   6. for t ∈ U_P do
      7. n' ← n.apply(t)
      8. fringe.add(n')
   9. t ← selectAbstractTask(U_C)
   10. for m ∈ t.methods do
       11. n' ← n.decompose(t, m)
       12. fringe.add(n')

\[ \pi = (\text{no-op()}, \text{pick-up}(T₂, C, P₂), \text{drive}(T₂, C, D)) \]
HTN Progression Search

Improved Progression Algorithm

\[
\text{ fringe } \leftarrow \{(s_0, t_{n_1}, ()\}\}
\]

\[\text{ while fringe } \neq \emptyset \text{ do}\]

\[n \leftarrow \text{fringe.poll()}\]

\[\text{ if } n.\text{isgoal} \text{ then return } n\]

\[(U_C, U_P) \leftarrow n.\text{unconstrainedNodes}\]

\[\text{ for } t \in U_P \text{ do}\]

\[n' \leftarrow n.\text{apply}(t)\]

\[\text{ fringe.add}(n')\]

\[t \leftarrow \text{selectAbstractTask}(U_C)\]

\[\text{ for } m \in t.\text{methods} \text{ do}\]

\[n' \leftarrow n.\text{compose}(t, m)\]

\[\text{ fringe.add}(n')\]

\[\pi = (\text{no-op()}, \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D), \text{no-op}())\]

\[\text{pick-up}(T_1, A, P_1) \rightarrow \ldots \text{get-to}(T_1, B) \rightarrow \ldots \text{drop}(T_1, B, P_1)\]

\[\text{drop}(T_2, D, P_2)\]
**Improved Progression Algorithm**

1. \( \text{fringe} \leftarrow \{(s_0, t_{n_1}, ())\} \)
2. \( \textbf{while} \ \text{fringe} \neq \emptyset \ \textbf{do} \)
   3. \( n \leftarrow \text{fringe.poll()} \)
   4. \( \textbf{if} \ n.\text{isgoal} \ \textbf{then} \ \textbf{return} \ n \)
   5. \( (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \)
   6. \( \textbf{for} \ t \in U_P \ \textbf{do} \)
      7. \( n' \leftarrow n.\text{apply}(t) \)
      8. \( \text{fringe}.\text{add}(n') \)
   9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
  10. \( \textbf{for} \ m \in t.\text{methods} \ \textbf{do} \)
     11. \( n' \leftarrow n.\text{decompose}(t, m) \)
     12. \( \text{fringe}.\text{add}(n') \)

\[ \pi = (\text{no-op()}, \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D), \text{no-op()}, \text{pick-up}(T_1, A, P_1)) \]

\[ \text{get-to}(T_1, B) \leftarrow \text{drop}(T_1, B, P_1) \]

\[ \text{drop}(T_2, D, P_2) \]
Improved Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s_0, t_i, (\))\} \)
2. \( \text{while } \text{fringe} \neq \emptyset \text{ do} \)
3. \( n \leftarrow \text{fringe.poll()} \)
4. \( \text{if } n \text{.isgoal } \text{then return } n \)
5. \( (U_C, U_P) \leftarrow n \text{.unconstrainedNodes} \)
6. \( \text{for } t \in U_P \text{ do} \)
7. \( n' \leftarrow n \text{.apply}(t) \)
8. \( \text{fringe.add}(n') \)
9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
10. \( \text{for } m \in t \text{.methods do} \)
11. \( n' \leftarrow n \text{.decompose}(t, m) \)
12. \( \text{fringe.add}(n') \)

\[ \pi = (\text{no-op()}, \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D), \text{no-op()}, \text{pick-up}(T_1, A, P_1), \text{drop}(T_2, D, P_2)) \]

\[ \text{get-to}(T_1, B) \leftarrow \text{drop}(T_1, B, P_1) \]
Improved Progression Algorithm

1. $\text{fringe} \leftarrow \{(s_0, tn_1, (\))\}$
2. while $\text{fringe} \neq \emptyset$ do
3.   $n \leftarrow \text{fringe.poll}()$
4.   if $n.\text{isgoal}$ then return $n$
5.   $(U_C, U_P) \leftarrow n.\text{unconstrainedNodes}$
6.   for $t \in U_P$ do
7.     $n' \leftarrow n.\text{apply}(t)$
8.     $\text{fringe.add}(n')$
9.   $t \leftarrow \text{selectAbstractTask}(U_C)$
10.  for $m \in t.\text{methods}$ do
11.     $n' \leftarrow n.\text{decompose}(t, m)$
12.     $\text{fringe.add}(n')$

$\pi = (\text{no-op()}, \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D), \text{no-op}(), \text{pick-up}(T_1, A, P_1), \text{drop}(T_2, D, P_2))$
Improved Progression Algorithm

fringe ← \{(s_0, t_n, (\))\}

while fringe ≠ ∅ do

n ← fringe.poll()

if n.isgoal then return n

(U_C, U_P) ← n.unconstrainedNodes

for t ∈ U_P do

n' ← n.apply(t)

fringe.add(n')

t ← selectAbstractTask(U_C)

for m ∈ t.methods do

n' ← n.decompose(t, m)

fringe.add(n')

\[ \pi = (\text{no-op}(), \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D), \text{no-op}(), \text{pick-up}(T_1, A, P_1), \text{drop}(T_2, D, P_2)) \]
Improved Progression Algorithm

1. $\text{fringe} \leftarrow \{(s_0, tn_i, ())\}$
2. while $\text{fringe} \neq \emptyset$ do
   3. $n \leftarrow \text{fringe.poll()}$
   4. if $n.\text{isgoal}$ then return $n$
   5. $(U_C, U_P) \leftarrow n.\text{unconstrainedNodes}$
   6. for $t \in U_P$ do
      7. $n' \leftarrow n.\text{apply}(t)$
      8. $\text{fringe.add}(n')$
   9. $t \leftarrow \text{selectAbstractTask}(U_C)$
   10. for $m \in t.\text{methods}$ do
       11. $n' \leftarrow n.\text{decompose}(t, m)$
       12. $\text{fringe.add}(n')$

$$\pi = (no-op(), \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, C, D), no-op(), \text{pick-up}(T_1, A, P_1), \text{drop}(T_2, D, P_2), \text{drive}(T_1, A, B))$$

$\text{drop}(T_1, B, P_1)$
fringe ← \{(s_0, tn_I, (\))\}

while fringe ≠ ∅ do
  n ← fringe.poll()
  if n.isgoal then return n
  (U_C, U_P) ← n.unconstrainedNodes
  for t ∈ U_P do
    n' ← n.apply(t)
    fringe.add(n')
  t ← selectAbstractTask(U_C)
  for m ∈ t.methods do
    n' ← n.decompose(t, m)
    fringe.add(n')

\( \pi = (no-op(), pick-up(T_2, C, P_2), drive(T_2, C, D), no-op(),
  pick-up(T_1, A, P_1), drop(T_2, D, P_2), drive(T_1, A, B), drop(T_1, B, P_1)) \)
Improved version of progression search is still **sound** and **complete**
Improved version of progression search is still **sound** and **complete**.

Searching the search space more than once is **avoided** (to a certain extend) but still **possible**.
- Improved version of progression search is still **sound** and **complete**

- Searching the search space more than once is **avoided** (to a certain extent) but still **possible**

- It may **increase the progression bound** necessary to solve the problem (problematic for some planning systems)
Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search

- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms
The basic idea is quite simple:

- **Translate** the input (HTN) problem into a classical planning problem
- Use a **classical planning system** to solve it
- Compile classical solution back to one for the HTN problem
The basic idea is quite simple:

- **Translate** the input (HTN) problem into a classical planning problem
- Use a **classical planning system** to solve it
- Compile classical solution back to one for the HTN problem

**Approach:**

- Add a **new part to the state** that represents the current task network
The basic idea is quite simple:

- **Translate** the input (HTN) problem into a classical planning problem
- Use a **classical planning system** to solve it
- Compile classical solution back to one for the HTN problem

**Approach:**

- Add a **new part to the state** that represents the current task network
- **Simulate a progression search** on this part of the state
  - Adapt original actions with respect to applicability and to maintain the new state features
  - Add actions that simulate decomposition methods
Compilation to STRIPS/ADL

HTN to STRIPS/ADL – Example

\[
pick-up(T_1, A, P_1) \leadsto get-to(T_1, B) \leadsto drop(T_1, B, P_1) \\
\]

\[
deliver(P_2, D) \\
\]

- Introduce id variables: \( t_0, t_1, \ldots, t_b \)
- Introduce new predicate for every task
Compilation to STRIPS/ADL

HTN to STRIPS/ADL – Example

\[ \text{pick-up}(T_1, A, P_1) \leadsto \text{get-to}(T_1, B) \leadsto \text{drop}(T_1, B, P_1) \]

\[ \text{deliver}(P_2, D) \]

- **tasks:**
  - \( \text{ppick-up}(T_1, A, P_1, t_2) \),
  - \( \text{pget-to}(T_1, B, t_3) \),
  - \( \text{pdrop}(T_1, B, P_1, t_5) \),
  - \( \text{pdeliver}(P_2, D, t_4) \),

- **orderings:**
  - \( \text{before}(t_2, t_3) \), \( \text{before}(t_3, t_5) \)

- Introduce id variables: \( t_0, t_1, \ldots, t_b \)

- Introduce new predicate for every task, represent current \( tn \) in the state
In the HTN approach, we can use STRIPS/ADL to represent the planning problem. The key steps are:

1. **Introduce id variables:** $t_0, t_1, \ldots, t_b$

2. **Introduce new predicate for every task:** To represent the current $tn$ in the state.

3. **Modify existing actions:**

   - $pick-up(T_1, A, P_1, t_2)$
     - $pre$: Precs from domain,
       - $ppick-up(T_1, A, P_1, t_2)$
       - $\forall t_i \in \{t_0 \ldots t_b\}: \neg before(t_i, t_2)$

   - $drop(T_1, B, P_1)$
     - $eff$: Effects from domain,
       - $\neg ppick-up(T_1, A, P_1, t_2), \ free(t_2)$
       - $\forall t_i \in \{t_0 \ldots t_b\}: \neg before(t_2, t_i)$

   - $deliver(P_2, D)$
     - $pre$: Precs from domain,
       - $ppick-up(T_1, A, P_1, t_2)$
     - $eff$: Effects from domain,
       - $\neg pdeliver(P_2, D, t_4), \ before(t_1, t_6), \ before(t_3, t_7)$

   - $get-to(T_1, B, t_3)$
     - $pre$: Precs from domain,
       - $pget-to(T_1, B, t_3)$
     - $eff$: Effects from domain,
       - $\neg pget-to(T_1, B, t_3), \ before(t_3, t_5)$

   - $drop(T_1, B, P_1, t_5)$
     - $pre$: Precs from domain,
       - $pdrop(T_1, B, P_1, t_5)$
     - $eff$: Effects from domain,
       - $\neg pdrop(T_1, B, P_1, t_5), \ before(t_5, t_7)$

   - $deliver(P_2, D, t_4)$
     - $pre$: Precs from domain,
       - $pdeliver(P_2, D, t_4)$
     - $eff$: Effects from domain,
       - $\neg pdeliver(P_2, D, t_4), \ before(t_4, t_7)$

   - $before(t_2, t_3), before(t_3, t_5)$

By introducing these id variables and predicates, we can effectively represent the hierarchical structure of the HTN planning problem.
HTN to STRIPS/ADL – Example

- Introduce id variables: \(t_0, t_1, \ldots, t_b\)
- Introduce new predicate for every task, represent current \(tn\) in the state
- Modify existing actions
- Add new actions simulating methods

\[\text{pick-up}(T_1, A, P_1) \leadsto \ldots \text{get-to}(T_1, B) \leadsto \ldots \text{drop}(T_1, B, P_1)\]

\[\text{deliver}(P_2, D)\]

\[\text{p pick-up}(T_1, A, P_1, t_2),\]
\[\text{p get-to}(T_1, B, t_3),\]
\[\text{p drop}(T_1, B, P_1, t_5),\]
\[\text{p deliver}(P_2, D, t_4),\]
\[\text{before}(t_2, t_3), \text{before}(t_3, t_5)\]

\[\text{m get-to}(C, T_2) \text{p pick-up}(T_2, C, P_2) \text{get-to}(D, T_2), \text{drop}(D, T_2, P_2)\]

\[\text{m deliver}(P_2, C, D, T_2, t_4, t_1, t_6, t_7)\]

\[\text{p deliver}(P_2, D, t_4)\]

\[\text{p get-to}(C, T_2, t_1)\]
\[\text{p pick-up}(C, T_2, P_2, t_6)\]
\[\text{p get-to}(D, T_2, t_7)\]
\[\text{p drop}(D, T_2, P_2, t_4)\]
\[\text{before}(t_1, t_6), \ldots \neg \text{free}(t_1), \ldots\]
Benefits:

- Sophisticated planning system(s) available
- Large portfolio of heuristics available
Benefits:

- Sophisticated planning system(s) available
- Large portfolio of heuristics available

Challenges:

- How to represent the task network? (example was simplified)
  - To get a compact state
  - To get a small set of actions
  - To break symmetry
  - To preserve information when using available classical heuristics (e.g. delete-relaxation)
Compilation to STRIPS/ADL

Translating HTN Problems to STRIPS/ADL

Benefits:

- Sophisticated planning system(s) available
- Large portfolio of heuristics available

Challenges:

- How to represent the task network? (example was simplified)
  - To get a compact state
  - To get a small set of actions
  - To break symmetry
  - To preserve information when using available classical heuristics (e.g. delete-relaxation)
- How many ids are sufficient?
  - Only computable for subclasses of HTN planning problems
Benefits:

- Sophisticated planning system(s) available
- Large portfolio of heuristics available

Challenges:

- How to represent the task network? (example was simplified)
  - To get a compact state
  - To get a small set of actions
  - To break symmetry
  - To preserve information when using available classical heuristics (e.g. delete-relaxation)
- How many ids are sufficient?
  - Only computable for subclasses of HTN planning problems
  - Approach for general HTN planning problems:
    - Incrementally increase it like in SAT-based classical planning
    - But there is no upper bound, so only stop when a plan was found
Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search

- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms
Basic idea:

- Translate HTN planning problem to a **propositional formula**
- Solve it with a standard **SAT solver**
- Formula represents solution to the HTN
Compilation to SAT

HTN to SAT Compilations

Basic idea:

- Translate HTN planning problem to a **propositional formula**
- Solve it with a standard **SAT solver**
- Formula represents solution to the HTN

Similar to approach in classical planning:

- Encodings of **state transition** can be re-used
- Translation to **a series of increasing** problems (instead of a single one)
Basic idea:

- Translate HTN planning problem to a **propositional formula**
- Solve it with a standard **SAT solver**
- Formula represents solution to the HTN

Similar to approach in classical planning:

- Encodings of **state transition** can be re-used
- Translation to **a series of increasing** problems (instead of a single one)

Challenges:

- How to represent **decomposition**?
- What is the best way to **bound** the problem?
We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)
We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)

But: There are (double-exponentially) **many trees** for a single planning problem
We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)

But: There are (double-exponentially) **many trees** for a single planning problem

→ Compact representation of all possible decompositions of the initial task network: **Path Decomposition Trees** (PDTs)
We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)

But: There are (double-exponentially) **many trees** for a single planning problem

→ Compact representation of all possible decompositions of the initial task network: **Path Decomposition Trees** (PDTs)
We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)

But: There are (double-exponentially) **many trees** for a single planning problem

→ Compact representation of all possible decompositions of the initial task network: **Path Decomposition Trees** (PDTs)
We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)

But: There are (double-exponentially) many trees for a single planning problem

→ Compact representation of all possible decompositions of the initial task network: **Path Decomposition Trees** (PDTs)
All Decomposition Trees can not be represented
All Decomposition Trees can not be represented
⇒ bound the height of represented trees
All Decomposition Trees can not be represented
⇒ bound the height of represented trees
**All** Decomposition Trees can not be represented
⇒ bound the height of represented trees

```
c_I \rightarrow ABC \text{ and } c_I \rightarrow ACp \text{ and } c_I \rightarrow Ar
```
All Decomposition Trees can not be represented
⇒ bound the height of represented trees

\[ c_I \rightarrow ABC \text{ and } c_I \rightarrow ACp \text{ and } c_I \rightarrow Ar \]
\[
\{A\} \quad \{B, C\} \quad \{C, p, r\}
\]
All Decomposition Trees cannot be represented
⇒ bound the height of represented trees

\[ c_l \rightarrow ABC \quad \text{and} \quad c_l \rightarrow APC \quad \text{and} \quad c_l \rightarrow Ar \]

\( \{ A \} \quad \{ B, C \} \quad \{ C, p, r \} \)
All Decomposition Trees can not be represented
⇒ bound the height of represented trees

\[ c_i \rightarrow ABC \text{ and } c_i \rightarrow ACp \text{ and } c_i \rightarrow Ar \]
\[ \{ A \} \quad \{ B, C \} \quad \{ C, p, r \} \]
**All** Decomposition Trees can not be represented
⇒ bound the height of represented trees

\[ c_1 \rightarrow ABC \text{ and } c_1 \rightarrow ACp \text{ and } c_1 \rightarrow Ar \]

\[
\begin{align*}
\{A\} & \quad \{B, C\} & \quad \{C, p, r\}
\end{align*}
\]
All Decomposition Trees can not be represented
⇒ bound the height of represented trees

\[ c_I \rightarrow ABC \text{ and } c_I \rightarrow ACp \text{ and } c_I \rightarrow Ar \]
\[ \{ A \} \quad \{ B, C \} \quad \{ C, p, r \} \]

Also possible: \[ \{ A \} \quad \{ A, B, C \} \quad \{ C, p, r \} \]
All Decomposition Trees can not be represented
⇒ bound the height of represented trees

\[
c_I \rightarrow ABC \text{ and } c_I \rightarrow ACp \text{ and } c_I \rightarrow Ar
\]

\[
\{A\} \quad \{B, C\} \quad \{C, p, r\}
\]

Also possible:
\[
\{A\} \quad \{A, B, C\} \quad \{C, p, r\}
\]

Also possible:
\[
\{A\} \quad \{B\} \quad \{C\} \quad \{p, r\}
\]
PDTs can be generated by locally deciding on how to assign sub-tasks to children
PDTs can be generated by locally deciding on how to assign sub-tasks to children.

Difficult question: How does an optimal PDT look like?
PDTs can be generated by locally deciding on how to assign sub-tasks to children.

Difficult question: How does an optimal PDT look like?
- Least amount of leaves?
PDTs can be generated by locally deciding on how to assign sub-tasks to children.

Difficult question: How does an optimal PDT look like?
- Least amount of leafs?
- Fewer tasks per leaf?
PDTs can be generated by locally deciding on how to assign sub-tasks to children

Difficult question: How does an optimal PDT look like?
- Least amount of leaves?
- Fewer tasks per leaf?
- Fewer tasks per inner node?
Generating PDTs

- PDTs can be generated by locally deciding on how to assign sub-tasks to children
- Difficult question: How does an optimal PDT look like?
  - Least amount of leafs?
  - Fewer tasks per leaf?
  - Fewer tasks per inner node?

⇒ Locally optimizing #children does not lead to global minimum!
PDTs *can* be generated by locally deciding on how to assign sub-tasks to children

Difficult question: How does an optimal PDT look like?
- Least amount of leaves?
- Fewer tasks per leaf?
- Fewer tasks per inner node?

⇒ Locally optimizing #children does not lead to global minimum!

Current work tries greedily to put as few tasks as possible to each child
What are PDTs good for?

- A PDT contains **every** Decomposition Tree of height \( \leq K \) as a sub-graph.
What are PDTs good for?

- A PDT contains every Decomposition Tree of height $\leq K$ as a sub-graph.
- Let the valuation of a SAT formula describe such a tree.
What are PDTs good for?

- A PDT contains **every** Decomposition Tree of height \( \leq K \) as a sub-graph
- Let the valuation of a SAT formula describe such a tree
- The formula then asserts that it is a valid Decomposition Tree
Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search

- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms
What do we want to estimate?

- Number of missing actions (or their costs, resp.) or
What do we want to estimate?

- Number of missing actions (or their costs, resp.) or
- Number of missing modifications, i.e.,
  - decompositions,
  - task insertions (if allowed),
  - causal link and ordering insertions (in plan space-based search), and
  - action applications (in progression-based search)
Possible Heuristic Estimates

What do we want to estimate?

- Number of missing actions (or their costs, resp.) or
- Number of missing modifications, i.e.,
  - decompositions,
  - task insertions (if allowed),
  - causal link and ordering insertions (in plan space-based search), and
  - action applications (in progression-based search)

→ To be used for the selection of a search node (task network/partial plan) out of the fringe
Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search

- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms
How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?
How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

- Perform task insertion
How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

- Perform task insertion
- Perform delete relaxation
How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

- Perform task insertion
- Perform delete relaxation

→ This makes the (TI)HTN plan existence problem decidable in $\mathbb{P}$
How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

- Perform task insertion
- Perform delete relaxation

→ This makes the (TI)HTN plan existence problem decidable in P

We introduce the Task Decomposition Graph (TDG) – which bases upon task insertion and delete relaxation – as a means to represent the task hierarchy.
A TDG represents the decomposition structure:

A TDG is a (possibly cyclic) bipartite graph $\mathcal{G} = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with
A TDG represents the decomposition structure:

A TDG is a (possibly cyclic) bipartite graph $\mathcal{G} = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

- $N_T$, the task nodes,
Decomposition Graph-based Heuristics

A TDG represents the decomposition structure:

A TDG is a (possibly cyclic) bipartite graph $\mathcal{G} = \langle N_T, N_M, E(T,M), E(M,T) \rangle$ with

- $N_T$, the task nodes,
- $N_M$, the method nodes,
A TDG represents the decomposition structure:

A TDG is a (possibly cyclic) bipartite graph $G = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

- $N_T$, the task nodes,
- $N_M$, the method nodes,
- $E_{(T,M)}$, the task edges,
A TDG represents the decomposition structure:

A TDG is a (possibly cyclic) bipartite graph $G = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with:

- $N_T$, the task nodes,
- $N_M$, the method nodes,
- $E_{(T,M)}$, the task edges,
- $E_{(M,T)}$, the method edges.
A TDG represents the decomposition structure:

How to use the TDG to calculate an heuristic estimate?

**Step 1:**
Calculate the TDG in a preprocessing step.

**Step 2:**
Calculate heuristic $h(t)$ for each task $t$ in TDG (still via preprocessing).

**Step 3:**
For a search node (partial plan) $P$ and its task identifiers $T$, calculate

$$h(P) := \sum_{t \in T} h(t).$$
Let $\langle N_T, N_M, E_{T\rightarrow M}, E_{M\rightarrow T} \rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} 
\text{cost}(n_t) & \text{if } n_t \text{ primitive} \\
\min_{(n_t, n_m) \in E_{T\rightarrow M}} h_M(n_m) & \text{else}
\end{cases}$$
Let $\langle N_T, N_M, E_{T \rightarrow M}, E_{M \rightarrow T} \rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} 
\text{cost}(n_t) & \text{if } n_t \text{ primitive} \\
\min_{(n_t, n_m) \in E_{T \rightarrow M}} h_M(n_m) & \text{else}
\end{cases}$$

For method nodes $n_m = \langle T, \prec, \alpha \rangle$:

$$h_M(n_m) := \sum_{(n_m, n_t) \in E_{M \rightarrow T}} h_T(n_t)$$
Decomposition Graph-based Heuristics

Cost-aware heuristic TDG-c

Let $\langle N_T, N_M, E_{T \rightarrow M}, E_{M \rightarrow T} \rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} 
\text{cost}(n_t) & \text{if } n_t \text{ primitive} \\
\min \limits_{(n_t, n_m) \in E_{T \rightarrow M}} h_M(n_m) & \text{else}
\end{cases}$$

For method nodes $n_m = \langle T, \prec, \alpha \rangle$

$$h_M(n_m) := \sum \limits_{(n_m, n_t) \in E_{M \rightarrow T}} h_T(n_t)$$

For a given partial plan $P = (T, \prec, \alpha, CL)$, i.e., a search node, its heuristic is $h(P) := \sum_{t \in T} h(t)$ to estimate the cost of the cheapest reachable plan.
Decomposition Graph-based Heuristics

Cost-aware heuristic TDG-c (Example)

A TDG represents the decomposition structure:

How to use the TDG to calculate an heuristic estimate?

**Step 1:**
Calculate the TDG in a preprocessing step.

**Step 2:**
Calculate heuristic $h(t)$ for each task $t$ in TDG (still via preprocessing).

**Step 3:**
For a search node (partial plan) $P$ and its task identifiers $T$, calculate $h(P) := \sum_{t \in T} h(t)$.
A TDG represents the decomposition structure:

Example:

\[ h_T(t_0) = \min \{ h_M(m_1), h_M(m_2) \} \]
Decomposition Graph-based Heuristics

Cost-aware heuristic TDG-c (Example)

A TDG represents the decomposition structure:

Example:

Method $m_1 = (t_0, tn)$ with task network $tn$:

$$h_M(m_1) = \sum_{t_i \in \{t_1, t_2, t_3\}} h_T(t_i)$$

$$= h_T(t_1) + \text{cost}(t_2) + h_T(t_3)$$
A TDG represents the decomposition structure:

Example:

\[ h_T(t_1) = \min \{ h_M(m_3), h_M(m_4) \} \]
A TDG represents the decomposition structure:

**Example:**

Method $m_4 = (t_1, tn)$ with task network $tn$:

$$h_M(m_4) = \sum_{t_i \in \{t_5, t_6\}} h_T(t_i)$$

$$= h_T(t_5) + h_T(t_6)$$

$$= \text{cost}(t_5) + \text{cost}(t_6)$$
Decomposition Graph-based Heuristics

Modification-aware heuristic TDG-m

Let \( \langle N_T, N_M, E_{T \rightarrow M}, E_{M \rightarrow T} \rangle \) be a TDG.

The estimates of the TDG are defined as follows:

\[
h_T(n_t) := \begin{cases} 
|pre(n_t)| & \text{if } n_t \text{ primitive} \\ 
1 + \min_{(n_t, n_m) \in E_{T \rightarrow M}} h_M(n_m) & \text{else} 
\end{cases}
\]
Let \( \langle N_T, N_M, E_{T\rightarrow M}, E_{M\rightarrow T} \rangle \) be a TDG.

The estimates of the TDG are defined as follows:

\[
h_T(n_t) := \begin{cases} 
|\text{pre}(n_t)| & \text{if } n_t \text{ primitive} \\
1 + \min_{(n_t,n_m) \in E_{T\rightarrow M}} h_M(n_m) & \text{else}
\end{cases}
\]

For method nodes \( n_m = \langle T, \prec, \alpha \rangle \):

\[
h_M(n_m) := \sum_{(n_m,n_t) \in E_{M\rightarrow T}} h_T(n_t)
\]
Let $\langle N_T, N_M, E_{T\rightarrow M}, E_{M\rightarrow T} \rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} |pre(n_t)| & \text{if } n_t \text{ primitive} \\ 1 + \min_{(n_t, n_m) \in E_{T\rightarrow M}} h_M(n_m) & \text{else} \end{cases}$$

For method nodes $n_m = \langle T, \prec, \alpha \rangle$:

$$h_M(n_m) := \sum_{(n_m, n_t) \in E_{M\rightarrow T}} h_T(n_t)$$

For a given partial plan $P = (T, \prec, \alpha, CL)$, i.e, a search node, its heuristic is $h(P) := \sum_{t \in T} h(t) - |CL|$ to estimate the least number of required modifications to turn $P$ into a plan.
TDG-c and TDG-m are admissible estimates of:

- The costs of still missing actions – or
- The number of still missing decompositions and causal link insertions (the latter is specific for plan space-based planners)
TDG-c and TDG-m are admissible estimates of:

- The costs of still missing actions – or
- The number of still missing decompositions and causal link insertions (the latter is specific for plan space-based planners)

Further properties:

- Both can be calculated in polynomial time (also for the general, undecidable case)
TDG-c and TDG-m are admissible estimates of:

- The costs of still missing actions – or
- The number of still missing decompositions and causal link insertions (the latter is specific for plan space-based planners)

Further properties:

- Both can be calculated in polynomial time (also for the general, undecidable case)
- Both rely on task insertion and delete relaxation (for the construction process of the TDG)
TDG-c and TDG-m are admissible estimates of:

- The costs of still missing actions – or
- The number of still missing decompositions and causal link insertions (the latter is specific for plan space-based planners)

Further properties:

- Both can be calculated in polynomial time (also for the general, undecidable case)
- Both rely on task insertion and delete relaxation (for the construction process of the TDG)
- Only tasks within the the TDG account for the heuristic estimate, so task insertion is not reflected within the estimates (→ room for improvement; but this guarantees admissibility)
Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search

- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms
We have seen two search-based approaches that can be instantiated as **heuristic search**.

- We need to **sort** the fringe (according to what?)
- In a first step, estimate **goal distance** (→ Satisficing Planning)
We have seen two search-based approaches that can be instantiated as **heuristic search**.

- We need to **sort** the fringe (according to what?)
- In a first step, estimate **goal distance** (→ Satisficing Planning)

Using Techniques from Classical Planning – Challenges:
- More expressive formalism → techniques not applicable directly
- Hierarchy has huge impact on valid solutions
We have seen two search-based approaches that can be instantiated as **heuristic search**

We need to **sort** the fringe (according to what?)

In a first step, estimate **goal distance** (→ Satisficing Planning)

Using Techniques from Classical Planning – Challenges:

- More expressive formalism → techniques not applicable directly
- Hierarchy has huge impact on valid solutions
  - Which actions are reachable?
We have seen two search-based approaches that can be instantiated as **heuristic search**

- We need to **sort** the fringe (according to what?)
- In a first step, estimate **goal distance** (→ Satisficing Planning)

Using Techniques from Classical Planning – Challenges:

- More expressive formalism → techniques not applicable directly
- Hierarchy has huge impact on valid solutions
  - Which actions are reachable?
  - What is the objective, the “goal”? → usually no state-based goal given
Approach:

1. Relax HTN to a classical planning problem
   - Search is done in an HTN planning system on the original model
   - This model is only used for heuristic calculation
Using Classical Heuristics to Guide HTN Search

Classical Heuristics in HTN Planning

Approach:

1. Relax HTN to a classical planning problem
   - Search is done in an HTN planning system on the original model
   - This model is only used for heuristic calculation

2. Apply classical heuristics to that problem
   - For some search node, the “heuristic model” is adapted
   - Goal distance is estimated
Approach:

1. **Relax HTN to a classical planning problem**
   - Search is done in an HTN planning system on the original model
   - This model is only used for heuristic calculation

2. **Apply classical heuristics to that problem**
   - For some search node, the “heuristic model” is adapted
   - Goal distance is estimated

3. **Use heuristic value in HTN planning**
   - The fringe of the HTN planning system is sorted according to the heuristic value
Using Classical Heuristics to Guide HTN Search

Simulating Composition

- Introduce new state features
- Modify actions
- Introduce new action for every method
- Goal is to reach current task network

```
A /truck
B /archive
C
D
```

```
deliver(P, D)
```

```
m-deliver(P, C, D, T)
```

```
get-to(T, C)
m-via(T, B, C)
```

```
pick-up(T, C, P)
```

```
get-to(T, D)
m-via(T, B, D)
```

```
drop(T, D, P)
```

```
get-to(T, B)
m-direct(T, A, B)
```

```
drive(T, B, C)
```

```
get-to(T, B)
m-direct(T, C, B)
```

```
drive(T, C, B)
```

```
drive(T, A, B)
```

```
drive(T, B, D)
```

```
drive(T, B, C)
```

```
A 🚗
```

```
B
```

```
C
```

```
D
```
Using Classical Heuristics to Guide HTN Search

Simulating Composition

- Introduce new state features
- Modify actions
- Introduce new action for every method

Goal is to reach current task network

A /truck
B /archive
C
D

\[\text{deliver} (P, D)\]
\[\text{m-deliver} (P, C, D, T)\]
\[\text{get-to} (T, C)\]
\[\text{m-via} (T, B, C)\]
\[\text{pick-up} (T, C, P)\]
\[\text{get-to} (T, D)\]
\[\text{m-via} (T, B, D)\]
\[\text{drop} (T, D, P)\]
\[\text{get-to} (T, B)\]
\[\text{m-direct} (T, A, B)\]

1. \[\text{drive} (T, B, C)\]
2. \[\text{drive} (T, B, D)\]
3. \[\text{drive} (T, A, B)\]
4. \[\text{drive} (T, C, B)\]
Simulating Composition

Using Classical Heuristics to Guide HTN Search

A /truck

B /archive

C

D

\[
\text{deliver}(P, D) \\
\text{m-deliver}(P, C, D, T) \\
\text{get}(T, C) \\
\text{m-via}(T, B, C) \\
\text{get-to}(T, B) \\
\text{m-direct}(T, A, B) \\
\text{drive}(T, A, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, B, C) \\
\text{drive}(T, B, D) \\
\text{get-to}(T, B) \\
\text{m-direct}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{get-to}(T, D) \\
\text{m-via}(T, B, D) \\
\text{drop}(T, D, P) \\
\text{pick-up}(T, C, P) \\
\text{get-to}(T, D) \\
\text{m-via}(P, C, D, T) \\
\text{get}(P, D) \\
\text{m-via}(T, B, C) \\
\text{get}(T, B) \\
\text{m-direct}(T, A, B) \\
\text{drive}(T, A, B) \\
\text{drive}(T, B, D) \\
\text{drive}(T, B, C) \\
\text{drive}(T, B, D) \\
\text{drive}(T, B, C) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, B, D) \\
\text{drive}(T, B, C) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, B, D) \\
\text{drive}(T, B, C) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, B, D) \\
\text{drive}(T, B, C) \\
\text{drive}(T, B, D) \\
\text{drive}(T, B, C) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B) \\
\text{drive}(T, C, B)
Using Classical Heuristics to Guide HTN Search

Simulating Composition

```
introduce new state features
modify actions
introduce new action for every method
goal is to reach current task network
```

```
deli 1 (P, D)
m-deliver (P, C, D, T)
get-to (T, D)
m-via (T, B, D)
drop (T, D, P)
```

```
get 2 (T, C)
m-via (T, B, C)
get-to (T, B)
m-direct (T, A, B)
drive (T, A, B)
```

```
pick-up 3 (C, P)
get-to (T, D)
m-direct (T, C, B)
drive (T, C, B)
```

```
drive (T, B, C)
get-to (T, B)
m-direct (T, A, B)
drive (T, B)
```

```
drive (T, A, B)
```
Using Classical Heuristics to Guide HTN Search

Simulating Composition

Introduce new state features
Modify actions
Introduce new action for every method
Goal is to reach current task network

1. deli 1 (P, D)
   m-deliver (P, C, D, T)

2. ge 2 (T, C)
   m-via (T, B, C)

3. pick-up 3 (C, P)

4. ge 4 (T, D)
   m-via (T, B, D)

5. drop 5 (D, P)

6. ge 6 (T, B)
   m-direct (T, A, B)

7. drive 7 (B, C)

8. ge 8 (T, B)
   m-direct (T, C, B)

9. drive 9 (B, D)

10. drive 10 A, B

11. drive 11 C, B
Using Classical Heuristics to Guide HTN Search

Simulating Composition

- **deliver(P, D)**
- **m-deliver(P, C, D, T)**
- **get-to(T, D)**
- **m-via(T, B, D)**
- **drop(T, D, P)**
- **get-to(T, C)**
- **m-via(T, B, C)**
- **pick-up(T, C, P)**
- **get-to(T, B)**
- **m-direct(T, A, B)**
- **drive(T, A, B)**
- **drive(T, B, C)**
- **get-to(T, B)**
- **m-direct(T, C, B)**
- **drive(T, C, B)**
- **drive(T, B, D)**
- **drive(T, A, B)**
Introduce new state features

- Introduce new state features
- Modify actions
- Introduce new action for every method
- Goal is to reach current task network
Introduce new state features

- Introduce new state features
- Modify actions
- Introduce new action for every method

1. Goal is to reach current task network

   A /truck
   B /archive
   C
   D

   deliver(P, D)
   m-deliver(P, C, D, T)

   get-to(T, C)
   m-via(T, B, C)

   pick-up(T, C, P)
   get-to(T, D)
   m-via(T, B, D)

   drop(T, D, P)

   get-to(T, B)
   m-direct(T, A, B)

   drive(T, B, C)

   drive(T, A, B)

   drive(T, C, B)

   drive(T, B, C)

   drive(T, B, D)

   drive(T, C, B)

   drive(T, B, C)

   drive(T, B, D)

   drive(T, C, B)

   drive(T, B, C)

   drive(T, B, D)

   drive(T, C, B)

   drive(T, B, C)

   drive(T, B, D)

   drive(T, C, B)

   drive(T, B, C)
Introduce new state features
- Modify actions

- Introduce new state features
- Modify actions

Using Classical Heuristics to Guide HTN Search

Simulating Composition

- Introduce new state features
- Modify actions
- Introduce new state features
- Modify actions
Introduce new state features
- Modify actions
- Introduce new action for every method

```
A /truck
B /archive
C
D
```

```
deliver(P, D)
m-deliver(P, C, D, T)
goto(T, C)
m-via(T, B, C)
pick-up(T, C, P)
goto(T, D)
m-via(T, B, D)
drop(T, D, P)
goto(T, B)
m-direct(T, A, B)
drive(T, B, C)
goto(T, B)
m-direct(T, C, B)
drive(T, B, D)
```
Using Classical Heuristics to Guide HTN Search

Simulating Composition

- Introduce new state features
- Modify actions
- Introduce new action for every method

Goal is to reach current task network

```
A /truck
B /archive
C
D
```

```
deliver(P, D)
m-deliver(P, C, D, T)
```

```
get-to(T, C)
m-via(T, B, C)
```

```
pick-up(T, C, P)
get-to(T, D)
m-via(T, B, D)
drop(T, D, P)
```

```
drive(T, B)
m-direct(T, A, B)
drive(T, B, C)
```

```
get-to(T, B)
m-direct(T, A, B)
drive(T, B, C)
```

```
get-to(T, B)
m-direct(T, C, B)
drive(T, B, D)
```

```
drive(T, C, B)
```

```
get-to(v, l2)
m-direct(v, l1, l2)
drive(v, l1, l2)
```

```
b-drive(v, l1, l2) am-direct(v, l1, l2) b-get-to(v, l2)
```

```
¬at(v, l1)
road(l1, l2)
¬at(v, l2)
b-drive(v, l1, l2)
```

```
¬at(v, l1)
at(v, l2)
¬at(v, l)
in(p, v)
b-pick-up(v, l, p)
```

```
¬at(v, l)
at(p, l)
in(p, v)
drop(v, l, p)
```

```
¬in(p, v)
b-drop(v, l, p)
```

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

June 25th, ICAPS 2018 (Delft)
Simulating Composition

- Introduce new state features
- Modify actions
- Introduce new action for every method

---

**Using Classical Heuristics to Guide HTN Search**

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

June 25th, ICAPS 2018 (Delft)
Introduce new state features
Modify actions
Introduce new action for every method

- Introduce new state features
- Modify actions
- Introduce new action for every method
Introduce new state features

Modify actions

Introduce new action for every method
Introduce new state features
- Modify actions
- Introduce new action for every method
- Goal is to reach current task network

```
get-to(T, C) m-via(T, B, C)
```

```
drive(T, A, B)
```

```
drive(T, B, C)
```

```
drive(T, B, D)
```

```
drop(T, D, P)
```

```
get-to(T, B) m-direct(T, A, B)
```

```
pick-up(T, C, P)
```

```
get-to(T, D) m-via(T, B, D)
```

```
deliver(P, D)
```

```
m-deliver(P, C, D, T)
```

```
get-to(T, C) m-via(T, B, C)
```

```
get-to(T, B) m-direct(T, C, B)
```

```
drive(T, C, B)
```

```
drive(T, B, D)
```

```
drive(T, B, D)
```

```
drive(T, A, B)
```

```
drive(T, C, B)
```

Using Classical Heuristics to Guide HTN Search

Simulating Composition – Resulting Model

\[
\begin{align*}
\text{at}(v, l_1) \quad & \text{drive}(v, l_1, l_2) \quad \text{at}(v, l_2) \quad \text{at}(v, l_1) \\
\text{road}(l_1, l_2) \quad & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qua
Planning in the Transformed Model

Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Planning in the Transformed Model

Heuristic value: 10
Planning in the Transformed Model

\[
\begin{align*}
\text{deliver}(P, D) & \quad \text{m-deliver}(P, C, D, T) \\
\text{get-to}(T, C) & \quad \text{m-via}(T, B, C) \\
\text{pick-up}(T, C, P) & \quad \text{get-to}(T, D) \quad \text{m-via}(T, B, D) \\
\text{drop}(T, D, P) & \quad \text{get-to}(T, B) \quad \text{drive}(T, B, C) \\
\text{drive}(T, A, B) & \quad \text{drive}(T, B, D) \\
\end{align*}
\]
Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Planning in the Transformed Model

dm Deliver(P, D) m-deliver(P, C, D, T)

gt(T, C) m-via(T, B, C)

gt(T, B) m-direct(T, A, B)

drive(T, B, C)

drive(T, A, B)

gt(T, D) m-via(T, B, D)

gt(T, B) m-direct(T, C, B)

drive(T, B, D)

drive(T, C, B)

drive(T, A, B)

{at(T, A), at(P, C)}

{at(T, B), at(P, C), b-drive(T, A, B)}

{at(T, B), at(P, C), b-drive(T, A, B), b-get-to(T, B)}

{at(T, C), at(P, C), b-drive(T, A, B), b-get-to(T, B), b-drive(T, B, C)}

{at(T, C), at(P, T), b-drive(T, A, B), b-get-to(T, B), b-drive(T, B, C), b-pick-up(T, C, P)}
Using Classical Heuristics to Guide HTN Search

Planning in the Transformed Model

Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Planning in the Transformed Model

Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning
Planning in the Transformed Model

Using Classical Heuristics to Guide HTN Search

Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Planning in the Transformed Model

Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Planning in the Transformed Model

Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

\[ \text{deliver}(P, D) \]
\[ m\text{-deliver}(P, C, D, T) \]
\[ \text{get-to}(T, C) \]
\[ m\text{-via}(T, B, C) \]
\[ \text{pick-up}(T, C, P) \]
\[ \text{get-to}(T, D) \]
\[ m\text{-via}(T, B, D) \]
\[ \text{drop}(T, D, P) \]
\[ \text{get-to}(T, B) \]
\[ m\text{-direct}(T, A, B) \]
\[ \text{drive}(T, B, C) \]
\[ \text{drive}(T, C, B) \]

{\text{at}(T, A), \text{at}(P, C)}

Diagram:
- B
- C
- D
- A
- Truck
- Archive
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

- $\text{deliver}(P, D)$
- $\text{m-deliver}(P, C, D, T)$
- $\text{get-to}(T, C)$
- $\text{m-via}(T, B, C)$
- $\text{pick-up}(T, C, P)$
- $\text{get-to}(T, D)$
- $\text{m-via}(T, B, D)$
- $\text{drop}(T, D, P)$
- $\text{get-to}(T, B)$
- $\text{m-direct}(T, A, B)$
- $\text{drive}(T, B, C)$
- $\text{drive}(T, C, B)$
- $\text{drive}(T, A, B)$

Heuristics

Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

```plaintext
Solving Techniques

Heuristics

Excursion

Heuristic Calculation (Delete Relaxed)

\[
deliver(P, D) \quad m\text{-deliver}(P, C, D, T)
\]

\[
get-to(T, C) \quad m\text{-via}(T, B, C)
\]

\[
pick-up(T, C, P) \quad get-to(T, D) \quad m\text{-via}(T, B, D)
\]

\[
drop(T, D, P) \quad \text{get-to}(T, C) \quad m\text{-direct}(T, A, B)
\]

\[
drive(T, B, C) \quad m\text{-direct}(T, C, B) \quad drive(T, B, D) \quad \text{get-to}(T, B)
\]

\[
drive(T, A, B) \quad drive(T, C, B) \quad \text{get-to}(T, B)
\]

\[
\{ \text{at}(T, A), \text{at}(P, C) \} \quad \{ \text{at}(T, A), \text{at}(T, B), \text{at}(P, C), \text{b\text{-drive}}(T, A, B) \} \quad \text{am\text{-direct}}(T, A, B)
\]

\[
\text{...}
\]

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

June 25th, ICAPS 2018 (Delft)
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)
Heuristic Calculation (Delete Relaxed)

Using Classical Heuristics to Guide HTN Search

- Heuristics
- Excursion

Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning
Heuristic Calculation (Delete Relaxed)

Using delete-relaxed classical heuristic: 9
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

Using delete-relaxed classical heuristic: 9
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

\[
deliver(P, D) \quad m-deliver(P, C, D, T)
\]

\[
get-to(T, C) \quad m-via(T, B, C)
\]

\[
pick-up(T, C, P) \quad get-to(T, D) \quad m-via(T, B, D)
\]

\[
drop(T, D, P)
\]

\[
get-to(T, B) \quad m-direct(T, A, B)
\]

\[
drive(T, B, C)
\]

\[
drive(T, B, D)
\]

\[
drive(T, C, B)
\]

\[
\{at(T, A), at(P, C)\}
\]

\[
drive(T, A, B)
\]

\[
\{at(T, A), at(T, B), at(P, C), b-drive(T, A, B)\}
\]

\[
am-direct(T, A, B) \quad am-via(T, B, C)
\]

\[
drive(T, B, C) \quad drive(T, B, D)
\]

\[
pick-up(T, C, P) \quad drop(T, D, P)
\]

\[
\}
\]

...
Heuristic Calculation (Delete Relaxed)

\[
deliver(P, D) \\
m-deliver(P, C, D, T)
\]

\[
get-to(T, C) \\
m-via(T, B, C)
\]

\[
pick-up(T, C, P) \\
m-via(T, B, D)
\]

\[
get-to(T, D) \\
drop(T, D, P)
\]

\[
get-to(T, B) \\
m-direct(T, A, B)
\]

\[
drive(T, B, C)
\]

\[
drive(T, A, B)
\]

\[
\{at(T, A), at(P, C)\}
\]

\[
\{at(T, A), at(T, B), at(P, C), \text{drive}(T, A, B)\}
\]

\[
\text{drive}(T, A, B)
\]

\[
\text{drive}(T, C, B)
\]

\[
\text{am-direct}(T, A, B)
\]

\[
\text{driv}(T, B, C)
\]

\[
\text{pick-up}(T, C, P)
\]

\[
\text{am-via}(T, B, C)
\]

\[
\text{drive}(T, B, D)
\]

\[
\text{drop}(T, D, P)
\]

\[
\text{am-via}(T, B, D)
\]
Using delete-relaxed classical heuristic: 9
Using Classical Heuristics to Guide HTN Search

General Characteristics

- Simulates task composition
Using Classical Heuristics to Guide HTN Search

General Characteristics

- Simulates task composition
- Incorporates hierarchical reachability information
- Combines it with information on state-based executability
- Solves the problem of a missing state-based goal
Using Classical Heuristics to Guide HTN Search

General Characteristics

- Simulates task composition
- Incorporates hierarchical reachability information
- Combines it with information on state-based executability
- Solves the problem of a missing state-based goal

- The transformation from HTN to classical problem is a relaxation
  - The set of valid solutions increases
Using Classical Heuristics to Guide HTN Search

General Characteristics

- Simulates task composition
- Incorporates hierarchical reachability information
- Combines it with information on state-based executability
- Solves the problem of a missing state-based goal

The transformation from HTN to classical problem is a relaxation

→ The set of valid solutions increases

Heuristic function is allowed to do

- Task sharing (every task must be proceeded only once)
- Task insertion (e.g. to fulfill preconditions)
- HTN ordering relations are relaxed
General Characteristics

- Simulates task composition
- Incorporates hierarchical reachability information
- Combines it with information on state-based executability
- Solves the problem of a missing state-based goal

- The transformation from HTN to classical problem is a relaxation
  → The set of valid solutions increases

- Heuristic function is allowed to do
  - Task sharing (every task must be proceeded only once)
  - Task insertion (e.g. to fulfill Preconditions)
  - HTN ordering relations are relaxed

- Heuristic function may only insert tasks that lie within the decomposition hierarchy (not given here)
Computational Aspects

- Size is **linear** in the input HTN domain, but the model is **large**
- **State** and **action** set are **extended**
Computational Aspects

- Size is **linear** in the input HTN domain, but the model is **large**
- **State** and **action** set are **extended**
- Most parts of the **model** are **static** during search, one needs to update
  - Initial state
  - Goal

→ **Efficient update** of the “heuristic model” possible
Computational Aspects

- Size is **linear** in the input HTN domain, but the model is **large**
- **State** and **action** set are **extended**
- Most parts of the **model** are **static** during search, one needs to update
  - Initial state
  - Goal
  - **Efficient update** of the “heuristic model” possible
  - Classical heuristic combined with the encoding should be able to deal with changed goal efficiently
Perfect HTN solution (in terms of modifications) corresponds to a classical plan in the transformation with equal costs.

Perfect classical heuristic on the transformation has **less or equal costs**.
Using Classical Heuristics to Guide HTN Search

Resulting Heuristic Values

- Perfect HTN solution (in terms of modifications) corresponds to a classical plan in the transformation with equal costs.
- Perfect classical heuristic on the transformation has **less or equal costs**.
- When the used classical heuristic has one of the following properties, the resulting HTN heuristic has it too:
  - Safety
  - Goal-awareness
  - Admissibility
Can be combined with many classical heuristics
Can be combined with many classical heuristics

In principle applicable in both – plan space or progression search

→ Progression search provides more precise state information
Discussing Classical Heuristics to Guide HTN Search

- Can be combined with many classical heuristics
- In principle applicable in both – plan space or progression search
  - Progression search provides more precise state information
- Comparison to “HTN to STRIPS/ADL translation”
  - This transformation is a relaxation (set of solutions changes)
  - It is smaller
  - It is easier to compute
Overview Part II

Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search

- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT

- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms
Which variants of HTN planning and further hierarchical planning problem classes exist?
Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
- Task sharing
Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
- Task sharing
- Hybrid planning (i.e., HTN + POCL Planning)
Overview of Hierarchical Planning Variants

Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
- Task sharing
- Hybrid planning (i.e., HTN + POCL Planning)
- Decompositional planning (i.e., hybrid without initial plan)
Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
- Task sharing
- Hybrid planning (i.e., HTN + POCL Planning)
- Decompositional planning (i.e., hybrid without initial plan)
- GTN planning (decompose goals, not tasks)
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*. 
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.
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 $\overline{m}$ 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)
Task sharing allows unconstrained tasks to be merged:

Let $\mathcal{P}^* = (V, P, \delta, C, M, s_I, c_I)$ be an **HTN problem with task sharing**.
Task sharing allows unconstrained tasks to be merged:

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

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

- There is a sequence of decomposition methods $\overline{m}$ and task mergings that transform $c_I$ into $tn$ (two tasks can be merged if they are identical and not ordered with respect to another),
- $tn$ 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 sharing:

- Allows to eliminate duplicates that might just be modeling artifacts

\[
\text{connect}(\text{DVD-player, Adapter}) \quad \xrightarrow{<} \quad \text{connect}(\text{Adapter, TV})
\]

\[
\text{connect}(\text{Blu-ray-player, Adapter}) \quad \xrightarrow{<} \quad \text{connect}(\text{Adapter, TV})
\]

← actions come from some method

← actions come from another method
Benefits of allowing task sharing:

- Allows to eliminate duplicates that might just be modeling artifacts

```
connect(DVD-player, Adapter) < connect(Blu-ray-player, Adapter) < task sharing < connect(Adapter, TV) < actions come from some method
```

```
connect(DVD-player, Adapter) < connect(Adapter, TV) < actions come from another method
```

```
connect(Blu-ray-player, Adapter) < connect(Adapter, TV)
```
**Hybrid planning** fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

---

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

June 25th, ICAPS 2018 (Delft)
**Hybrid planning** fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Core differences to standard HTN planning:
- Compound tasks can have preconditions and effects as well
Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Core differences to standard HTN planning:

- Compound tasks can have preconditions and effects as well
- Decomposition methods must adhere certain criteria (so that they are implementations of their compound tasks)
Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Core differences to standard HTN planning:

- Compound tasks can have preconditions and effects as well
- Decomposition methods must adhere certain criteria (so that they are implementations of their compound tasks)
- Rather than task networks, we have partial plans that may contain causal links
**Hybrid planning** fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Core differences to standard HTN planning:

- Compound tasks can have preconditions and effects as well
- Decomposition methods must adhere certain criteria (so that they are implementations of their compound tasks)
- Rather than task networks, we have partial plans that may contain causal links
- In solution plans, *all* linearizations must be executable
**Hybrid planning** fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.
Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.
**Hybrid planning** fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.
Benefits of hybrid planning:

- Modeling support due to preconditions and effects of compound tasks and legality criteria for their methods.
Benefits of hybrid planning:

- Modeling support due to preconditions and effects of compound tasks and legality criteria for their methods
- In combination with task insertion: compound tasks can be inserted easier due to their preconditions and effects
Benefits of hybrid planning:

- Modeling support due to preconditions and effects of compound tasks and legality criteria for their methods.
- In combination with task insertion: compound tasks can be inserted easier due to their preconditions and effects.
- Solution criteria (all linearizations are executable) is more practical than the classical one (there exist an executable linearization).
Benefits of hybrid planning:

- Modeling support due to preconditions and effects of compound tasks and legality criteria for their methods
- In combination with task insertion: compound tasks can be inserted easier due to their preconditions and effects
- Solution criteria (*all* linearizations are executable) is more practical than the classical one (there *exist* an executable linearization)
- Plan explanation and visualization becomes more natural
Decomposition planning is defined just as hybrid planning with task insertion – with the exception that there is no initial partial plan.
Benefits of decompositional planning:

- Everything like in hybrid planning, except:
  
  lower expressivity (identical to non-hierarchical, classical planning), because the hierarchy does not induce constraints
Hierarchical Goal Network (HGN) planning is concerned with the decomposition of goals instead of tasks.
Hierarchical Goal Network (HGN) planning is concerned with the decomposition of goals instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
Hierarchical Goal Network (HGN) planning is concerned with the decomposition of goals instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
- Instead of task networks, HGN planning uses goal networks: partially ordered sets of goals (each being a formula over state variables)
Hierarchical Goal Network (HGN) planning is concerned with the decomposition of \textit{goals} instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
- Instead of task networks, HGN planning uses \textit{goal networks}: partially ordered sets of goals (each being a formula over state variables)
- Decomposition methods refine/substitute \textit{goals} rather than \textit{tasks}
Hierarchical Goal Network (HGN) planning is concerned with the decomposition of goals instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
- Instead of task networks, HGN planning uses goal networks: partially ordered sets of goals (each being a formula over state variables)
- Decomposition methods refine/substitute goals rather than tasks
- The hierarchy induced on goals does not partition them into primitive and non-primitive goals
Hierarchical Goal Network (HGN) planning is concerned with the decomposition of goals instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
- Instead of task networks, HGN planning uses goal networks: partially ordered sets of goals (each being a formula over state variables)
- Decomposition methods refine/substitute goals rather than tasks
- The hierarchy induced on goals does not partition them into primitive and non-primitive goals
- All actions can be applied to the current state, as long as they achieve a possibly first goal
Benefits of HGN planning?

- The application of state-based heuristics is more directly applicable than in HTN planning
**Benefits of HGN planning?**

- The application of state-based heuristics is more directly applicable than in HTN planning.
- In some domains, defining a hierarchy on state features might be easier than defining a hierarchy on tasks.
Thank you for your attention!

Are there questions?