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)
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**
**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)

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)

get-to(v, l)

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

no-op()

mnoop(v, l)

l1 \neq l2
```

**Diagram:**

- **Notation:**
  - `deliver(p, l2)`: Delivering a package `p` to location `l2`.
  - `get-to(v, l)`: Getting to location `l`.
  - `pick-up(v, l1, p)`: Picking up a package `p` at location `l1`.
  - `drop(v, l2, p)`: Dropping a package `p` at location `l2`.
  - `drive(v, l1, l2)`: Driving from location `l1` to location `l2`.
  - `m-direct(v, l1, l2)`: Direct movement from `l1` to `l2`.
  - `m-via(v, l1, l2)`: Via movement from `l1` to `l2`.
  - `m-noop(v, l)`: No operation at location `l`.

**Example:**

- **Scenario:**
  - **Initial State:** Location `A` with truck `T1`.
  - **Goal:** Deliver package `p` from `A` to `B`.

- **Steps:**
  1. Get to `l1`.
  2. Pick up package `p`.
  5. Drive from `l2` to `l1`.
  6. No operation at `l1`.
  7. Drive from `l1` to `l2`.
  8. No operation at `l2`.

**Diagram Representation:**

- **Nodes:**
  - `A` and `B` with corresponding trucks `T1` and `T2`.
  - Locations `l1` and `l2`.

- **Edges:**
  - Directed arrows between nodes indicating the sequence of operations.

**Notes:**

- **Concepts:**
  - HTN Planning:
    - Hierarchical Task Network planning is a planning approach where tasks are organized into a hierarchy with primitive tasks at the leaf nodes.
    - The goal is to find a sequence of primitive actions that achieves the overall goal.
    - The HTN planning algorithm is typically used to solve complex problems by decomposing them into simpler subproblems.

- **Application:**
  - In logistics and transportation, HTN planning can be used to optimize delivery routes and schedules.
  - It can also be applied in robotics for task planning in environments with unknown or changing conditions.
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
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
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:
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
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
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** $t^\frac{1}{2} (t', c, t'')$: 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 $t$. 
**Flaws in Partial Plans**

*get-to(T₁,B)*

**Modifications for compound task flaws:**

- Decompose the compound task (one modification for each method)
Modifications for *open precondition flaws*:
Modifications for *open precondition flaws*:
- Insert a causal link from existing plan step (one modification for each possible producer)
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)
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*:
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.
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
Standard Plan Space-based Algorithm

Input: fringe = \{ P_{\text{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

- Initial partial plan \( P_{\text{init}} \) equals the initial task network preceded by an artificial task encoding the initial state.
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 $\langle T, \prec, \alpha, CL \rangle$.
Standard Plan Space-based Algorithm

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

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

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

- **Fringe** is sorted according to some heuristic.
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.     \[ P := \text{PlanSel}(\text{fringe}) \]
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

- **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
Initial partial plan $P_{\text{init}}$ 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 a the set of all flaws of the current partial plan.

FlawSel selects (not a backtrack point!) a flaw according to a flaw selection strategy.
### 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. \( P := \text{PlanSel}(fringe) \)
3. \( F := \text{FlawDet}(P) \)
4. if \( F = \emptyset \) then return \( P \)
5. \( f := \text{FlawSel}(F) \)
6. \( fringe := (fringe \setminus \{ P \}) \cup \text{Successors}(P, f) \)
7. return `fail`

**Flaws**

- `at(T_1, A)`
- `at(P_1, A)`
- `road(B, A)`
- `road(A, B)`
- `at(T_2, C)`
- `at(P_2, C)`
- `road(D, C)`
- `road(C, D)`

**Modifications**

- `deliver(P_1, B)`
- `deliver(P_2, D)`
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 ≠ \emptyset do
2     P := PlanSel(fringe)
3     F := FlawDet(P)
4     if F = \emptyset then return P
5     f := FlawSel(F)
6     fringe := (fringe \ {P}) ∪ Successors(P, f)
7 return fail

---

Flaws

<table>
<thead>
<tr>
<th>compound task: deliver(P_1, B)</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>decompose with m-deliver(P_1, A, B, T_1)</td>
<td></td>
</tr>
</tbody>
</table>

| compound task: deliver(P_2, D) |
| decompose with m-deliver(P_2, C, D, T_2) |
**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 ≠ ∅ do
2    P := PlanSel(fringe)
3    F := FlawDet(P)
4    if F = ∅ then return P
5    f := FlawSel(F)
6    fringe := (fringe \ {P}) \cup\ Successors(P, f)
7 return fail

Flaws

compound task: \text{deliver}(P_2, D)  
compound task: \text{get-to}(T_1, A)  
compound task: \text{get-to}(T_1, B)  
compound task: \text{drop}(T_1, B, P_1)

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)

decompose \text{get-to}(T_1, A) with m-direct(T_1, B, A)
decompose \text{get-to}(T_1, A) with m-via(T_1, B, A)

insert causal link from init

decompose \text{get-to}(T_1, A) with m-direct(T_1, B, A)
decompose \text{get-to}(T_1, A) with m-via(T_1, B, A)

decompose \text{get-to}(T_1, A) with m-direct(T_1, B, A)
decompose \text{get-to}(T_1, A) with m-via(T_1, B, A)

insert causal link from init

decompose \text{get-to}(T_1, B) with m-direct(T_1, A, B)
decompose \text{get-to}(T_1, B) with m-via(T_1, A, B)

insert causal link from pickup(T_1, A, P_1)
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 \setminus \{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 init
  - decompose get-to\((T_1, A)\) with \( m\text{-direct}(T_1, B, A) \)
  - decompose get-to\((T_1, A)\) with \( m\text{-via}(T_1, B, A) \)

- at\((P_1, A)\) of pick-up\((T_1, A, P_1)\)
  - insert causal link from init

- 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 \( m\text{-via}(T_1, A, B) \)

- 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 := 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)
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

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)
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

### Modifications

**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\((T_1, A, B)\)
Solving Techniques

Heuristics

Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

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

Output: A solution plan or fail.

\[\begin{align*}
\text{while } & \text{ fringe } \neq \emptyset \text{ do} \\
P & := \text{PlanSel}(\text{fringe}) \\
F & := \text{FlawDet}(P) \\
\text{if } & F = \emptyset \text{ then return } P \\
f & := \text{FlawSel}(F) \\
\text{fringe := ( fringe \ \backslash \ \{P\} )} \\
\cup & \text{Successors}(P, f) \\
\text{return fail}
\end{align*}\]
**Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning**

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

---

**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.  \( \text{fringe} := (\text{fringe} \setminus \{ P \}) \cup \text{Successors}(P, f) \)
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\((T_1, B)\) of drop\((T_1, B, P_1)\)

**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 \text{init}
- decompose get-to\((T_1, A)\) with m-direct\((T_1, B, A)\)
- decompose get-to\((T_1, A)\) with m-via\((T_1, B, A)\)
- decompose get-to\((T_1, B)\) with m-direct\((T_1, A, B)\)
- decompose get-to\((T_1, B)\) with m-via\((T_1, A, B)\)

---

**Diagram**

- HTN Plan Space Search
- Standard Plan Space-based Algorithm

---

**Overview**

- Heuristics
- Solving Techniques
- 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**

<table>
<thead>
<tr>
<th>compound task: get-to(T₁, A)</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>decompose with m-direct(T₁, B, A)</td>
<td></td>
</tr>
<tr>
<td>decompose with m-via(T₁, B, A)</td>
<td></td>
</tr>
<tr>
<td>decompose with m-noop(T₁, A)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>open prec.: at(T₁, A) of pick-up(T₁, A, P₁)</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert causal link from init</td>
<td></td>
</tr>
<tr>
<td>decompose get-to(T₁, A) with m-direct(T₁, B, A)</td>
<td></td>
</tr>
<tr>
<td>decompose get-to(T₁, A) with m-via(T₁, B, A)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>compound task: get-to(T₁, B)</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>decompose with m-direct(T₁, A, B)</td>
<td></td>
</tr>
<tr>
<td>decompose with m-via(T₁, A, B)</td>
<td></td>
</tr>
<tr>
<td>decompose with m-noop(T₁, B)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>open prec.: at(T₁, B) of drop(T₁, B, P₁)</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>decompose get-to(T₁, B) with m-direct(T₁, A, B)</td>
<td></td>
</tr>
<tr>
<td>decompose get-to(T₁, B) with m-via(T₁, A, B)</td>
<td></td>
</tr>
</tbody>
</table>

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

Flaws

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

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

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

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

...
**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}) \cup Successors(P,f)
   7. 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)

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

...
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

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)

- open prec.: road(A, B) of drive(T_1, A, B)
  - insert causal link from init

...
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 := 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: 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)

open prec.: road(A, B) 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 \textit{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 \textit{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) 9 / 62
**H TN 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. \( \text{fringe} := (\text{fringe} \setminus \{P\}) \cup \text{Successors}(P, f) \)
7. 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)
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. \( \text{fringe} := (\text{fringe} \setminus \{P\}) \cup \text{Successors}(P, f) \)
   7. 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 \( \text{init} \)
  - decompose get-to\((T_1, A)\) with \( m\text{-direct}(T_1, B, A) \)
  - decompose get-to\((T_1, A)\) with \( m\text{-via}(T_1, B, A) \)

- **open prec.**: at\((T_1, A)\) of drive\((T_1, A, B)\)
  - 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 \( m\text{-via}(T_1, B, A) \)

...
Solving Techniques

HTN Plan Space Search

Standard Plan Space-based Algorithm

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

Output : A solution plan or fail.

\[
\textbf{while } \text{ fringe } \neq \emptyset \textbf{ do}
\]
\[
\begin{align*}
P & := \text{PlanSel}(\text{fringe}) \\
F & := \text{FlawDet}(P) \\
\text{if } F = \emptyset \text{ then return } P \\
f & := \text{FlawSel}(F) \\
\text{fringe} & := (\text{fringe} \setminus \{P\}) \\
& \cup \text{Successors}(P, f)
\end{align*}
\]

\textbf{return} fail

\[
\begin{align*}
\textbf{Flaws} & \quad \textbf{Hypotheses} \\
\text{compound task: } & \text{get-to}(T_1, A) \\
\text{decompose with } & \text{m-direct}(T_1, B, A) \\
\text{decompose with } & \text{m-via}(T_1, B, A) \\
\text{decompose with } & \text{m-noop}(T_1, A)
\end{align*}
\]

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

\[
\begin{align*}
\text{open prec.: } & \text{at}(T_1, A) \text{ of drive}(T_1, A, B) \\
\text{insert causal link from } & \text{init} \\
\text{decompose } & \text{get-to}(T_1, A) \text{ with m-direct}(T_1, B, A) \\
\text{decompose } & \text{get-to}(T_1, A) \text{ with m-via}(T_1, B, A)
\end{align*}
\]
Solving Techniques

 HTN Plan Space Search

Standard Plan Space-based Algorithm

---

**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)

---

**Modifications**

---

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 \( \text{fringe} := (\text{fringe} \setminus \{P\}) \cup \text{Successors}(P, f) \)
7 return fail
Solving Techniques
Heuristics
Excursion

HTN Plan Space Search

Standard Plan Space-based Algorithm

Flaws Modifications

<table>
<thead>
<tr>
<th>Flaws</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>open prec: at(T₁, B) of drive(T₁, B, A)</td>
<td>—</td>
</tr>
<tr>
<td>open prec.: road(B, A) of drive(T₁, B, A)</td>
<td>insert causal link from init</td>
</tr>
<tr>
<td>open prec.: at(T₁, A) of pick-up(T₁, A, P₁)</td>
<td>insert causal link from init</td>
</tr>
<tr>
<td>open prec.: at(T₁, A) of drive(T₁, A, B)</td>
<td>insert causal link from init</td>
</tr>
<tr>
<td>open prec.: at(T₁, A) of drive(T₁, B, A)</td>
<td>insert causal link from drive(T₁, B, A)</td>
</tr>
</tbody>
</table>

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\}) \cup Successors(P, f)
7 return fail
Input : fringe = \{P_{init}\}

Output : A solution plan or fail.

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

**Flaws Modifications**

- open prec: at(T₁, B) of drive(T₁, B, A) —
- open prec: road(B, A) of drive(T₁, B, A) insert causal link from init
- 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
- open prec: at(T₁, A) of drive(T₁, B, A) insert causal link from init

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

**Flaws**

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

... ...

**Diagram**

- Nodes represent tasks and actions.
- Edges indicate the flow of plan execution.
- Highlighted tasks indicate potential flaws and 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
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 return fail

Flaws Modifications

open prec.: at(T_1, A) of pick-up(T_1, A, P_1) insert causal link from init
open prec.: at(T_1, A) of drive(T_1, A, B) insert causal link from init
... ...

... ...
... ...

at(T_1, A)
at(P_1, A)
road(B, A)
road(A, B)
at(T_2, C)
at(P_2, C)
road(D, C)
road(C, D)

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

... ...
... ...

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

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

**HTN Plan Space Search**

**Standard Plan Space-based Algorithm**

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

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

```plaintext
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\}
7           \cup Successors(P, f)
8    return fail
```

**Flaws**

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

**Modifications**

- insert causal link from init
**Solving Techniques**

**Heuristics**

**Excursion**

---

**HTN Plan Space Search**

**Standard Plan Space-based Algorithm**

---

**Input**:

\[ \text{fringe} = \{ P_{\text{init}} \} \]

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

---

1. while \( \text{fringe} \neq \emptyset \) 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. \( \text{fringe} := (\text{fringe} \setminus \{ P \}) \cup \text{Successors}(P, f) \)
7. return fail

---

**Flaws**

- open prec.: \( \text{at}(T_1, A) \) of \( \text{drive}(T_1, A, B) \)

**Modifications**

- insert causal link from \( \text{init} \)

---

...
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\}) ∪ Successors(P, f)
7. 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 = ∅ then return P
5 \[ f := \text{FlawSel}(F) \]
6 fringe := (fringe \ {P}) ∪ Successors(P, f)
7 return fail
Input: \( \text{fringe} = \{ P_{\text{init}} \} \)

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

1. \textbf{while} \( \text{fringe} \neq \emptyset \) \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} \( \text{fringe} := (\text{fringe} \setminus \{P\}) \cup \text{Successors}(P, f) \)
7. \hspace{1em} \textbf{return} \text{fail}
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})                       
                             \cup Successors(P, f)
   7 return fail

This partial plan has no flaws, so it is a solution and returned
Plan Space-based search is **sound**
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 – 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)
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**.
Progression Search – Basic Characteristics

- 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
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

1. \( fringe \leftarrow \{(s_I, tn_I, ())\} \)
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. \( U \leftarrow n.\text{unconstrainedNodes} \)
6. \( \textbf{for} \ t \in U \ \textbf{do} \)
7. \( \quad \textbf{if} \ isPrimitive(t) \ \textbf{then} \)
8. \( \quad \quad n' \leftarrow n.\text{apply}(t) \)
9. \( \quad \quad fringe.\text{add}(n') \)
10. \( \quad \textbf{else} \)
11. \( \quad \quad \textbf{for} \ m \in t.\text{methods} \ \textbf{do} \)
12. \( \quad \quad \quad n' \leftarrow n.\text{decompose}(t, m) \)
13. \( \quad \quad \quad fringe.\text{add}(n') \)

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.
Action application removes node and causes state transition.
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s_I, t_{n_I}, ())\} \)
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. \( U \leftarrow n.\text{unconstrainedNodes} \)
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. \( \text{fringe.add}(n') \)

- **Search nodes** contain task network, state, and solution prefix.
HTN Progression Search

Standard Progression Algorithm

1. \( fringe \leftarrow \{(s_I, tn_I, ()\}\} \)
2. \( \textbf{while} \ fringe \neq \emptyset \ \textbf{do} \)
   3. \( n \leftarrow fringe.poll() \)
   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. \( fringe.\text{add}(n') \)
      10. \( \textbf{else} \)
          11. \( \textbf{for} \ m \in t.\text{methods} \ \textbf{do} \)
              12. \( n' \leftarrow n.\text{decompose}(t, m) \)
              13. \( fringe.\text{add}(n') \)

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

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

1. \( fringe \leftarrow \{(s_I, t_{nI}, ())\} \)
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. Initialize the fringe with the initial state and an empty task network:
   
   \[ fringe \leftarrow \{ (s_{I}, t_{I}, ()) \} \]

2. While the fringe is not empty, do:
   
   - Pop the next node from the fringe:
     
     \[ n \leftarrow fringe.poll() \]
   - If the node is a goal, return the node:
     
     \[ \text{if } n.\text{isgoal} \text{ then return } n \]
   - If the task is unconstrained, add its primitive action:
     
     \[ U \leftarrow n.\text{unconstrainedNodes} \]
     
     For each task \( t \) in \( U \):
     
     - If the task is primitive:
       
       \[ n^{' } \leftarrow n.\text{apply}(t) \]
     
     - Otherwise, decompose the task:
       
       \[ n^{' } \leftarrow n.\text{decompose}(t, m) \]
   
3. Add the updated node to the fringe:
   
   \[ fringe.\text{add}(n^{' }) \]
Standard Progression Algorithm

1. \( fringe \leftarrow \{(s_I, tn_I, ())\}\)
2. \( \text{while} \ fringe \neq \emptyset \ \text{do} \)
   3. \( n \leftarrow fringe\.poll() \)
   4. \( \text{if} \ n\.isgoal \ \text{then} \ \text{return} \ n \)
   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)
- **Unconstrained tasks** have no predecessor
- **Action application** removes node and causes state transition
1. \( fringe \leftarrow \{(s, t_1, (\))\} \)

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. \( fringe.add(n') \)

\[ \pi = () \]

**Example:**

- \( deliver(P_1, B) \)
- \( deliver(P_2, D) \)
- \( drive(T_2, A, B) \)
- \( no-op() \)
- \( pick-up(T_1, A, P_1) \)
- \( drop(T_1, B, P_1) \)
- \( deliver(P_2, D) \)
- \( pick-up(T_2, C, P_2) \)
- \( drop(T_2, D, P_2) \)
HTN Progression Search

Standard Progression Algorithm

1. $\text{fringe} \leftarrow \{(s_t, \text{tn}_t, ())\}$
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. $\text{fringe.add}(n')$

$\pi = ()$
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')
      else
         10. for m ∈ t.methods do
             11. n' ← n.decompose(t, m)
             12. fringe.add(n')

\( \pi = () \)

get-to(T_1, A) ... \langle pick-up(T_1, A, P_1) ... get-to(T_1, B) ... drop(T_1, B, P_1) \rangle

deliver(P_2, D)
**HTN Progression Search**

**Standard Progression Algorithm**

1. $\text{fringe} \leftarrow \{(s, tn, ())\}$
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.             $\text{fringe.add}(n')$
10.        else
11.            for $m \in t.\text{methods}$ do
12.                $n' \leftarrow n.\text{decompose}(t, m)$
13.                $\text{fringe.add}(n')$

$\pi = ()$
Solving Techniques
Heuristics
Excursion

HTN Progression Search

Standard Progression Algorithm

1. fringe ← \{(s_i, t_{n_i}, ())\}
2. while fringe ≠ \{
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.\}

\quad π = ( )

\begin{align*}
\text{deliver}(P_1, B) \\
\text{get-to}(T_2, C) \cdots \text{pick-up}(T_2, C, P_2) \cdots \text{get-to}(T_2, D) \cdots \text{drop}(T_2, D, P_2) \end{align*}
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

```
fringe ← {(s₁, t₁, ())}

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')

get-to(T₁, A) ... pick-up(T₁, A, P₁) ... get-to(T₁, B) ... drop(T₁, B, P₁)

deliver(P₂, D)

π = ()
```
**Solving Techniques**

**Heuristics**

**Excursion**

**HTN Progression Search**

**Standard Progression Algorithm**

\[
\begin{align*}
\text{fringe} & \leftarrow \{(s_i, t_{i1}, ())\} \\
\text{while} \; \text{fringe} \neq \emptyset \; \text{do} \\
\text{n} & \leftarrow \text{fringe.poll()} \\
\text{if} \; n.\text{isgoal} \; \text{then} \; \text{return} \; n \\
U & \leftarrow n.\text{unconstrainedNodes} \\
\text{for} \; t \in U \; \text{do} \\
\quad \text{if} \; \text{isPrimitive}(t) \; \text{then} \\
\quad \quad n' & \leftarrow n.\text{apply}(t) \\
\quad \quad \text{fringe.add}(n') \\
\quad \text{else} \\
\quad \quad \text{for} \; m \in t.\text{methods} \; \text{do} \\
\quad \quad \quad n' & \leftarrow n.\text{decompose}(t, m) \\
\quad \quad \quad \text{fringe.add}(n') \\
\end{align*}
\]

\[\pi = ()\]

---

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

June 25th, ICAPS 2018 (Delft)
Solving Techniques

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s, t^n, \cdot)\} \)
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. \( \text{fringe.add}(n') \)
10. else
11. for \( m \in t.\text{methods} \) do
12. \( n' \leftarrow n.\text{decompose}(t, m) \)
13. \( \text{fringe.add}(n') \)

\( \pi = (\)
Solving Techniques

HTN Progression Search

Standard Progression Algorithm

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 = (\) \]

```
get-to(T1, A) ... pick-up(T1, A, P1) ... get-to(T1, B) ... drop(T1, B, P1)
```

```
deliver(P2, D)
m-deliver(P2, C, D, T2)
```

```
get-to(T2, C) ... pick-up(T2, C, P2) ... get-to(T2, D) ... drop(T2, D, P2)
```
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)`
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) \leftarrow \ldots \\downarrow \;\text{pick-up}(T_1, A, P_1) \ldots \text{get-to}(T_1, B) \ldots \;\text{drop}(T_1, B, P_1)
```

```
get-to(T_2, C) \leftarrow \ldots \text{pick-up}(T_2, C, P_2) \ldots \text{get-to}(T_2, D) \ldots \text{drop}(T_2, D, P_2)
```
**Standard Progression Algorithm**

1. \( \text{fringe} \leftarrow \{(s, tn, ())\} \)
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** \( n.\text{isPrimitive}(t) \) **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. \( \text{fringe.add}(n') \)

\[ \pi = () \]
1. $\text{fringe} \leftarrow \{(s, t, n, ())\}$
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.             $\text{fringe.add}(n')$
10.        else
11.            for $m \in t.\text{methods}$ do
12.                $n' \leftarrow n.\text{decompose}(t, m)$
13.                $\text{fringe.add}(n')$

$\pi = ()$
Solving Techniques

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 = ()

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

deliver(P₁, B)

\(\text{pick-up}(T₂, C, P₂)\) \(\leadsto\) \(\text{get-to}(T₂, D)\) \(\leadsto\) \(\text{drop}(T₂, D, P₂)\)
HTN Progression Search

Standard Progression Algorithm

```
 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′)
```

```
get-to(T₁, A) ≺ pick-up(T₁, A, P₁) ≺ get-to(T₁, B) ≺ drop(T₁, B, P₁)

get-to(T₂, C) ≺ pick-up(T₂, C, P₂) ≺ get-to(T₂, D) ≺ drop(T₂, D, P₂)
```

π = ()
```
fringe ← {(sI, tnI, ())}
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 = ()
```

```plaintext
get-to(T₁, A)≺pick-up(T₁, A, P₁)≺get-to(T₁, B)≺drop(T₁, B, P₁)

get-to(T₂, C)≺pick-up(T₂, C, P₂)≺get-to(T₂, D)≺drop(T₂, D, P₂)
```
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)
9.            fringe.add(n')
10.       else
11.          for m ∈ t.methods do
12.              n' ← n.decompose(t, m)
13.              fringe.add(n')

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

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

π = ()
HTN Progression Search

Standard Progression Algorithm

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

\(\pi = \langle \rangle\)

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

get-to(\(T_2, C\)) \(\Prec\) pick-up(\(T_2, C, P_2\)) \(\Prec\) get-to(\(T_2, D\)) \(\Prec\) drop(\(T_2, D, P_2\))

\(\ldots\)

\(\pi = \langle \rangle\)
fringe ← \{(s_1, t_{n_1}, ())\}

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')

\[ \pi = () \]
Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning

June 25th, ICAPS 2018 (Delft)
HTN Progression Search

Standard Progression Algorithm

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

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

---

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

\(\text{pick-up}(T_2, C, P_2) \prec \text{get-to}(T_2, D) \prec \text{drop}(T_2, D, P_2)\)
### Standard Progression Algorithm

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

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

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

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

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

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

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

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

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

\[\text{else}\]

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

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

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

\[\pi = (\text{no-op}(\))\]
1. \( \text{fringe} \leftarrow \{(s_1, t_{n_1}, ())\} \)
2. while \( \text{fringe} \neq \emptyset \) do
3. \( n \leftarrow \text{fringe.poll()} \)
4. if \( n.\text{isgoal} \) 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\( (n') \)

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

**Heuristics**

**Excursion**

---

**HTN Progression Search**

**Standard Progression Algorithm**

---

```plaintext
 fringe ← \{(s_i, t_{n_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')
```

---

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

---

Diagram:

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

---

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

June 25th, ICAPS 2018 (Delft)

---

```
Solving Techniques

HTN Progression Search

Standard Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s_1, t_1, ())\} \)
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. \( \text{fringe.add}(n') \)
10. else
11. for \( m \in t.\text{methods} \) do
12. \( n' \leftarrow n.\text{decompose}(t, m) \)
13. \( \text{fringe.add}(n') \)

\[ \pi = (\text{no-op()}, \text{no-op()}, \text{pick-up}(T_1, A, P_1)) \]

\( \text{get-to}(T_1, B) \leadsto \text{drop}(T_1, B, P_1) \)

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

Standard Progression Algorithm

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 = (no-op(), no-op(), pick-up(T1, A, P1))
HTN Progression Search

Standard Progression Algorithm

\[ fringe \leftarrow \{(s_1, t_1, ())\} \]

while \( fringe \neq \emptyset \) do

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

if \( n.isgoal \) then return \( n \)

\[ U \leftarrow n.unconstrainedNodes \]

for \( t \in U \) do

if \( isPrimitive(t) \) then

\[ n' \leftarrow n.apply(t) \]

fringe.add(\( n' \))

else

for \( m \in t.methods \) do

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

fringe.add(\( n' \))

\[ \pi = (no-op(), no-op(), pick-up(T_1, A, P_1)) \]
**HTN Progression Search**

**Standard Progression Algorithm**

1. $\text{fringe} \leftarrow \{(s_1, \text{tn}_1, ())\}$
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. $\text{fringe.add}(n')$
      10. else
         11. for $m \in t.$methods do
             12. $n' \leftarrow n.$decompose$(t, m)$
             13. $\text{fringe.add}(n')$

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

Heuristics

Excursion

HTN Progression Search

Standard Progression Algorithm

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

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

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

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

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

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

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

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

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

\[
\text{else}
\]

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

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

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

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

\[
\text{drop}(T_1, B, P_1)
\]

\[
\text{get-to}(T_2, D) \prec \cdots \text{drop}(T_2, D, P_2)
\]

\[
P_1 \quad A \quad T_1 \quad D
\]

\[
P_2 \quad C \quad T_2
\]
**Standard Progression Algorithm**

1. $\text{fringe} \leftarrow \{(s_1, t_{n_1}, (\))\}$
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 $\text{isPrimitive}(t)$ then
         8. $n' \leftarrow n.$apply$(t)$
         9. $\text{fringe.add}(n')$
      else
         10. for $m \in t.$methods do
             11. $n' \leftarrow n.$decompose$(t, m)$
             12. $\text{fringe.add}(n')$

**HTN Progression Search Example**

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

**Heuristics**

**Excursion**

---

**HTN Progression Search**

**Standard Progression Algorithm**

```
1 fringe ← {(s₀, t₀, ( ))}
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′)
14
π = (no-op(), no-op(), pick-up(T₁, A, P₁), drive(T₁, A, B),
     pick-up(T₂, C, P₂))
```

---

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

June 25th, ICAPS 2018 (Delft)
HTN Progression Search

Standard Progression Algorithm

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

\[
\pi = (\text{no-op()}, \text{no-op()}, \text{pick-up}(T_1, A, P_1), \text{drive}(T_1, A, B), \\
\text{pick-up}(T_2, C, P_2), \text{drive}(T_2, A, B))
\]
1. \( \text{fringe} \leftarrow \{(s, tn, ())\} \)
2. \( \textbf{while} \ \text{fringe} \neq \emptyset \ \textbf{do} \)
3. \( n \leftarrow \text{fringe.poll()} \)
4. \( \textbf{if} \ n.\text{isgoal} \textbf{then} \ \text{return} n \)
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. \( \text{fringe.add}(n') \)

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

1. fringe ← \{(s, t, n, ())\}
2. 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), drive(T_1, A, B),
     pick-up(T_2, C, P_2), drive(T_2, A, B), drop(T_2, D, P_2),
     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 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 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.

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

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

Unconstrained tasks are split into compound and primitive tasks

```
fringe ← {(s₀, tn₁, ())}
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′)
```
HTN Progression Search

Improved Progression Algorithm

1. $\text{fringe} \leftarrow \{(s_0, t_1, ())\}$
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. $\text{fringe.add}(n')$

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

1. \( \text{fringe} \leftarrow \{(s_0, t_{n1}, ())\} \)
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. \( \text{fringe.add}(n') \)
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. \( \text{fringe.add}(n') \)

- **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 ← \{(s_0, tn_I, ())\}`
2. **while** `fringe ≠ ∅` **do**
3. \( n ← \text{fringe.poll}() \)
4. \( \text{if } n.\text{isgoal} \text{ then return } n \)
5. \((U_C, U_P) ← n.\text{unconstrainedNodes} \)
6. **for** \( t ∈ U_P \) **do**
7. \( n' ← n.\text{apply}(t) \)
8. \( \text{fringe.add}(n') \)
9. \( t ← \text{selectAbstractTask}(U_C) \)
10. **for** \( m ∈ t.\text{methods} \) **do**
11. \( n' ← n.\text{decompose}(t, m) \)
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
- **Method application** is done 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 \( n.\text{isgoal} \) 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. \( \text{fringe.add}(n') \)

\[ \pi = () \]
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. $\text{fringe.add}(n')$
   9. $t \leftarrow \text{selectAbstractTask}(U_C)$
  10. for $m \in t.\text{methods}$ do
      11. $n' \leftarrow n.\text{decompose}(t, m)$
      12. $\text{fringe.add}(n')$

$\pi = ()$
Improved Progression Algorithm

1. fringe ← \{ (s_0, t_{n_i}, ()) \}
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 = ()

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

deliver(P_2, D)
1 \( \text{fringe} \leftarrow \{ (s_0, tn_1, ()) \} \)

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

3 \hspace{1em} n \leftarrow \text{fringe.poll()} \)

4 \hspace{1em} \text{if } n.\text{isgoal} \ \text{then return } n \)

5 \hspace{1em} (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \)

6 \hspace{1em} \text{for } t \in U_P \ \text{do} \)

7 \hspace{2em} n' \leftarrow n.\text{apply}(t) \)

8 \hspace{2em} \text{fringe.add}(n') \)

9 \hspace{1em} t \leftarrow \text{selectAbstractTask}(U_C) \)

10 \hspace{1em} \text{for } m \in t.\text{methods} \ \text{do} \)

11 \hspace{2em} n' \leftarrow n.\text{decompose}(t, m) \)

12 \hspace{2em} \text{fringe.add}(n') \)

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

1. \( fringe \leftarrow \{(s_0, tn_1, ())\} \)
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.   \( fringe.add(n') \)

\( \pi = () \)

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

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

\[ \text{no-op}() \]
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 $n$.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. $t \leftarrow \text{selectAbstractTask}(U_C)$
   10. for $m \in t.methods$ do
       11. $n' \leftarrow n.decompose(t, m)$
       12. fringe.add($n'$)

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

\begin{algorithm}
\begin{algorithmic}
\State $\text{fringe} \leftarrow \{ (s_0, t_{n_1}, () ) \}$
\While{$\text{fringe} \neq \emptyset$}
\State $n \leftarrow \text{fringe.poll()}$
\If{$n$ is goal} \State \textbf{return} $n$
\EndIf
\State ($U_C$, $U_P$) $\leftarrow n.unconstrainedNodes$
\For{$t \in U_P$}
\State $n' \leftarrow n.apply(t)$
\State $\text{fringe.add}(n')$
\EndFor
\State $t \leftarrow \text{selectAbstractTask}(U_C)$
\For{$m \in t.methods$}
\State $n' \leftarrow n.decompose(t, m)$
\State $\text{fringe.add}(n')$
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

$\pi = ()$
1. \( \text{fringe} \leftarrow \{(s_0, t_{ni}, (\))\} \)
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. \( \text{fringe}.\text{add}(n') \)
   9. \( t \leftarrow \text{selectAbstractTask}(U_C) \)
   10. for \( m \in t.\text{methods} \) do
       11. \( n' \leftarrow n.\text{decompose}(t, m) \)
       12. \( \text{fringe}.\text{add}(n') \)

\( \pi = () \)
fringe ← \{(s_0, t_{n_1}, ())\}

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')

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

1. \(\text{fringe} \leftarrow \{(s_0, t_{n1}, (\_))\}\)
2. \(\text{while fringe} \neq \emptyset \text{ do}\)
   3. \(n \leftarrow \text{fringe.poll()}\)
   4. \(\text{if } n\text{.isgoal then return } n\)
   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. \(\text{fringe.add}(n')\)

\[
\pi = ()
\]
```
fringe ← \{(s_0, t_{n_1}, ( ))\}
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')
    π = ( )
```
Improved Progression Algorithm

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

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

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

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

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

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

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

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

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

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

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

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

\[ \pi = (\text{no-op}()) \]
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.     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($n'$)

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

\[ \begin{align*}
\text{fringe} & \leftarrow \{(s_0, tn_1, ())\} \\
\text{while} \ \text{fringe} \neq \emptyset & \text{ do} \\
& n \leftarrow \text{fringe.poll()} \\
& \text{if} \ n.\text{isgoal} \ \text{then} \ \text{return} \ n \\
& (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \\
& \text{for} \ t \in U_P \ \text{do} \\
& \quad n' \leftarrow n.\text{apply}(t) \\
& \quad \text{fringe}.\text{add}(n') \\
& t \leftarrow \text{selectAbstractTask}(U_C) \\
& \text{for} \ m \in t.\text{methods} \ \text{do} \\
& \quad n' \leftarrow n.\text{decompose}(t, m) \\
& \quad \text{fringe}.\text{add}(n') \\
\end{align*} \]

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

Heuristics

Excursion

HTN Progression Search

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

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

\[
\begin{align*}
\text{ fringe } & \leftarrow \{ (s_0, t_{n_1}, ()) \} \\
\text{ while fringe } & \neq \emptyset \text{ do} \\
& \quad n \leftarrow \text{ fringe.poll()} \\
& \quad \text{ if } n.\text{isgoal} \text{ then return } n \\
& \quad (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \\
& \text{ for } t \in U_P \text{ do} \\
& \quad n' \leftarrow n.\text{apply}(t) \\
& \quad \text{ fringe.add}(n') \\
& \quad t \leftarrow \text{ selectAbstractTask}(U_C) \\
& \text{ for } m \in t.\text{methods} \text{ do} \\
& \quad n' \leftarrow n.\text{decompose}(t, m) \\
& \quad \text{ fringe.add}(n')
\end{align*}
\]

\[\pi = (\text{no-op()}, \text{pick-up}(T_2, C, P_2))\]
Improving the Progression Algorithm

1. \( \text{fringe} \leftarrow \{(s_0, t_1, ())\} \)
2. \( \text{while fringe} \neq \emptyset \) do
   - \( n \leftarrow \text{fringe.poll()} \)
   - if \( n.\text{isgoal} \) then return \( n \)
   - \( (U_C, U_P) \leftarrow n.\text{unconstrainedNodes} \)
   - for \( t \in U_P \) do
     - \( n' \leftarrow n.\text{apply}(t) \)
     - fringe.add(\( n' \))
   - \( t \leftarrow \text{selectAbstractTask}(U_C) \)
   - for \( m \in t.\text{methods} \) do
     - \( n' \leftarrow n.\text{decompose}(t, m) \)
     - fringe.add(\( n' \))

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

1. fringe ← \{(s_0, tn_i, ())\}
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')

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

```plaintext
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')
13
π = (no-op(), pick-up(T₂, C, P₂), drive(T₂, A, B), no-op())
```

```
pick-up(T₁, A, P₁) ... get-to(T₁, B) ... drop(T₁, B, P₁)
drop(T₂, D, P₂)
```
Improved Progression Algorithm

```
fringe ← \{ (s₀, tn₁, () ) \}
while fringe ≠ ∅ do
  n ← fringe.poll()
  if n.isgoal then return n
  \( (UC, UP) \leftarrow n.unconstrainedNodes \)
  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')

\( \pi = (\text{no-op()}, \text{pick-up}(T₂, C, P₂), \text{drive}(T₂, A, B), \text{no-op()}, \text{pick-up}(T₁, A, P₁)) \)
```

Diagram:
- \( get-to(T₁, B) \)
- \( drop(T₁, B, P₁) \)
- \( drop(T₂, D, P₂) \)

```
B P₂ C T₂

| P₁ | A | T₁ | D |
```
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(), pick-up(T₂, C, P₂), drive(T₂, A, B), no-op(),
pick-up(T₁, A, P₁), drop(T₂, D, P₂))
```

```
get-to(T₁, B) ≪ drop(T₁, B, P₁)
```

- `get-to(T₁, B)`
- `drop(T₁, B, P₁)`
Improved Progression Algorithm

\[ \pi = (\text{no-op}(), \text{pick-up}(T_2, C, P_2), \text{drive}(T_2, A, B), \text{no-op}(), \text{pick-up}(T_1, A, P_1), \text{drop}(T_2, D, P_2)) \]
1 \( fringe \leftarrow \{(s_0, tn, ())\} \)
2 \( \textbf{while } fringe \neq \emptyset \textbf{ do} \)
3 \( n \leftarrow fringe.poll() \)
4 \( \textbf{if } n.isgoal \textbf{ then return } n \)
5 \((U_C, U_P) \leftarrow n.unconstrainedNodes \)
6 \( \textbf{for } t \in U_P \textbf{ do} \)
7 \( n' \leftarrow n.apply(t) \)
8 \( fringe.add(n') \)
9 \( t \leftarrow \text{selectAbstractTask}(U_C) \)
10 \( \textbf{for } m \in t.methods \textbf{ do} \)
11 \( n' \leftarrow n.decompose(t, m) \)
12 \( fringe.add(n') \)

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

\[ fringe \leftarrow \{ (s_0, tn_1, () ) \} \]

\[ while \ fringe \neq \emptyset do \]

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

\[ if \ n.isgoal \ then \ return \ n \]

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

\[ for \ t \in U_P \ do \]

\[ n' \leftarrow n.apply(t) \]

\[ fringe.add(n') \]

\[ t \leftarrow selectAbstractTask(U_C) \]

\[ for \ m \in t.methods \ do \]

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

\[ fringe.add(n') \]

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

```
fringe ← \{ (s₀, t₀, () ) \}
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′)

π = (no-op(), pick-up(T₂, C, P₂), drive(T₂, A, B), no-op(),
pick-up(T₁, A, P₁), drop(T₂, D, P₂), drive(T₁, A, B), drop(T₁, B, P₁))
```
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 extend) 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

\[
\begin{align*}
\text{pick-up}(T_1, A, P_1) & \leadsto \text{get-to}(T_1, B) & \leadsto \text{drop}(T_1, B, P_1) \\
\text{deliver}(P_2, D)
\end{align*}
\]

- 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

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

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

**orderings:**
- \( before(t_2, t_3) \), \( before(t_3, t_5) \)
Compilation to STRIPS/ADL

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

\[
\text{pick-up}(T_1, A, P_1) \leftarrow \ldots \text{get-to}(T_1, B) \leftarrow \ldots \text{drop}(T_1, B, P_1) \\
\text{deliver}(P_2, D)
\]

\[ \text{pick-up}(T_1, A, P_1, t_2) \]

\textbf{pre} : prec from domain,

\[ \forall t_i \in \{t_0 \ldots t_b\} : \neg \text{before}(t_i, t_2) \]

\textbf{eff} : effects from domain,

\[ \neg \text{pick-up}(T_1, A, P_1, t_2), \text{free}(t_2) \]

\[ \forall t_i \in \{t_0 \ldots t_b\} : \neg \text{before}(t_2, t_i) \]

[pick-up(\(T_1, A, P_1, t_2\)),
geto(\(T_1, B\)),
drop(\(T_1, B, P_1\)),
deliver(\(P_2, D\))

\textbf{ppick-up}(T_1, A, P_1, t_2),
\textbf{pget-to}(T_1, B, t_3),
\textbf{pdrop}(T_1, B, P_1, t_5),
\textbf{pdeliver}(P_2, D, t_4),
\textbf{before}(t_2, t_3), \textbf{before}(t_3, t_5)\]
### Solve Techniques

#### Heuristics

#### Excursion

---

### Compilation to STRIPS/ADL

#### 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**

---

**Heuristics**

**Excursion**

---

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

June 25th, ICAPS 2018 (Delft) 22 / 62
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)
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
## 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
  - 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
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
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)
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)

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

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

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

$c_I \rightarrow ABC$ and $c_I \rightarrow ACp$ 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_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

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

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

\[ c_I \rightarrow ABC \text{ and } c_I \rightarrow ACp \text{ and } c_I \rightarrow Ar \]
\[ \{A\} \; \{B, C\} \; \{C, p, r\} \]
Also possible: \( \{A\} \; \{A, B, C\} \; \{C, p, r\} \)
**All** Decomposition Trees can not be represented

⇒ bound the height of represented trees

\[
\begin{align*}
c_l & \rightarrow ABC \quad \text{and} \quad c_l \rightarrow ACp \quad \text{and} \quad c_l \rightarrow Ar \\
\{ A \} & \quad \{ B, C \} \quad \{ C, p, r \} \\
\text{Also possible:} \quad \{ A \} & \quad \{ A, B, C \} \quad \{ C, p, r \} \\
\text{Also possible:} \quad \{ A \} & \quad \{ B \} \quad \{ C \} \quad \{ p, r \}
\end{align*}
\]
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 leafs?
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 leaves?
  - Fewer tasks per leaf?
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 leaves?
  - Fewer tasks per leaf?
  - Fewer tasks per inner node?
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!
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!

Current work tries greedily to put as few tasks as possible to each child.
A PDT contains every Decomposition Tree of height $\leq K$ as a sub-graph.
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
Possible Heuristic Estimates

What do we want to estimate?

- Number of missing actions (or their costs, resp.) or
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)
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
## Problem Relaxations for Heuristic Calculation

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

---

<table>
<thead>
<tr>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition Graph-based Heuristics</td>
</tr>
<tr>
<td>Problem Relaxations for Heuristic Calculation</td>
</tr>
</tbody>
</table>

- Perform task insertion
- Perform delete relaxation

→ This makes the (TI)HTN plan existence problem decidable in P
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 \( \mathbf{P} \)
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 $G = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

- $N_T$, the task nodes,
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:

\[
\begin{align*}
    h_T(n_t) &:= \\
           &\begin{cases}
                \text{cost}(n_t) & \text{if } n_t \text{ primitive} \\
                \min_{(n_t, n_m) \in E_{T \rightarrow M}} h_M(n_m) & \text{else}
            \end{cases}
\end{align*}
\]
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 \( n_m = \langle T, \prec, \alpha \rangle \):

\[
h_M(n_m) := \sum_{(n_m,n_t)\in E_{M\rightarrow T}} h_T(n_t)
\]
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 \( n_m = \langle T, \prec, \alpha \rangle \):

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

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.
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:**

\[ h_T(t_0) = \min \{ h_M(m_1), h_M(m_2) \} \]
A TDG represents the decomposition structure:

**Example:**
Method \( m_1 = (t_0, tn) \) with task network \( tn \):

\[
\begin{align*}
h_M(m_1) &= \sum_{t_i \in \{t_1, t_2, t_3\}} h_T(t_i) \\
&= h_T(t_1) + \text{cost}(t_2) + h_T(t_3)
\end{align*}
\]
Decomposition Graph-based Heuristics

Cost-aware heuristic TDG-c (Example)

A TDG represents the decomposition structure:

![Decomposition Graph](image)

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 $tn$:

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

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

$$= \text{cost}(t_5) + \text{cost}(t_6)$$
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:

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

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

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

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

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 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** ($\rightarrow$ Satisficing Planning)

**Using Techniques from Classical Planning – Challenges:**

- More expressive formalism $\rightarrow$ 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
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
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
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
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

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

drive(T, B, C)

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

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

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

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

Diagram:
- A ➔ B ➔ D
- A ➔ B ➔ C ➔ D
- A ➔ C ➔ D

Nodes:
- A
- B
- C
- D
- T

Actions:
- drive(T, A, B)
- drive(T, B, C)
- drive(T, B, D)
- m-direct(T, A, B)
- m-direct(T, C, B)
- m-deliver(P, C, D, T)
- get-to(T, C)
- m-via(T, B, C)
- get-to(T, B)
- pick-up(T, C, P)
- get-to(T, D)
- m-via(T, B, D)
- drop(T, D, P)
- deliver(P, 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

- begin
  - 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\((T, A, B)\)
- drive\((T, C, B)\)
Using Classical Heuristics to Guide HTN Search

Simulating Composition

\[
\text{drive}(T, A, B) \\
\text{get-to}(T, B) \\
\text{m-direct}(T, A, B) \\
\text{get-to}(T, C) \\
\text{m-via}(T, B, C) \\
\text{get-to}(T, D) \\
\text{m-via}(T, B, D) \\
deli 1 (P, D) \\
m-deliver(P, C, D, T) \\
pick-up(T, C, P) \\
get-to(T, D) \\
m-via(T, B, D) \\
drop(T, D, P) \\
drive(T, B, C) \\
drive(T, C, B) \\
drive(T, B, D)
\]
Using Classical Heuristics to Guide HTN Search

Simulating Composition

1. get \( (P, D) \)
2. m-deliver \((P, C, D, T)\)
3. pick-up \((C, P)\)
4. get-to \((T, D)\)
5. m-via \((T, B, D)\)
6. drop \((T, D, P)\)
7. m-via \((T, B, C)\)
8. drive \((T, B, C)\)
9. get-to \((T, B)\)
10. m-direct \((T, A, B)\)
11. drive \((T, A, B)\)

A

B

C

D

E
Simulating Composition

Using Classical Heuristics to Guide HTN Search

- Heuristics
- Excursion

### Solving Techniques

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

The goal is to reach the current task network.

- **deliver:** \((P, D)\)
- **m-deliver:** \((P, C, D, T)\)

- **get:** \((T, C)\)
- **m-via:** \((T, B, C)\)

- **pick-up:** \((C, P)\)

- **get:** \((T, D)\)
  - **m-via:** \((T, B, D)\)

- **drop:** \((D, P)\)

- **get:** \((T, C)\)
  - **m-direct:** \((T, A, B)\)

- **drive:** \((B, C)\)
- **drive:** \((T, C, B)\)

- **drive:** \((B, D)\)

- **drive:** \((A, B)\)

- **drive:** \((C, B)\)
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

```
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, C, B)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
drive(T, B, D)
Introduce new state features

- 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

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

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

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

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

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

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

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

\[
\text{drop}(T, D, P)
\]
Introduce new state features

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

Diagram:

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

Equations:

- \( \text{at}(v, l) \)
- \( \text{road}(l, l_1, l_2) \)
- \( \text{at}(v, l_2) \)
- \( \text{at}(v, l_1) \)
- \( \text{drive}(v, l_1, l_2) \)
- \( \text{at}(v, l_1) \)
- \( \text{at}(v, l_2) \)
- \( \text{pick-up}(v, l, p) \)
- \( \text{at}(p, l) \)
- \( \text{in}(p, v) \)
- \( \text{at}(p, l) \)
- \( \text{in}(p, v) \)
- \( \text{drop}(v, l, p) \)
- \( \text{at}(p, l) \)
- \( \text{in}(p, v) \)
Introduce new state features

- Modify actions

Using Classical Heuristics to Guide HTN Search

Simulating Composition
Introduce new state features

Modify actions
Solving Techniques

Simulating Composition

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

**Introduce new state features**

- **Modify actions**

**Introduce new action for every method**

![Diagram of HTN Planning](attachment:image.png)

- **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, B, D)
- **drive**(v, l, l)
- **m-direct**(v, l, l)
- **get-to**(v, l2)
- **drive**(v, l, l2)

**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

Using Classical Heuristics to Guide HTN Search
Simulating Composition
Introduce new state features
Modify actions
Introduce new action for every method
Introduce new state features
Modify actions
Introduce new action for every method

Using Classical Heuristics to Guide HTN Search
Simulating Composition

- 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

Heuristics

- Introduce new state features
- Modify actions
- Introduce new action for every method
- Goal is to reach current task network
Using Classical Heuristics to Guide HTN Search

Simulating Composition – Resulting Model

- Drive: \( \text{drive}(v, l_1, l_2) \)
- Road: \( \text{road}(l_1, l_2) \)
- At: \( \text{at}(v, l) \)
- No-op: \( \text{no-op}() \)
- Pick-up: \( \text{pick-up}(v, l, p) \)
- Drop: \( \text{drop}(v, l, p) \)
- Deliver: \( \text{am-deliver}(p, l_1, l_2, v) \)
- Direct: \( \text{am-direct}(v, l_1, l_2) \)
- Via: \( \text{am-via}(v, l_1, l_2) \)
- Get-to: \( \text{b-get-to}(v, l_1) \)
- No-op: \( \text{b-no-op}() \)

Formulas:
- \( \text{b-} \text{drive}(v, l_1, l_2) \) from \( \text{drive}(v, l_1, l_2) \)
- \( \text{b-} \text{no-op}() \) from \( \text{no-op}() \)
- \( \text{b-} \text{pick-up}(v, l, p) \) from \( \text{pick-up}(v, l, p) \)
- \( \text{b-} \text{drop}(v, l, p) \) from \( \text{drop}(v, l, p) \)
- \( \text{b-} \text{deliver}(p, l_1, l_2, v) \) from \( \text{am-deliver}(p, l_1, l_2, v) \)
- \( \text{b-} \text{get-to}(v, l_1) \) from \( \text{get-to}(v, l_1) \)
- \( \text{b-} \text{get-to}(v, l_2) \) from \( \text{get-to}(v, l_2) \)

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

June 25th, ICAPS 2018 (Delft)
Planning in the Transformed Model

\[
drive(T, A) \rightarrow m-direct(T, A, B) \rightarrow drive(T, A, B)
\]

\[
drive(T, B, C) \rightarrow m-direct(T, C, B) \rightarrow drive(T, C, B)\]

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

\[
drive(T, B, C) \rightarrow m-direct(T, A, B) \rightarrow drive(T, B, C)
\]

\[
drive(T, B) \rightarrow m-direct(T, B, C) \rightarrow drive(T, B, C)
\]

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

\[
\{\text{at}(T, A), \text{at}(P, C)\}
\]
Planning in the Transformed Model

Heuristic value: 10
Planning in the Transformed Model

Heuristic value: 10
Planning in the Transformed Model

- \( \text{get-to}(T, C) \)
- \( m\text{-via}(T, B, C) \)
- \( \text{drive}(T, B, C) \)
- \( \text{drive}(T, A, B) \)

\( \text{get-to}(T, B) \)
\( m\text{-direct}(T, A, B) \)
\( \text{drive}(T, A, B) \)

- \( \text{delivered}(P, D) \)
- \( m\text{-deliver}(P, C, D, T) \)

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

\( \text{get-to}(T, C) \)
\( \text{drive}(T, C, B) \)
\( \text{drive}(T, B, D) \)

Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Planning in the Transformed Model

Heuristic value: 10
Planning in the Transformed Model

Heuristic value: 10

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

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

Heuristic value: 10
Planning in the Transformed Model

Using Classical Heuristics to Guide HTN Search

Heuristic value: 10
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

\[
\begin{align*}
&\text{pick-up}(T, C, P) \quad \text{get-to}(T, D) \quad \text{drop}(T, D, P) \\
&\text{get-to}(T, B) \quad \text{drive}(T, B, C) \quad \text{drive}(T, B, D) \\
&\text{get-to}(T, C) \quad \text{m-via}(T, B, C) \quad \text{m-via}(T, B, D) \\
&\text{drive}(T, A, B) \quad \text{drive}(T, C, B) \\
\end{align*}
\]

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

Using delete-relaxed classical heuristic: 9
Solving Techniques

Heuristics

Excursion

Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

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

\{ \text{at}(T, A), \text{at}(P, C) \} \quad \{ \text{at}(T, A), \text{at}(T, B), \text{at}(P, C), b\text{-drive}(T, A, B) \} \
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

\[
\begin{align*}
&\text{deliver}(P,D) \\
&m\text{-deliver}(P,C,D,T) \\
&\quad \downarrow \\
&\text{get-to}(T,C) \\
&m\text{-via}(T,B,C) \\
&\quad \downarrow \\
&\text{get-to}(T,B) \\
&m\text{-direct}(T,A,B) \\
&\quad \downarrow \\
&\text{drive}(T,A,B) \\
&\quad \downarrow \\
&\{\text{at}(T,A), \text{at}(P,C)\} \quad \text{drive}(T,A,B) \quad \{\text{at}(T,A), \text{at}(T,B), \text{at}(P,C), \text{at}(P,C), \text{at}(T,B), \text{at}(P,C)\} \quad \text{am\text{-}direct}(T,A,B) \quad \ldots
\end{align*}
\]
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

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

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

{\text{at}(T, A), \text{at}(T, B), \text{at}(P, C), \text{b-drive}(T, A, B)}
\text{am-direct}(T, A, B)
\text{drive}(T, B, C)
\text{...}
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

\[
\begin{align*}
\text{deliver}(P, D) & \quad \text{m-deliver}(P, C, D, T) \\
\text{get-to}(T, C) & \quad \text{pick-up}(T, C, P) \\
\text{m-via}(T, B, C) & \quad \text{get-to}(T, D) \\
\text{m-via}(T, B, D) & \\
\text{get-to}(T, B) & \quad \text{drive}(T, B, C) \\
\text{m-direct}(T, A, B) & \\
\text{drive}(T, A, B) & \\
\{\text{at}(T, A), \text{at}(P, C)\} & \\
\text{drive}(T, A, B) & \\
\{\text{at}(T, A), \text{at}(T, B), \text{at}(P, C), \text{at}(P, C), b\text{-drive}(T, A, B)\} & \quad \text{am-direct}(T, A, B) & \quad \cdots & \quad \text{drive}(T, B, C) & \quad \cdots & \quad \text{pick-up}(T, C, P) & \quad \cdots
\end{align*}
\]
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)
Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)

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

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

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

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

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

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

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

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

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

\[
\text{drive}(T, B, C) \quad \text{pick-up}(T, C, P) \quad \text{am\text{-}via}(T, B, C)
\]

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

\[
\text{...}
\]
Using Classical Heuristics to Guide HTN Search

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)**
- **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, B, D)**
- **drive(T, A, B)**
- **drive(T, C, B)**

- **at(T, A)**,
- **at(P, C)**
- **drive(T, A, B)**
- **am-direct(T, A, B)**
- **drive(T, B, C)**
- **pick-up(T, C, P)**
- **am-via(T, B, C)**

- **drive(T, B, D)**
- **drop(T, D, P)**

Heuristics Calculation (Delete Relaxed)
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)

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
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

- Heuristic function may only insert tasks that lie within the decomposition hierarchy (not given here)
Size is **linear** in the input HTN domain, but the model is **large**

**State** and **action** set are **extended**
- 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
Using Classical Heuristics to Guide HTN Search

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**.
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
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?
Overview of Hierarchical Planning Variants

Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN 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
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 $\mathcal{P}^* = (V, P, \delta, C, M, s_l, c_l)$ 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 \( P^* = (V, P, \delta, C, M, s_I, c_I) \) 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 $\bar{m}$ 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 $\bar{t}$ 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 $\mathcal{P}^* = (V, P, \delta, C, M, s_l, c_l)$ 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

```
connect(DVD-player, Adapter) ←→ connect(Adapter, TV)
connect(Blu-ray-player, Adapter) ←→ connect(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
**Hybrid planning** fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.
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**

**Problem Definition**

*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.

**Diagram:**

- **pick** (?obj, ?from)
  - at(?obj, ?from)
  - handopen

- **move** (?obj, ?from, ?to)
  - at(?obj, ?from)
  - handopen

- **place** (?obj, ?to)
  - handopen
  - at(?obj, ?to)
  - holding(?obj)

- **Example:**
  - handopen
  - at(?obj, ?from)
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.
Decompositional 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 *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*
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?