Bound to Plan: Exploiting Classical Heuristics via Automatic Translations of Tail-Recursive HTN Problems

Ron Alford\textsuperscript{1}  Gregor Behnke\textsuperscript{2}  Daniel Höller\textsuperscript{2}  Pascal Bercher\textsuperscript{2}  Susanne Biundo\textsuperscript{2}  David W. Aha\textsuperscript{3}

\textsuperscript{1}ASEE/NRL Postdoctoral Fellow (now at MITRE)
ralford@mitre.org

\textsuperscript{2}Institute of Artificial Intelligence; Ulm University
first.last@uni-ulm.de

\textsuperscript{3}Navy Center for Applied Research in AI; Naval Research Laboratory
david.aha@nrl.navy.mil

June 17th, 2016
Hierarchical Task Network (HTN) Planning

HTN planning is the problem of decomposing an initial task to accomplish into a sequence of executable steps.

- Encoding control knowledge for planning
  - Agent planning in robotics and simulation: ARTUE (2010), Johnny (2012)
  - Planning and mission generation for games: KILLZONE 2, Elder Scrolls, Armed Assault

- Representing and planning with processes and hierarchies
  - Composing web services (Sirin, 2004; Tang 2013; others)
  - Program configuration (Soltani, 2012)
Towards heuristic HTN planning

Combining HTN and heuristic search
- HTNs are used to prune the search space until the problem is solvable
- Heuristics guide the search towards good solutions

First cut solution: Translate HTN problems into classical problems
- Many domain independent heuristics for classical (PDDL) planning
- Translation allows us to directly use these heuristics for HTN planning
Towards heuristic HTN planning

Combining HTN and heuristic search
- HTNs are used to prune the search space until the problem is solvable
- Heuristics guide the search towards good solutions

First cut solution: Translate HTN problems into classical problems
- Many domain independent heuristics for classical (PDDL) planning
- Translation allows us to directly use these heuristics for HTN planning
- Con: Can’t translate everything (HTN planning is too expressive)
HTN Planning (Overview)

The purpose of HTN planning is to complete a task. Tasks are either:

- **Primitive**, which corresponds to some concrete action we know how to perform, e.g., `walk(room, hall)`, or `drink(coffee)`
- **Non-primitive**, which is an abstract task. E.g. `travel(home, ICAPS)`
- Must recursively decompose non-primitive tasks until we get primitive tasks we know how to execute directly
- We are given a set of methods, which are recipes on how to accomplish abstract tasks. E.g., to travel from `home` to `ICAPS`, we might decompose as follows:

```
travel(h, ICAPS)
```

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buy_ticket</td>
<td></td>
</tr>
<tr>
<td>travel(h, a1)</td>
<td></td>
</tr>
<tr>
<td>fly(a1, a2)</td>
<td></td>
</tr>
<tr>
<td>travel(a2, ICAPS)</td>
<td></td>
</tr>
<tr>
<td>travel(h, stop1)</td>
<td></td>
</tr>
<tr>
<td>bus(stop1, a1)</td>
<td></td>
</tr>
<tr>
<td>taxi(a2, ICAPS)</td>
<td></td>
</tr>
</tbody>
</table>
```
Methods and Decomposition

- A method \((t, tn)\) is a non-primitive task \(t\) paired with a network \(tn\)

Method:

```
train(x, y)  
  ticket(a1, a2)  
    train(x, a1)  
      fly(a1, a2)  
    train(a2, y)  
```

Method:

```
train(x, y)  
  taxi(x, y)  
  train(x, y)  
    walk(x, y)  
```

- We decompose a task network by replacing a node in the network with a corresponding method’s network.
**Methods and Decomposition**

- A method \((t, tn)\) is a non-primitive task \(t\) paired with a network \(tn\)

We decompose a task network by replacing a node in the network with a corresponding method’s network.
A method \((t, tn)\) is a non-primitive task \(t\) paired with a network \(tn\).

We decompose a task network by replacing a node in the network with a corresponding method’s network.
Methods and Decomposition

- A method \((t, tn)\) is a non-primitive task \(t\) paired with a network \(tn\)

Method:

```
Method:

\text{fly}(\text{a}_1, \text{a}_2) \quad \text{ticket}(\text{a}_1, \text{a}_2) \quad \text{travel}(\text{a}_1, \text{a}_2) \quad \text{travel}(\text{a}_2, \text{y})
```

Method:

```
Method:

\text{travel}(\text{x}, \text{y}) \quad \text{taxi}(\text{x}, \text{y}) \quad \text{travel}(\text{x}, \text{y}) \quad \text{walk}(\text{x}, \text{y})
```

- We decompose a task network by replacing a node in the network with a corresponding method’s network.
HTN Progression

A number of HTN planners (SHOP, SHOP2, etc) plan for tasks in the order that they are to be executed. This is called progression.

For an HTN problem \( P = (D, s_0, tn_0) \), we progress \( P \) by picking a task \( t \) in \( tn_0 \) with no predecessors. Then:

- If \( t \) is non-primitive:
  - Decompose \( t \) in \( tn_0 \), let \( tn' \) be the result
  - Then \( (D, s_0, tn') \) is a progression of \( P \).
- If \( t \) is primitive (\( t \in O \)) and executable, then:
  - Apply \( t \): \( \gamma(s_0, t) = s_1 \)
  - Remove it from \( tn_0 \): \( tn' = tn_0 \setminus \{t\} \)
  - Then \( (D, s_1, tn') \) is a progression of \( P \).
HTN Progression

A number of HTN planners (SHOP, SHOP2, etc) plan for tasks in the order that they are to be executed. This is called progression. For an HTN problem $P = (D, s_0, tn_0)$, we progress $P$ by picking a task $t$ in $tn_0$ with no predecessors. Then:

- If $t$ is non-primitive:
  - Decompose $t$ in $tn_0$, let $tn'$ be the result
  - Then $(D, s_0, tn')$ is a progression of $P$.

- If $t$ is primitive ($t \in O$) and executable, then:
  - Apply $t$: $\gamma(s_0, t) = s_1$
  - Remove it from $tn_0$: $tn' = tn_0 \setminus \{t\}$
  - Then $(D, s_1, tn')$ is a progression of $P$. 
HTN Progression

A number of HTN planners (SHOP, SHOP2, etc) plan for tasks in the order that they are to be executed. This is called progression. For an HTN problem $P = (D, s_0, tn_0)$, we progress $P$ by picking a task $t$ in $tn_0$ with no predecessors. Then:

- If $t$ is non-primitive:
  - Decompose $t$ in $tn_0$, let $tn'$ be the result
  - Then $(D, s_0, tn')$ is a progression of $P$.

- If $t$ is primitive ($t \in O$) and executable, then:
  - Apply $t$: $\gamma(s_0, t) = s_1$
  - Remove it from $tn_0$: $tn' = tn_0 \setminus \{t\}$
  - Then $(D, s_1, tn')$ is a progression of $P$. 

```
work
state: thirsty

drink
work
write
state: thirsty

write
work
state: ¬thirsty
```
Translating Task Networks into PDDL

- $ticket(IAD, LHR)$
- $travel(home, IAD)$
- $fly(IAD, LHR)$
- $travel(LHR, ICAPS)$

- $t_1$, $t_2$, $t_3$, $t_4$, ... , $t_n$  

} Insert set of task identifier constants
Translating Task Networks into PDDL

- \( t_1, t_2, t_3, t_4, \ldots, t_n \)
- \( \text{ticket}(IAD, LHR) \)
- \( \text{travel}(home, IAD) \) → \( \text{fly}(IAD, LHR) \) → \( \text{travel}(LHR, ICAPS) \)

}\) Insert set of task identifier constants

}\) Label task names with identifier constants

Alford, Behnke, Höller, Bercher, Biundo, Aha

Bound to Plan

June 17th, 2016
Translating Task Networks into PDDL

\[
\begin{align*}
\text{ticket}(IAD, LHR) \\
\text{travel}(home, IAD) &\rightarrow \text{fly}(IAD, LHR) &\rightarrow \text{travel}(LHR, ICAPS) \\
\end{align*}
\]

- \(t_1, t_2, t_3, t_4, \ldots, t_n\)
- \(\text{ticket}(IAD, LHR, t_4)\)
- \(\text{travel}(home, IAD, t_3)\)
- \(\text{fly}(IAD, LHR, t_2)\)
- \(\text{travel}(LHR, ICAPS, t_1)\)

\[
\begin{align*}
t_2 &\prec t_1, t_3 \prec t_2, t_4 \prec t_2 &\text{Insert set of task identifier constants} \\
&\quad \}
\end{align*}
\]

\[
\begin{align*}
&\quad \}
\end{align*}
\]

\[
\begin{align*}
&\quad \}
\end{align*}
\]

\[
\begin{align*}
&\quad \}
\end{align*}
\]

\[
\begin{align*}
&\quad \}
\end{align*}
\]
Translating Progression

HTN progression is defined with set notation

PDDL Preconditions: Universal quantification ($\forall v \phi(v)$)

PDDL Effects: Universally-quantified, conditional ($\forall v \phi(v) \Rightarrow \psi(v)$)

Use an appropriate bagging strategy for unused task identifiers

Derived predicates useful

Need to know how many task identifier constants to insert!
Translating Progression

- HTN progression is defined with set notation
  - PDDL Preconditions: Universal quantification ($\forall v \phi(v)$)
  - PDDL Effects: Universally-quantified, conditional ($\forall v \phi(v) \implies \psi(v)$)
Translating Progression

- HTN progression is defined with set notation
  - PDDL Preconditions: Universal quantification (\(\forall v \phi(v)\))
  - PDDL Effects: Universally-quantified, conditional (\(\forall v \phi(v) \implies \psi(v)\))
- Use an appropriate bagging strategy for unused task identifiers
  - Derived predicates useful
Translating Progression

- HTN progression is defined with set notation
  - PDDL Preconditions: Universal quantification ($\forall v \phi(v)$)
  - PDDL Effects: Universally-quantified, conditional ($\forall v \phi(v) \implies \psi(v)$)
- Use an appropriate bagging strategy for unused task identifiers
  - Derived predicates useful
- Need to know how many task identifier constants to insert!
Progression bounds

- Progression bound (PB): The size of the largest task network in a sequence of progressions (below, PB=3)
- Maximum PB: The maximum PB of any sequence of progressions to a solution
- Minimum PB: The minimum PB of any sequence of progressions to a solution
Progression bounds

- Progression bound (PB): The size of the largest task network in a sequence of progressions (below, PB=3)
- Maximum PB: The maximum PB of any sequence of progressions to a solution
- Minimum PB: The minimum PB of any sequence of progressions to a solution
- Min and Max PB are calculated on state-relaxed problems (no preconditions or effects)
Progression bounds

- Progression bound (PB): The size of the largest task network in a sequence of progressions (below, PB=3)
- Maximum PB: The maximum PB of any sequence of progressions to a solution ← focus of next slides
- Minimum PB: The minimum PB of any sequence of progressions to a solution ← dynamic programming, exists for all problems
- Min and Max PB are calculated on state-relaxed problems (no preconditions or effects)
Maximum Progression Bounds: $\leq_r$-stratification

- Tail-recursive problems are identified by $\leq_r$-stratification
- A $\leq_r$-stratification of task names: For every method $(t, tn)$,
  - **Last task** on equal or lower stratum (tail recursion).
  - Every other task of $tn$ must be on lower strata.
- A **maximal** $\leq_r$-stratification has tasks on the same level iff they are mutually recursive

**Methods:**

- `work` \(\rightarrow\) `drink` \(\rightarrow\) `work`
- `work` \(\rightarrow\) `write`
- `work` \(\rightarrow\) `drink` \(\rightarrow\) `write`

**Constraints:**

- $\text{strat}(\text{work}) \leq \text{strat}(\text{work})$
- $\text{strat}(\text{drink}) < \text{strat}(\text{work})$
- $\text{strat}(\text{write}) < \text{strat}(\text{work})$
Maximum Progression Bounds: $\leq_r$-stratification

- Tail-recursive problems are identified by $\leq_r$-stratification
- A $\leq_r$-stratification of task names: For every method $(t, tn)$,
  - **Last task** on equal or lower stratum (tail recursion).
  - Every other task of $tn$ must be on lower strata.
- A **maximal** $\leq_r$-stratification has tasks on the same level iff they are mutually recursive

Methods:

```
work → drink → work → write
work → drink → work → write
```

Constraints:

- $\text{strat(work)} \leq \text{strat(work)}$
- $\text{strat(drink)} < \text{strat(work)}$
- $\text{strat(write)} < \text{strat(work)}$

Here's one stratification:

<table>
<thead>
<tr>
<th>Method</th>
<th>Stratification</th>
</tr>
</thead>
<tbody>
<tr>
<td>work</td>
<td>3</td>
</tr>
<tr>
<td>drink</td>
<td>2</td>
</tr>
<tr>
<td>write</td>
<td>1</td>
</tr>
</tbody>
</table>
Bounding PBs for tasks with $\leq_r$-stratifications

- $\leq_r$-stratification gives an exponential upper bound to PB
- Finding **exact** MaxPB:
  - Start at bottom stratum.
  - $PB$(primitive stratum) = 1
  - Otherwise, PB is max PB of any associated method

Methods:

```
<table>
<thead>
<tr>
<th>Method</th>
<th>strat(work)</th>
<th>strat(drink)</th>
<th>strat(write)</th>
</tr>
</thead>
<tbody>
<tr>
<td>work</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>drink</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```
Max PB for task networks

**Task network** $tn$

- $PB(A) = 6$
- $PB(B) = 5$
- $PB(C) = 3$
- $PB(D) = 2$

- **Thm:** If $PB(t) = x$, $t$ adds 0, 1, or $x$ to the PB of a task network
- $PB(tn) \geq 6 + 5 \cdot 1 = 11$
Max PB for task networks

Thm: If $PB(t) = x$, $t$ adds 0, 1, or $x$ to the PB of a task network

$PB(A) = 6$
$PB(B) = 5$
$PB(C) = 3$
$PB(D) = 2$

$PB(tn) \geq 5 + 3 + 3 \cdot 1 = 11$
Max PB for task networks

Task network $tn$

- Thm: If $PB(t) = x$, $t$ adds 0, 1, or $x$ to the PB of a task network
- $PB(tn) \geq 5 + 5 + 2 \cdot 1 = 12$

$PB(A) = 6$
$PB(B) = 5$
$PB(C) = 3$
$PB(D) = 2$
Max PB for task networks

Thm: If $PB(t) = x$, $t$ adds 0, 1, or $x$ to the PB of a task network

$PB(\text{tn}) \geq 5 + 5 + 2 \cdot 1 = 12$

Want to find the progression bound without exhaustive search

$PB(A) = 6$
$PB(B) = 5$
$PB(C) = 3$
$PB(D) = 2$
Max-weighted independent set for transitive DAGs

Finding MaxPB with graph algorithms:

Task network $tn$

$PB(A) = 6$
$PB(B) = 5$
$PB(C) = 3$
$PB(D) = 2$
Max-weighted independent set for transitive DAGs

Task network $tn$

Finding MaxPB with graph algorithms:

- Label tasks with their maximum progression bounds

$PB(A) = 6$
$PB(B) = 5$
$PB(C) = 3$
$PB(D) = 2$
Max-weighted independent set for transitive DAGs

Task network $tn$

$PB(A) = 6$
$PB(B) = 5$
$PB(C) = 3$
$PB(D) = 2$

Finding MaxPB with graph algorithms:

- Label tasks with their maximum progression bounds
- Add dummy weights which can be selected when nodes are constrained
Max-weighted independent set for transitive DAGs

Task network $tn$

Finding MaxPB with graph algorithms:

- Label tasks with their maximum progression bounds
- Add dummy weights which can be selected when nodes are constrained
- Apply the Kagaris and Tragoudas alg for finding maximum weighted independent sets in transitively closed DAGs

$PB(A) = 6$
$PB(B) = 5$
$PB(C) = 3$
$PB(D) = 2$
Can classical planners efficiently handle translated HTN problems?
Evaluation on totally-ordered domains

- Used the Blocks World, Towers of Hanoi, and Robot Delivery domains
- Mean times (s) with Jasper:
  - 285 (Original domain), 5.2 (IJCAI-09), 3.2 (HTN2ADL)
Evaluation (2)

- Can we find reasonable bounds for HTN problems “in the wild?”
- Can we use classical planners to solve them?
### Evaluation on partially-ordered domains

<table>
<thead>
<tr>
<th>Domain</th>
<th>Problem Count</th>
<th>Translated Instances</th>
<th>Theoretical PB Min</th>
<th>Theoretical PB Max</th>
<th>Observed PB Min</th>
<th>Observed PB Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satellite</td>
<td>26</td>
<td>3340</td>
<td>1 – 8</td>
<td>5 – 40</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>SmartPhone</td>
<td>4</td>
<td>190</td>
<td>4 – 6</td>
<td>8 – 11</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>SmartPhone*</td>
<td>3</td>
<td>350</td>
<td>6 – 17</td>
<td></td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>UM-Translog</td>
<td>15</td>
<td>390</td>
<td>5 – 8</td>
<td>5 – 16</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>UM-Translog*</td>
<td>7</td>
<td>530</td>
<td>6 – 8</td>
<td></td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>Woodworking</td>
<td>5</td>
<td>220</td>
<td>3 – 6</td>
<td>3 – 10</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Combined</td>
<td>60</td>
<td>5020</td>
<td>1 – 17</td>
<td>3 – ∞</td>
<td>3</td>
<td>16</td>
</tr>
</tbody>
</table>

- Adapted hybrid HTN problems from the literature
- Translated each problem 10 times for each feasible PB
  - Translation is deterministic, planners are not.
- Jasper with the ADL HTN translation solved (or showed unsolvable) 74% of translated instances, and covering all but one problem
- Solved 78% of tail-recursive instances with max PB (41/50 probs).
Conclusions

Contributions:
- Translations for partially-ordered HTN problems
  - Paper includes two STRIPS-compatible translations
- Practical translation bounds for problems “in the wild”

Future work:
- Better bounding
  - Smart grounding
  - Reachability analysis
- Direct adaptation of classical heuristics
  - Already know this isn’t straight forward