Chapter 11
Hierarchical Task Network Planning

Dana S. Nau

University of Maryland
Fall 2009
Motivation

- We may already have an idea how to go about solving problems in a planning domain
- Example: travel to a destination that’s far away:
  - Domain-independent planner:
    » many combinations of vehicles and routes
  - Experienced human: small number of “recipes”
    e.g., flying:
    1. buy ticket from local airport to remote airport
    2. travel to local airport
    3. fly to remote airport
    4. travel to final destination

- How to enable planning systems to make use of such recipes?
Two Approaches

- **Control rules (previous chapter):**
  - Write rules to prune every action that doesn’t fit the recipe

- **Hierarchical Task Network (HTN) planning:**
  - Describe the actions and subtasks that do fit the recipe

```
Abstract-search(u)
    if Terminal(u) then return(u)
    u ← Refine(u) ;; refinement step
    B ← Branch(u) ;; branching step
    B' ← Prune(B) ;; pruning step
    if B' = ∅ then return(failure)
    nondeterministically choose v ∈ B'
    return(Abstract-search(v))
end
```

```
Abstract-search(u)
    if Terminal(u) then return(u)
    u ← Refine(u) ;; refinement step
    B ← Branch(u) ;; branching step
    B' ← Prune(B) ;; pruning step
    if B' = ∅ then return(failure)
    nondeterministically choose v ∈ B'
    return(Abstract-search(v))
end
```
HTN Planning

- Problem reduction
  - *Tasks* (activities) rather than goals
  - *Methods* to decompose tasks into subtasks
  - Enforce constraints
    - E.g., taxi not good for long distances
  - Backtrack if necessary
HTN Planning

- HTN planners may be domain-specific
  - e.g., see Chapters 20 (robotics) and 23 (bridge)
- Or they may be domain-configurable
  - Domain-independent planning engine
  - Domain description that defines not only the operators, but also the methods
  - Problem description
    » domain description, initial state, initial task network

Task: travel(x,y)

Method: taxi-travel(x,y)
get-taxi → ride(x,y) → pay-driver

Method: air-travel(x,y)
get-ticket(a(x),a(y))
fly(a(x),a(y)) → travel(a(y),y)

travel(x,a(x))
Simple Task Network (STN) Planning

- A special case of HTN planning
- States and operators
  - The same as in classical planning
- Task: an expression of the form \( t(u_1, \ldots, u_n) \)
  - \( t \) is a task symbol, and each \( u_i \) is a term
  - Two kinds of task symbols (and tasks):
    » primitive: tasks that we know how to execute directly
      - task symbol is an operator name
    » nonprimitive: tasks that must be decomposed into subtasks
      - use methods (next slide)
Methods

- Totally ordered method: a 4-tuple
  \[ m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m)) \]
  - **name**(m): an expression of the form \( n(x_1, \ldots, x_n) \)
    - \( x_1, \ldots, x_n \) are parameters - variable symbols
  - **task**(m): a nonprimitive task
  - **precond**(m): preconditions (literals)
  - **subtasks**(m): a sequence of tasks \( \langle t_1, \ldots, t_k \rangle \)

\[ \text{air-travel}(x,y) \]

- **task**: \( \text{travel}(x,y) \)
- **precond**: \( \text{long-distance}(x,y) \)
- **subtasks**: \( \langle \text{buy-ticket}(a(x), a(y)), \text{travel}(x, a(x)), \text{fly}(a(x), a(y)), \text{travel}(a(y), y) \rangle \)
Methods (Continued)

- Partially ordered method: a 4-tuple
  \[ m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m)) \]
  - name(m): an expression of the form \( n(x_1, \ldots, x_n) \)
    \( x_1, \ldots, x_n \) are parameters - variable symbols
  - task(m): a nonprimitive task
  - precond(m): preconditions (_literals)
  - subtasks(m): a partially ordered set of tasks \( \{t_1, \ldots, t_k\} \)

\[ \text{air-travel}(x,y) \]
\[ \text{task}: \quad \text{travel}(x,y) \]
\[ \text{precond}: \quad \text{long-distance}(x,y) \]
\[ \text{network}: \quad u_1=\text{buy-ticket}(a(x), a(y)), \ u_2=\text{travel}(x, a(x)), \ u_3=\text{fly}(a(x), a(y)) \]
\[ u_4=\text{travel}(a(y), y), \ \{(u_1,u_3), (u_2,u_3), (u_3,u_4)\} \]
Domains, Problems, Solutions

- STN planning domain: methods, operators
- STN planning problem: methods, operators, initial state, task list
- Total-order STN planning domain and planning problem:
  - Same as above except that all methods are totally ordered

- Solution: any executable plan that can be generated by recursively applying
  - methods to nonprimitive tasks
  - operators to primitive tasks
Example

- Suppose we want to move three stacks of containers in a way that preserves the order of the containers.
Example (continued)

- A way to move each stack:
  - first move the containers from $p$ to an intermediate pile $r$
  - then move them from $r$ to $q$
take-and-put\((c, k, l_1, l_2, p_1, p_2, x_1, x_2)\):
  \[\text{task: } \text{move-topmost-container}(p_1, p_2)\]
  \[\text{precond: } \text{top}(c, p_1), \text{on}(c, x_1), \text{attached}(p_1, l_1), \text{belong}(k, l_1), \text{attached}(p_2, l_2), \text{top}(x_2, p_2)\] ; true if \(p_1\) is not empty
  \[\text{subtasks: } \langle\text{take}(k, l_1, c, x_1, p_1), \text{put}(k, l_2, c, x_2, p_2)\rangle\]

recursive-move\((p, q, c, x)\):
  \[\text{task: } \text{move-stack}(p, q)\]
  \[\text{precond: } \text{top}(c, p), \text{on}(c, x)\] ; true if \(p\) is not empty
  \[\text{subtasks: } \langle\text{move-topmost-container}(p, q), \text{move-stack}(p, q)\rangle\]
  ;; the second subtask recursively moves the rest of the stack

do-nothing\((p, q)\)
  \[\text{task: } \text{move-stack}(p, q)\]
  \[\text{precond: } \text{top}(\text{pallet}, p)\] ; true if \(p\) is empty
  \[\text{subtasks: } \langle\rangle\] ; no subtasks, because we are done

move-each-twice()
  \[\text{task: } \text{move-all-stacks}()\]
  \[\text{precond: } \text{no preconditions}\]
  \[\text{network: } \text{move each stack twice:}\]
  \[u_1 = \text{move-stack}(p1a, p1b), u_2 = \text{move-stack}(p1b, p1c), u_3 = \text{move-stack}(p2a, p2b), u_4 = \text{move-stack}(p2b, p2c), u_5 = \text{move-stack}(p3a, p3b), u_6 = \text{move-stack}(p3b, p3c), \{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}\]
take-and-put\( (c, k, l_1, l_2, p_1, p_2, x_1, x_2) \):
  task: \( \text{move-topmost-container}(p_1, p_2) \)
  precond: \( \text{top}(c, p_1), \text{on}(c, x_1), \text{true} \text{ if } p_1 \text{ is not empty} \)
    \( \text{attached}(p_1, l_1), \text{belong}(k, l_1), \text{bind } l_1 \text{ and } k \)
    \( \text{attached}(p_2, l_2), \text{top}(x_2, p_2), \text{bind } l_2 \text{ and } x_2 \)
  subtasks: \( \langle \text{take}(k, l_1, c, x_1, p_1), \text{put}(k, l_2, c, x_2, p_2) \rangle \)

recursive-move\( (p, q, c, x) \):
  task: \( \text{move-stack}(p, q) \)
  precond: \( \text{top}(c, p), \text{on}(c, x), \text{true} \text{ if } p \text{ is not empty} \)
  subtasks: \( \langle \text{move-topmost-container}(p, q), \text{move-stack}(p, q) \rangle \)
    ;; the second subtask recursively moves the rest of the stack

do-nothing\( (p, q) \)
  task: \( \text{move-stack}(p, q) \)
  precond: \( \text{top}(\text{pallet}, p), \text{true} \text{ if } p \text{ is empty} \)
  subtasks: \( \langle \rangle \) ;; no subtasks, because we are done

move-each-twice()\)
  task: \( \text{move-all-stacks()} \)
  precond: \( ; \text{no preconditions} \)
  subtasks: \( ; \text{move each stack twice:} \)
    \( \langle \text{move-stack}(p1a, p1b), \text{move-stack}(p1b, p1c), \text{move-stack}(p2a, p2b), \text{move-stack}(p2b, p2c), \text{move-stack}(p3a, p3b), \text{move-stack}(p3b, p3c) \rangle \)
Solving Total-Order STN Planning Problems

\[ \text{TFD}(s, (t_1, \ldots, t_k), O, M) \]
\[ \text{if } k = 0 \text{ then return } \emptyset \] (i.e., the empty plan)
\[ \text{if } t_1 \text{ is primitive then} \]
\[ \begin{align*}
\text{active} & \leftarrow \{(a, \sigma) \mid a \text{ is a ground instance of an operator in } O, \\
& \quad \sigma \text{ is a substitution such that } a \text{ is relevant for } \sigma(t_1), \\
& \quad \text{and } a \text{ is applicable to } s\} \\
\text{if } \text{active} = \emptyset \text{ then return failure} \\
\text{nondeterministically choose any } (a, \sigma) \in \text{active} \\
\pi & \leftarrow \text{TFD}(\gamma(s, a), \sigma((t_2, \ldots, t_k)), O, M) \\
\text{if } \pi = \text{failure then return failure} \\
\text{else return } a. \pi
\end{align*} \]
\[ \text{else if } t_1 \text{ is nonprimitive then} \]
\[ \begin{align*}
\text{active} & \leftarrow \{m \mid m \text{ is a ground instance of a method in } M, \\
& \quad \sigma \text{ is a substitution such that } m \text{ is relevant for } \sigma(t_1), \\
& \quad \text{and } m \text{ is applicable to } s\} \\
\text{if } \text{active} = \emptyset \text{ then return failure} \\
\text{nondeterministically choose any } (m, \sigma) \in \text{active} \\
w & \leftarrow \text{subtasks}(m). \sigma((t_2, \ldots, t_k)) \\
\text{return } \text{TFD}(s, w, O, M)
\end{align*} \]
Comparison to Forward and Backward Search

- In state-space planning, must choose whether to search forward or backward

- In HTN planning, there are two choices to make about direction:
  - forward or backward
  - up or down

- TFD goes down and forward
Comparison to Forward and Backward Search

- Like a backward search, TFD is goal-directed
  - Goals correspond to tasks

- Like a forward search, it generates actions in the same order in which they’ll be executed

- Whenever we want to plan the next task
  - we’ve already planned everything that comes before it
  - Thus, we know the current state of the world
Limitation of Ordered-Task Planning

- TFD requires totally ordered methods
  - Can’t interleave subtasks of different tasks
  - Sometimes this makes things awkward
    - Need to write methods that reason globally instead of locally
Partially Ordered Methods

- With partially ordered methods, the subtasks can be interleaved

- Fits many planning domains better
- Requires a more complicated planning algorithm
Algorithm for Partial-Order STNs

\[
\begin{align*}
\text{PFD}(s, w, O, M) & \\
& \text{if } w = \emptyset \text{ then return the empty plan} \\
& \text{nondeterministically choose any } u \in w \text{ that has no predecessors in } w \\
& \text{if } t_u \text{ is a primitive task then} \\
& \quad \text{active} \leftarrow \{(a, \sigma) \mid a \text{ is a ground instance of an operator in } O, \\
& \quad \quad \sigma \text{ is a substitution such that } \text{name}(a) = \sigma(t_u), \\
& \quad \quad \text{and } a \text{ is applicable to } s\} \\
& \text{if } \text{active} = \emptyset \text{ then return failure} \\
& \text{nondeterministically choose any } (a, \sigma) \in \text{active} \\
& \pi \leftarrow \text{PFD}(\gamma(s, a), \sigma(w - \{u\}), O, M) \\
& \text{if } \pi = \text{failure} \text{ then return failure} \\
& \text{else return } a. \pi \\
& \text{else} \\
& \quad \text{active} \leftarrow \{(m, \sigma) \mid m \text{ is a ground instance of a method in } M, \\
& \quad \quad \sigma \text{ is a substitution such that } \text{name}(m) = \sigma(t_u), \\
& \quad \quad \text{and } m \text{ is applicable to } s\} \\
& \text{if } \text{active} = \emptyset \text{ then return failure} \\
& \text{nondeterministically choose any } (m, \sigma) \in \text{active} \\
& \text{nondeterministically choose any task network } w' \in \delta(w, u, m, \sigma) \\
& \text{return}(\text{PFD}(s, w', O, M))
\end{align*}
\]
Algorithm for Partial-Order STNs

- Intuitively, $w$ is a partially ordered set of tasks \( \{t_1, t_2, \ldots \} \)
  - But $w$ may contain a task more than once
    » e.g., travel from UMD to LAAS twice
  - The mathematical definition of a set doesn’t allow this
- Define $w$ as a partially ordered set of task nodes \( \{u_1, u_2, \ldots \} \)
  - Each task node $u$ corresponds to a task $t_u$
- In my explanations, I talk about $t$ and ignore $u$

Else

active $\leftarrow \{(m, \sigma) \mid m \text{ is a ground instance of a method in } M,$
\[ \sigma \text{ is a substitution such that } \text{name}(m) = \sigma(t_u), \]
and $m$ is applicable to $s\}$

if active $= \emptyset$ then return failure
nondeterministically choose any $(m, \sigma) \in \text{active}$
nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
return(PFD($s, w', O, M$))
Algorithm for Partial-Order STNs

\[ \pi = \{a_1, \ldots, a_k\}; \quad w = \{t_1, t_2, t_3, \ldots\} \]

operator instance \(a\)

\[ \pi = \{a_1, \ldots, a_k, a\}; \quad w' = \{t_2, t_3, \ldots\} \]

method instance \(m\)

\[ w = \{t_1, t_2, \ldots\} \]

\[ w' = \{t_{11}, \ldots, t_{1k}, t_2, \ldots\} \]
Algorithm for Partial-Order STNs

\[ \pi = \{ a_1, \ldots, a_k \}; \quad w' = \{ t_2, t_3, \ldots \} \]

δ(w, u, m, σ) has a complicated definition in the book. Here’s what it means:

- We nondeterministically selected \( t_1 \) as the task to do first
- Must do \( t_1 \)’s first subtask before the first subtask of every \( t_i \neq t_1 \)
- Insert ordering constraints to ensure that this happens
Comparison to Classical Planning

STN planning is strictly more expressive than classical planning

- Any classical planning problem can be translated into an ordered-task-planning problem in polynomial time
- Several ways to do this. One is roughly as follows:
  - For each goal or precondition $e$, create a task $t_e$
  - For each operator $o$ and effect $e$, create a method $m_{o,e}$
    - Task: $t_e$
    - Subtasks: $t_{c_1}, t_{c_2}, \ldots, t_{c_n} o$, where $c_1, c_2, \ldots, c_n$ are the preconditions of $o$
    - Partial-ordering constraints: each $t_{c_i}$ precedes $o$

- (I left out some details, such as how to handle deleted-condition interactions)
Some STN planning problems aren’t expressible in classical planning

Example:

- Two STN methods:
  - No arguments
  - No preconditions

- Two operators, $\text{a}$ and $\text{b}$
  - Again, no arguments and no preconditions

- Initial state is empty, initial task is $\text{t}$

- Set of solutions is $\{\text{a}^n\text{b}^n \mid n > 0\}$

- No classical planning problem has this set of solutions
  - The state-transition system is a finite-state automaton
  - No finite-state automaton can recognize $\{\text{a}^n\text{b}^n \mid n > 0\}$

Can even express undecidable problems using STNs
Increasing Expressivity Further

- Knowing the current state makes it easy to do things that would be difficult otherwise

  - States can be arbitrary data structures

  | Us:     | East declarer, West dummy |
  | Opponents: | defenders, South & North |
  | Contract: | East – 3NT |
  | On lead:  | West at trick 3 |

  - Preconditions and effects can include

    - logical inferences (e.g., Horn clauses)
    - complex numeric computations
    - interactions with other software packages

- e.g., SHOP and SHOP2:

  http://www.cs.umd.edu/projects/shop

Dana Nau: Lecture slides for Automated Planning
Licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License: http://creativecommons.org/licenses/by-nc-sa/2.0/
**Example**

- Simple travel-planning domain
  - Go from one location to another
  - State-variable formulation

```plaintext
method travel-by-foot
  precond: distance(x, y) ≤ 2
  task:    travel(a, x, y)
  subtasks: walk(a, x, y)

method travel-by-taxi
  task:    travel(a, x, y)
  precond: cash(a) ≥ 1.5 + 0.5 × distance(x, y)
  subtasks: ⟨call-taxi(a, x), ride(a, x, y), pay-driver(a, x, y)⟩

operator walk
  precond: location(a) = x
  effects: location(a) ← y

operator call-taxi(a, x)
  effects: location(taxi) ← x

operator ride(a, x)
  precond: location(taxi) = x, location(a) = x
  effects: location(taxi) ← y, location(a) ← y

operator pay-driver(a, x, y)
  precond: cash(a) ≥ 1.5 + 0.5 × distance(x, y)
  effects: cash(a) ← cash(a) − 1.5 − 0.5 × distance(x, y).
```
Planning Problem: I am at home, I have $20, I want to go to a park 8 miles away

Initial task: travel(me,home,park)

Precond: distance(home,park) ≤ 2

Precondition fails

travel-by-foot

Precond: cash(me) ≥ 1.50 + 0.50*distance(home,park)

Precondition succeeds

travel-by-taxi

Decomposition into subtasks

Initial state: s₀ = {location(me)=home, cash(me)=20, distance(home,park)=8}

Call taxi

s₁ = {location(me)=home, location(taxi)=home, cash(me)=20, distance(home,park)=8}

Precond: ...

Effects: ...

Initial state: s₁

Ride taxi

s₂ = {location(me)=park, location(taxi)=park, cash(me)=20, distance(home,park)=8}

Precond: ...

Effects: ...

Initial state: s₂

Pay driver

s₃ = {location(me)=park, location(taxi)=park, cash(me)=14.50, distance(home,park)=8}

Precond: ...

Effects: ...

Initial state: s₃

Final state: s₃
SHOP2

- SHOP2: implementation of PFD-like algorithm + generalizations
  - Won one of the top four awards in the AIPS-2002 Planning Competition
  - Freeware, open source
  - Implementation available at
    http://www.cs.umd.edu/projects/shop
HTN Planning

● HTN planning is even more general
  ◆ Can have constraints associated with tasks and methods
    » Things that must be true before, during, or afterwards
  ◆ Some algorithms use causal links and threats like those in PSP
● There’s a little about this in the book
  ◆ I won’t discuss it
SHOP & SHOP2 vs. TLPlan & TALplanner

- These planners have equivalent expressive power
  - Turing-complete, because both allow function symbols
- They know the current state at each point during the planning process, and use this to prune actions
  - Makes it easy to call external subroutines, do numeric computations, etc.
- Main difference: how the pruning is done
  - SHOP and SHOP2: the methods say what *can* be done
    » Don’t do anything unless a method says to do it
  - TLPlan and TALplanner: the say what *cannot* be done
    » Try everything that the control rules don’t prohibit
- Which approach is more convenient depends on the problem domain
Domain-Configurable Planners Compared to Classical Planners

- Disadvantage: writing a knowledge base can be more complicated than just writing classical operators
- Advantage: can encode “recipes” as collections of methods and operators
  - Express things that can’t be expressed in classical planning
  - Specify standard ways of solving problems
    » Otherwise, the planning system would have to derive these again and again from “first principles,” every time it solves a problem
    » Can speed up planning by many orders of magnitude (e.g., polynomial time versus exponential time)
Example from the AIPS-2002 Competition

- The satellite domain
  - Planning and scheduling observation tasks among multiple satellites
  - Each satellite equipped in slightly different ways
- Several different versions. I’ll show results for the following:
  - **Simple-time:**
    - concurrent use of different satellites
    - data can be acquired more quickly if they are used efficiently
  - **Numeric:**
    - fuel costs for satellites to slew between targets; finite amount of fuel available.
    - data takes up space in a finite capacity data store
    - Plans are expected to acquire all the necessary data at minimum fuel cost.
  - **Hard Numeric:**
    - *no logical goals at all* – thus even the null plan is a solution
    - Plans that acquire more data are better – thus the null plan has no value
    - None of the classical planners could handle this