Chapter 11
Hierarchical Task Network Planning

Dana S. Nau
University of Maryland

2:26 PM April 18, 2012
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($u$)
  $u \leftarrow \text{Refine}(u)$ ;; refinement step
  $B \leftarrow \text{Branch}(u)$ ;; branching step
  $B' \leftarrow \text{Prune}(B)$ ;; pruning step
  if $B' = \emptyset$ then return(failure)
  nondeterministically choose $v \in B'$
  return(Abstract-search($v$))
end

Abstract-search($u$)
  if Terminal($u$) then return($u$)
  $u \leftarrow \text{Refine}(u)$ ;; refinement step
  $B \leftarrow \text{Branch}(u)$ ;; branching step
  $B' \leftarrow \text{Prune}(B)$ ;; pruning step
  if $B' = \emptyset$ then return(failure)
  nondeterministically choose $v \in B'$
  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

---

*Tasks:*

- 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($x,a(x)$)

**BACKTRACK**

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

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

- \text{task}: \text{travel}(x, y)
- \text{precond}: \text{long-distance}(x, y)
- \text{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 \( \{t_1, \ldots, t_k\} \)

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

- task: \( \text{travel}(x,y) \)

- precond: \( \text{long-distance}(x,y) \)

- network: \( u_1 = \text{buy-ticket}(a(x), a(y)), u_2 = \text{travel}(x, a(x)), u_3 = \text{fly}(a(x), a(y)) \)

  \( u_4 = \text{travel}(a(y), y), \{ (u_1, u_3), (u_2, u_3), (u_3, u_4) \} \)
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\((c, k, l_1, l_2, p_1, p_2, x_1, x_2)\):
  task: move-topmost-container(p_1, p_2)
  precond: top(c, p_1), on(c, x_1), ; true if \(p_1\) is not empty
           attached(p_1, l_1), belong(k, l_1), ; bind \(l_1\) and \(k\)
           attached(p_2, l_2), top(x_2, p_2) ; bind \(l_2\) 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: move-stack(p, q)
  precond: top(c, p), on(c, x) ; true if \(p\) 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: move-stack(p, q)
  precond: top(pallet, p) ; true if \(p\) is empty
  subtasks: \(\langle \rangle \) ; no subtasks, because we are done

move-each-twice()
  task: move-all-stacks()
  precond: ; no preconditions
  subtasks: ; move each stack twice:
             \(\langle \text{move-stack}(p_1a, p_1b), \text{move-stack}(p_1b, p_1c), \text{move-stack}(p_2a, p_2b), \text{move-stack}(p_2b, p_2c), \text{move-stack}(p_3a, p_3b), \text{move-stack}(p_3b, p_3c) \rangle\)
Partial-Order Formulation

take-and-put\(c, k, l_1, l_2, p_1, p_2, x_1, x_2\):
  task:  move-topmost-container\(p_1, p_2\)
  precond:  top\(c, p_1\), on\(c, x_1\), ; true if \(p_1\) is not empty
  attached\(p_1, l_1\), belong\(k, l_1\), ; bind \(l_1\) and \(k\)
  attached\(p_2, l_2\), top\(x_2, p_2\) ; bind \(l_2\) and \(x_2\)
  subtasks:  \(\langle\)take\(k, l_1, c, x_1, p_1\), put\(k, l_2, c, x_2, p_2\)\(\rangle\)

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

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

move-each-twice():
  task:  move-all-stacks()
  precond:  ; no preconditions
  network:  ; move each stack twice:
    \(u_1 =\)move-stack\(p_1a, p_1b\), \(u_2 =\)move-stack\(p_1b, p_1c\),
    \(u_3 =\)move-stack\(p_2a, p_2b\), \(u_4 =\)move-stack\(p_2b, p_2c\),
    \(u_5 =\)move-stack\(p_3a, p_3b\), \(u_6 =\)move-stack\(p_3b, p_3c\),
    \(\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}\)
Solving Total-Order STN Planning Problems

\[
\text{TFD}(s, (t_1, \ldots, t_k), O, M)
\]

if \( k = 0 \) then return \( \langle \rangle \) (i.e., the empty plan)

if \( t_1 \) is primitive then

\[
\text{active} \leftarrow \{(a, \sigma) \mid a \text{ is a ground instance of an operator in } O, \\
\sigma \text{ is a substitution such that } a \text{ is relevant for } \sigma(t_1), \\
\text{and } a \text{ is applicable to } s\}
\]

if \( \text{active} = \emptyset \) then return failure

nondeterministically choose any \((a, \sigma) \in \text{active}\)

\[
\pi \leftarrow \text{TFD}(\gamma(s, a), \sigma(\langle t_2, \ldots, t_k \rangle)), O, M)
\]

if \( \pi = \text{failure} \) then return failure

else return \( a. \pi \)

else if \( t_1 \) is nonprimitive then

\[
\text{active} \leftarrow \{m \mid m \text{ is a ground instance of a method in } M, \\
\sigma \text{ is a substitution such that } m \text{ is relevant for } \sigma(t_1), \\
\text{and } m \text{ is applicable to } s\}
\]

if \( \text{active} = \emptyset \) then return failure

nondeterministically choose any \((m, \sigma) \in \text{active}\)

\[
w \leftarrow \text{subtasks}(m). \sigma(\langle t_2, \ldots, t_k \rangle)
\]

return \( \text{TFD}(s, w, O, M) \)
Comparison to Forward and Backward Search

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

  ![State-space planning diagram]

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

  ![HTN planning diagram]

- 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

Algorithm: PFD($s, w, O, M$)

1. if $w = \emptyset$ then return the empty plan
2. nondeterministically choose any $u \in w$ that has no predecessors in $w$
3. if $t_u$ is a primitive task then
   a. $active \leftarrow \{(a, \sigma) \mid a \text{ is a ground instance of an operator in } O,$
      \hspace{1cm} \sigma \text{ is a substitution such that } \text{name}(a) = \sigma(t_u),$
      \hspace{1cm} \text{and } a \text{ is applicable to } s\}$
4. if $active = \emptyset$ then return failure
5. nondeterministically choose any $(a, \sigma) \in active$
6. $\pi \leftarrow \text{PFD}(\gamma(s, a), \sigma(w - \{u\}), O, M)$
7. if $\pi = \text{failure}$ then return failure
8. else return $a. \pi$

else

9. $active \leftarrow \{(m, \sigma) \mid m \text{ is a ground instance of a method in } M,$
   \hspace{1cm} \sigma \text{ is a substitution such that } \text{name}(m) = \sigma(t_u),$
   \hspace{1cm} \text{and } m \text{ is applicable to } s\}$
10. if $active = \emptyset$ then return failure
11. nondeterministically choose any $(m, \sigma) \in active$
12. nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$$$
13. return(PFD(s, w', O, M)
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’ll talk about \( t \) and ignore \( u \)
Algorithm for Partial-Order STNs

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

operator instance \( a \)

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

method instance \( m \)

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

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

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

\[
\delta(w, u, m, \sigma) \text{ has a complicated definition in the book. Here’s what it means:}
\]

- We nondeterministically selected \(t_1\) as the task to begin first
  - i.e., 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

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

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

method instance \(m\)
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)
Comparison to Classical Planning (cont.)

● Some STN planning problems aren’t expressible in classical planning

● Example:
  ◆ Two STN methods:
    » No arguments
    » No preconditions
  ◆ Two operators, a and b
    » Again, no arguments and no preconditions
  ◆ Initial state is empty, initial task is t
  ◆ Set of solutions is \{a^n 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 \{a^n b^n \mid n > 0\}

● Can even express undecidable problems using STNs
Increasing Expressivity Further

- If we always know the current state, we can make several enhancements:
  - States can be arbitrary data structures
  - Preconditions and effects can include
    - logical inferences (e.g., Horn clauses)
    - complex numeric computations
    - interactions with other software packages
  - e.g., SHOP and SHOP2
    - algorithms similar to PFD and PFD, with the above enhancements
    - SHOP2 won an award at the 2002 Planning Competition
Increasing Expressivity Further

- If we always know the current state, we can make several enhancements:
  - States can be arbitrary data structures

<table>
<thead>
<tr>
<th>Us: East declarer, West dummy</th>
<th>Opponents: defenders, South &amp; North</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contract: East – 3NT</td>
<td>On lead: West at trick 3</td>
</tr>
<tr>
<td>East: ♠KJ74</td>
<td>West: ♠A2</td>
</tr>
<tr>
<td>Out: ♠QT98653</td>
<td></td>
</tr>
</tbody>
</table>

- Preconditions and effects can include
  - logical inferences (e.g., Horn clauses)
  - complex numeric computations
  - interactions with other software packages

- TLPlan and TALplanner also have some (but not all) of these enhancements

- What about adding them to a planner like FastForward?
Example

- Simple travel-planning domain
  - State-variable formulation
- Planning problem:
  - I’m at home, I have $20
  - Want to go to a park 8 miles away

\[ s_0 = \{\text{location}(\text{me}) = \text{home}, \text{cash}(\text{me}) = 20, \text{distance}(\text{home, park}) = 8\} \]

\[ t_0 = \text{travel}(\text{me, home, park}) \]

- \text{method} travel-by-foot
  - precondition: \( \text{distance}(x, y) \leq 2 \)
  - task: \( \text{travel}(a, x, y) \)
  - subtasks: \( \text{walk}(a, x, y) \)

- \text{method} travel-by-taxi
  - task: \( \text{travel}(a, x, y) \)
  - precondition: \( \text{cash}(a) \geq 1.5 + 0.5 \times \text{distance}(x, y) \)
  - subtasks: \( \langle \text{call-taxi}(a, x), \text{ride}(a, x, y), \text{pay-driver}(a, x, y) \rangle \)

- \text{operator} walk
  - precondition: \( \text{location}(a) = x \)
  - effects: \( \text{location}(a) \leftarrow y \)

- \text{operator} call-taxi(a, x)
  - effects: \( \text{location}(\text{taxi}) \leftarrow x \)

- \text{operator} ride-taxi(a, x)
  - precondition: \( \text{location}(\text{taxi}) = x, \text{location}(a) = x \)
  - effects: \( \text{location}(\text{taxi}) \leftarrow y, \text{location}(a) \leftarrow y \)

- \text{operator} pay-driver(a, x, y)
  - precondition: \( \text{cash}(a) \geq 1.5 + 0.5 \times \text{distance}(x, y) \)
  - effects: \( \text{cash}(a) \leftarrow \text{cash}(a) - 1.5 - 0.5 \times \text{distance}(x, y) \)
Initial task: \( \text{travel}(\text{me}, \text{home}, \text{park}) \)

- **Precondition**: \( \text{distance}(\text{home}, \text{park}) \leq 2 \)
- **Precondition fails**

- **Decomposition into subtasks**
  - \( \text{travel-by-foot} \)
  - \( \text{travel-by-taxi} \)

**Example, Continued**

- **Initial state**:
  - \( \text{location}(\text{me}) = \text{home}, \text{cash}(\text{me}) = 20, \text{distance}(\text{home}, \text{park}) = 8 \)

- **Subtasks**:
  - \( \text{call-taxi}(\text{me}, \text{home}) \)
  - \( \text{ride}(\text{me}, \text{home}, \text{park}) \)
  - \( \text{pay-driver}(\text{me}, \text{home}, \text{park}) \)

- **Final state**:
  - \( \text{location}(\text{me}) = \text{park}, \text{location}(\text{taxi}) = \text{park}, \text{cash}(\text{me}) = 14.50, \text{distance}(\text{home}, \text{park}) = 8 \)
HTN Planning

- HTN planning can be 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