Chapter 11
Hierarchical Task Network Planning
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 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
Task: \( \text{travel}(x,y) \)

**Method:** \( \text{taxi-travel}(x,y) \)

- get-taxi
- \( \text{ride}(x,y) \)
- pay-driver

**Method:** \( \text{air-travel}(x,y) \)

- get-ticket(\( a(x),a(y) \))
- fly(\( a(x),a(y) \))
- \( \text{travel}(a(y),y) \)
- \( \text{travel}(x,a(x)) \)

**Method:** \( \text{taxi-travel}(x,y) \)

- get-ticket(\( a(x),a(y) \))
- fly(\( a(x),a(y) \))
- \( \text{travel}(a(y),y) \)

**Method:** \( \text{taxi-travel}(x,y) \)

- get-ticket(\( a(x),a(y) \))
- fly(\( a(x),a(y) \))
- \( \text{travel}(a(y),y) \)

**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., Robotics (Chapters 20) and Bridge (Chapter 23)

- Or they may be domain-configurable
  - Domain-independent planning engine
  - Domain description
    - methods, operators
  - Problem description
    - domain description, initial state, initial task network

Abstract-search($u$)
  if Terminal($u$) then return($u$)
  $u \leftarrow$ Refine($u$) ;; refinement step
  $B \leftarrow$ Branch($u$) ;; branching step
  $B' \leftarrow$ Prune($B$) ;; pruning step
  if $B' = \emptyset$ then return(failure)
  nondeterministically choose $v \in B'$
  return(Abstract-search($v$))
end
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 \(<t_1,\ldots,t_k>\)

- **air-travel**\((x,y)\)
  - **task**: \(\text{travel}(x,y)\)
  - **precond**: \(\text{long-distance}(x,y)\)
  - **subtasks**: \(<\text{buy-ticket}(a(x),a(y)), \text{travel}(x,a(x)), \text{fly}(a(x),a(y)), \text{travel}(a(y),y)>\)
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 \( \{ t_1, \ldots, t_k \} \)

**Network:**
- **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 subtasks 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\((c, k, l_1, l_2, p_1, p_2, x_1, x_2)\):
  \[\text{task: move-topmost-container}(p_1, p_2)\]
  \[\text{precond: top}(c, p_1), \text{on}(c, x_1), \quad \text{true if } p_1 \text{ is not empty}\]
  \[\text{attached}(p_1, l_1), \text{belong}(k, l_1), \quad \text{bind } l_1 \text{ and } k\]
  \[\text{attached}(p_2, l_2), \text{top}(x_2, p_2), \quad \text{bind } l_2 \text{ and } x_2\]
  \[\text{subtasks: } \langle \text{take}(k, l_1, c, x_1, p_1), \text{put}(k, l_2, c, x_2, p_2) \rangle\]

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

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

move-each-twice()
  \[\text{task: move-all-stacks()}\]
  \[\text{precond: } \text{no preconditions}\]
  \[\text{network: move each stack twice:}\]
    \[u_1 = \text{move-stack}(p_{1a}, p_{1b}), \quad u_2 = \text{move-stack}(p_{1b}, p_{1c}),\]
    \[u_3 = \text{move-stack}(p_{2a}, p_{2b}), \quad u_4 = \text{move-stack}(p_{2b}, p_{2c}),\]
    \[u_5 = \text{move-stack}(p_{3a}, p_{3b}), \quad u_6 = \text{move-stack}(p_{3b}, p_{3c}),\]
    \[\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}\]
take-and-put\((c, k, l_1, l_2, p_1, p_2, x_1, x_2)\):
  task: \move-topmost-container(p_1, p_2)
  precond: top(c, p_1), on(c, x_1), \(\text{true if } p_1 \text{ is not empty}\)
  attached(p_1, l_1), belong(k, l_1), \(\text{bind } l_1 \text{ and } k\)
  attached(p_2, l_2), top(x_2, p_2) \(\text{bind } l_2 \text{ and } x_2\)
  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) \(\text{true if } p \text{ is not empty}\)
  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(pallet, p) \(\text{true if } p \text{ is empty}\)
  subtasks: \(\langle \rangle\) \(\text{no subtasks, because we are done}\)

move-each-twice()
  task: \move-all-stacks()
  precond: \(\text{no preconditions}\)
  subtasks: \(\text{move each stack twice:}\)
  \(\langle \move-stack(p_{1a}, p_{1b}), \move-stack(p_{1b}, p_{1c}), \move-stack(p_{2a}, p_{2b}), \move-stack(p_{2b}, p_{2c}), \move-stack(p_{3a}, p_{3b}), \move-stack(p_{3b}, p_{3c}) \rangle\)
Solving Total-Order STN Planning Problems

TFD(s, \langle t_1, \ldots, t_k \rangle, 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, \right.
  \sigma \text{ is a substitution such that } a \text{ is relevant for } \sigma(t_1), \left.
  \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 TFD(\gamma(s, a), \sigma(\langle t_2, \ldots, t_k \rangle), O, M)
if \pi = failure then return failure
else return \(a \cdot \pi\)
else if t_1 is nonprimitive then
  \text{active} \leftarrow \{m \mid m \text{ is a ground instance of a method in } M, \right.
  \sigma \text{ is a substitution such that } m \text{ is relevant for } \sigma(t_1), \left.
  \text{and } m \text{ is applicable to } s\}\)
if \text{active} = \emptyset then return failure
nondeterministically choose any \((m, \sigma) \in \text{active} \)
\omega \leftarrow \text{subtasks}(m). \sigma(\langle t_2, \ldots, t_k \rangle)
return TFD(s, \omega, O, M)
Expressivity Relative to Classical Planning

- Any classical planning 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 $o$
- There are HTN planning problems that cannot be translated into classical planning problems at all
- Example on the next page
Example

- Two 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 \( \{a^n b^n \mid n > 0\} \)
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
Increasing Expressivity Further

- Knowing the current state makes it easy to do things that would be difficult otherwise
  - States can be arbitrary data structures
  - Preconditions and effects can include
    - logical inferences (e.g., Horn clauses)
    - complex numeric computations
- e.g., SHOP:
  http://www.cs.umd.edu/projects/shop
Example

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

Precondition: distance(home,park) ≤ 2
Precondition fails

Precondition succeeds

Decomposition into subtasks

Initial state $s_0 = \{\text{location}(\text{me})=\text{home}, \text{cash}(\text{me})=20, \text{distance}(\text{home},\text{park})=8\}$

Subtasks:
- $s_1$: call-taxi(me,home)
  - Precond: ...
  - Effects: ...

- $s_2$: ride(me,home,park)
  - Precond: ...
  - Effects: ...

- $s_3$: pay-driver(me,home,park)
  - Precond: ...
  - Effects: ...

Final state $s_3 = \{\text{location}(\text{me})=\text{park}, \text{location}(\text{taxi})=\text{park}, \text{cash}(\text{me})=14.50, \text{distance}(\text{home},\text{park})=8\}$
Limitation of Ordered-Task Planning

- Cannot interleave subtasks of different tasks
- Sometimes this can make things awkward
- Need methods that reason globally instead of locally
Generalize the Methods

- Generalize methods to allow the subtasks to be partially ordered
- Consequence: plans may interleave subtasks of different tasks

- This makes the planning algorithm more complicated
Generalize TFD to interleave subtasks

$$\pi = \{a_1, \ldots, a_k\}; \ w = \{t_1, t_2, t_3, \ldots\}$$

operator instance $a$

$$\pi = \{a_1, \ldots, a_k, \ a\}; \ w' = \{t_2, t_3, \ldots\}$$

method instance $m$

$$w = \{t_1, t_2, \ldots\}$$

$$w' = \{u_1, \ldots, u_k, t_2, \ldots\}$$
Generalize TFD to interleave subtasks

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

\[ w' = \{ u_1, \ldots, u_k, t_2, \ldots \} \]

- \( \delta(w, u, m, \sigma) \) has a complicated definition in the book. Here’s what it means:
  - We selected \( t_1 \) because it’s possible for \( t_1 \) to come first
  - We’re planning for \( t_1 \) under the assumption that it will come first
  - Insert ordering constraints to ensure it will come first
  - The same constraints also must apply to all subtasks of \( t_1 \)
Discussion

- PFD is sound and complete
- Can generalize in the same ways as TFD

- SHOP2: implementation of PFD-like algorithm + generalizations
  - Won one of the top four awards in the AIPS-2002 Planning Competition
  - Freeware, open source