Section 2.7.7
HTN Planning

Dana S. Nau
University of Maryland
Motivation

- For some planning problems, we may already have ideas for how to look for solutions

- Example: travel to a destination that’s far away:
  - Brute-force search:
    - 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 can we put such information into an actor?
Using Domain-Specific Information in an Actor

- Several ways to do it
  - Domain-specific algorithm
  - Refinement methods (RAE and SeRPE, Chapter 3)
  - HTN planning (SHOP, PyHop, Section 2.7.7)
  - Control rules (TLPAn, Section 2.7.8)
Total-Order HTN Planning

- Ingredients:
  - states and actions
  - tasks: activities to perform
  - HTN methods: ways to perform tasks

- Method format:
  - method-name(args)
    - Task: task-name(args)
    - Pre: preconditions
    - Sub: list of subtasks

- Two kinds of tasks
  - Primitive: name of an action
  - Compound: need to refine using methods

- Planning algorithm
  - use methods to refine each task into a set of subtasks
    - do it recursively
  - bottom level:
    - primitive tasks, i.e., actions

```
\[
\begin{align*}
&\text{task} \\
&\downarrow \\
&\text{method} \\
&\downarrow \\
&\begin{cases} 
&\text{task} \\
&s_0 \\
&s_1 \\
&s_2 \\
&s_3 \\
\end{cases}
\end{align*}
\]```
Total-Order HTN Planning

- Special case of refinement methods (Chapter 3)
  - Method body is a list of tasks, not a program

- Even with this restriction, Turing-complete
  - To encode loops
    - use recursion
  - To encode if-then-else
    - use multiple methods for a task
Total-Order HTN Planning

- HTN planning domain: a pair \((\Sigma, M)\)
  - \(\Sigma\): state-variable planning domain
  - \(M\): set of methods

- Planning problem: \((\Sigma, M, s_0, T)\)
  - \(T\): list of tasks \(\langle t_1, t_2, \ldots, t_k \rangle\)

- Solution: any executable plan that can be generated by applying
  - methods to nonprimitive tasks
  - actions to primitive tasks
**Simple Travel-Planning Problem**

- **Initial state:**
  - I’m at home,
  - I have $20,
  - there’s a park 8 miles away
  
    \[ \{\text{loc}(\text{me})=\text{home}, \]
    \[ \text{cash}(\text{me})=20, \]
    \[ \text{dist}(\text{home}, \text{park})=8, \]
    \[ \text{loc}(\text{taxi})=\text{elsewhere}\} \]

- **Task:** travel to the park
  
    \[ \text{travel}(\text{me}, \text{home}, \text{park}) \]

- **Methods**

  **travel-by-foot(}a,x,y)\**
  
  Task: \text{travel}(a,x,y)
  
  Pre: \text{loc}(a,x), \text{distance}(x, y) \leq 4
  
  Sub: \text{walk}(a, x, y)

  **travel-by-taxi(}a,x,y)\**
  
  Task: \text{travel}(a,x,y)
  
  Pre: \text{loc}(a,x),
  
  \[ \text{cash}(a) \geq 1.50 + \frac{1}{2} \text{dist}(x,y) \]
  
  Sub: \text{call-taxi} (a,x),
  
  \text{ride-taxi} (a,x,y),
  
  \text{pay-driver}(a)
**Simple Travel-Planning Problem**

- **Actions:**

  walk \((a,x,y)\)
  
  Pre: \(\text{loc}(a) = x\)
  
  Eff: \(\text{loc}(a) \leftarrow y\)

  call-taxi \((a,x)\)
  
  Pre: —
  
  Eff: \(\text{loc}(\text{taxi}) \leftarrow x\)

  ride-taxi \((a,x,y)\)
  
  Pre: \(\text{loc}(a) = x, \text{loc}(\text{taxi}) = x\)
  
  Eff: \(\text{loc}(a) \leftarrow y, \text{loc}(\text{taxi}) \leftarrow y, \text{owe}(a) \leftarrow 1.50 + \frac{1}{2} \text{dist}(x,y)\)

  pay-driver\((a,r)\)
  
  Pre: \(\text{owe}(a) = r, \text{cash}(a) \geq r\)
  
  Pre: \(\text{owe}(a) \leftarrow 0, \text{cash}(a) \leftarrow \text{cash}(a) - r\)

- **Parameters**

  - \(a \in \text{Agents}\)
  - \(x,y \in \text{Locations}\)
  - \(r \in \mathbb{R}\) (real numbers)
Simple Travel-Planning Problem

- Left-to-right backtracking search

![Travel Planning Diagram]

**Initial state**
- \( \text{loc}(\text{me}) = \text{home} \)
- \( \text{cash}(\text{me}) = 20 \)
- \( \text{dist}(\text{home, park}) = 8 \)
- \( \text{loc}(\text{taxi}) = \text{elsewhere} \)

**Call Taxi**
- Pre: \( \text{loc}(\text{me, home}) \)
- Effects: ...

**Ride Taxi**
- Pre: \( \text{loc}(\text{me, home}) \)
- Effects: ...

**Pay Driver**
- Pre: \( \text{loc}(\text{me}) = \text{park} \)
- Effects: ...

**Final state**
- \( \text{loc}(\text{me}) = \text{park} \)
- \( \text{cash}(\text{me}) = 14.5 \)
- \( \text{dist}(\text{home, park}) = 8 \)
- \( \text{loc}(\text{taxi}) = \text{park} \)
- \( \text{owe}(\text{me}) = 0 \)
Planning Algorithm

- TFD($\Sigma$, $M$, $s$, $T$
  - if $T = \langle \rangle$ then return $\langle \rangle$
  - let the tasks in $T$ be $t_1, t_2, \ldots, t_k$
  - if $t_1$ is primitive then
    - $Candidates \leftarrow \{a \mid \text{head}(a) \text{ matches } t_1 \text{ and } a \text{ is applicable in } s\}$
    - if $Candidates = \emptyset$ then return failure
    - nondeterministically choose any $a \in Act$
    - $\pi \leftarrow \text{TFD}(\Sigma, \gamma(s,a), \langle t_2, \ldots, t_k \rangle)$
    - if $\pi = \text{failure}$ then return failure
    - else return $a \cdot \pi$
  - else
    - $t_1$ is nonprimitive
    - $Candidates \leftarrow \{m \in M \mid \text{task}(m) \text{ matches } t_1 \text{ and } m \text{ is applicable in } s\}$
    - if $Candidates = \emptyset$ then return failure
    - nondeterministically choose any $a \in Act$
    - return TFD($\Sigma$, $M$, $s$, $\text{sub}(m) \cdot \langle t_2, \ldots, t_k \rangle$)
Pyhop

- A simple HTN planner written in Python
  - Works in both Python 2.7 and 3.2

- Implements the algorithm on the previous page
  - HTN operators and methods are ordinary Python functions
  - The current state is a Python object that contains variable bindings
    - Operators and methods refer to states explicitly
    - To say $c$ is on $a$, write $s\.loc['c'] = 'a'$ where $s$ is the current state

- Easy to implement and understand
  - Less than 150 lines of code

- Open-source software, Apache license
  - [http://bitbucket.org/dananau/pyhop](http://bitbucket.org/dananau/pyhop)
# Actions

**walk**\(a: Agents, x: Locations, y: Locations\)
- **Pre:** \(\text{loc}(a) = x\)
- **Eff:** \(\text{loc}(a) = y\)

**call-taxi**\(a: Agents, x: Locations\)
- **Pre:** —
- **Eff:** \(\text{loc}(\text{taxi}) = x\)

**ride-taxi**\(a: Agents, x: Locations, y: Locations\)
- **Pre:** \(\text{loc}(a) = x, \text{loc}(\text{taxi}) = x\)
- **Eff:** \(\text{loc}(a) = y, \text{loc}(\text{taxi}) = y, \text{owe}(a) = 1.50 + \frac{1}{2} \text{distance}(x,y)\)

**pay-driver**\(a: Agents\)
- **Pre:** \(\text{owe}(a) = r, \text{cash}(a) \geq r\)
- **Pre:** \(\text{owe}(a) = r, \text{cash}(a) = \text{cash}(a) - r\)

```python
def walk(state, a, x, y):
    if state.loc[a] == x:
        state.loc[a] = y
        return state
    else:
        return False

def call_taxi(state, a, x):
    state.loc['taxi'] = x
    return state

def ride_taxi(state, a, x, y):
    if state.loc['taxi'] == x and state.loc[a] == x:
        state.loc['taxi'] = y
        state.loc[a] = y
        state.owe[a] = 1.5 + 0.5 * state.dist[x][y]
        return state
    else:
        return False

def pay_driver(state, a):
    if state.cash[a] >= state.owe[a]:
        state.cash[a] = state.cash[a] - state.owe[a]
        state.owe[a] = 0
        return state
    else:
        return False
```

declare_operators(walk, call_taxi, ride_taxi, pay_driver)
Methods

travel-by-foot($a, x, y$)

Task: travel($a,x,y$)
Pre: $\text{loc}(a,x)$, $\text{distance}(x,y) \leq 4$
Sub: walk($a,x,y$)

def travel_by_foot(state,a,x,y):
    if state.dist[x][y] <= 4:
        return [('walk',a,x,y)]
    return False

def travel_by_taxi(state,a,x,y):
    if state.cash[a] >= 1.5 + 0.5*state.dist[x][y]:
        return [('call_taxi',a,x),
                ('ride_taxi',a,x,y),
                ('pay_driver',a,x,y)]
    return False

declare_methods('travel', travel_by_foot, travel_by_taxi)
Travel Planning Problem

Initial state:

loc(me) = home, cash(me) = 20, dist(home,park) = 8

state1 = State('state1')
state1.loc = {'me':'home'}
state1.cash = {'me':20}
state1.owe = {'me':0}
state1.dist = {'home':{'park':8}, 'park':{'home':8}}

Task:

call-taxi(me,home), ride-taxi(me,park), pay-driver(me)

Solution plan:

[('call_taxi', 'me', 'home'), ('ride_taxi', 'me', 'home', 'park'), ('pay_driver', 'me')]
Comparison to Forward and Backward Search

- In HTN planning, more possibilities than just forward or backward
  - A little like the choices to make in parsing algorithms
- SHOP, Pyhop, GDP, GoDeL:
  - down, then forward
    (like RAE and SeRPE)
- SIPE, O-Plan, UMCP
  - plan-space
    (down and backward)
- Angelic Hierarchical A*
  - use abstract actions to produce abstract states
  - forward A*, at the top level
  - forward A*, one level down
  - ...
HTN Planning in General

- Other formalisms and algorithms
  - Some of them use partially ordered tasks
    - Forward search (like IRT in Chapter 3) – SHOP2
    - Plan-space planning – SIPE, O-Plan, UMCP
      - These allow more constraints than just preconditions
        - postconditions, “during” conditions, etc.
  - Some of them use goals and subgoals instead of tasks and subtasks
    - Angelic Hierarchical A*
    - GDP, GoDeL
SHOP and SHOP2

- SHOP and SHOP2:
  - HTN planning systems
  - SHOP2 an award in the AIPS-2002 Planning Competition

- Instead of state variables, used “classical plus functions”

- Freeware, open source
  - Downloaded more than 20,000 times
  - Used in many hundreds of projects worldwide
    - Government labs, industry, academia
Bridge

- Ideal: game-tree search (all lines of play) to compute expected utilities
- Don’t know what cards other players have
  - Many moves they *might* be able to make
    - worst case about $6 \times 10^{44}$ leaf nodes
    - average case about $10^{24}$ leaf nodes
- About 1½ minutes available

```
Not enough time – need smaller tree
```

- **Bridge Baron**
  - 1997 world champion of computer bridge
  - Special-purpose HTN planner that generates game trees
    - Branches $\Leftrightarrow$ standard bridge card plays (finesse, ruff, cash out, …)
    - Much smaller game tree: can search it and compute expected utilities
- **Why it worked:**
  - Special-purpose planner to generate trees rather than linear plans
  - Lots of work to make the HTN methods as complete as possible
KILLZONE 2

- Special-purpose HTN planner for planning at the squad level
  - Method and operator syntax similar to SHOP’s and SHOP2’s
  - Quickly generates a linear plan that would work if nothing interferes
  - Replan several times per second as the world changes
- **Why it worked:**
  - Very different objective from a bridge tournament
  - Don’t *want* to look for the best possible play
  - Need actions that appear believable and consistent to human users
  - Need them very quickly