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, GTPyhop: Section 2.7.7)
  - Control rules (TLPlan: 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* task: name of an action
  - *Compound* task: need to *decompose* (or *refine*) using methods

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

- **Planning problem:** $P = (\Sigma, M, s_0, T)$
  - $T$: 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

- **Planning algorithm**
  - depth-first, left-to-right search
  - for each compound task, apply a method to decompose it into subtasks
  - for each primitive task, apply the action
Simple Travel-Planning Problem

• Action templates:

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

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

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

  pay-driver \((a, y)\)
  
  Pre: \(\text{owe}(a) \leq \text{cash}(a)\)
  
  Eff: \(\text{cash}(a) \leftarrow \text{cash}(a) - \text{owe}(a),\)
  
  \(\text{owe}(a) \leftarrow 0,\)
  
  \(\text{loc}(a) = y\)

• Action parameters
  
  \(a \in \text{Agents}\)
  
  \(x, y \in \text{Locations}\)
Simple Travel-Planning Problem

- **Initial state:**
  - I’m at home,
  - I have $20,
  - there’s a park 8 miles away
- $s_0 = \{\text{loc(me)=home, cash(me)=20, dist(home,park)=8, loc(taxi)=elsewhere}\}$

- **Task:** travel to the park
  - travel(me,home,park)

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

- **Method parameters**
  - $a \in \text{Agents}$
  - $x,y \in \text{Locations}$
Simple Travel-Planning Problem

- Left-to-right backtracking search

Solution plan: 
\{call-taxi(me,home), ride-taxi(me,home,park), pay-driver(me)\}

Initial state:
- loc(me) = home
- cash(me) = 20
- dist(home,park) = 8
- loc(taxi) = station

Precond: ...
Effects: ...

Final state:
- loc(me) = park
- cash(me) = 14.5
- dist(home,park) = 8
- loc(taxi) = park
- owe(me) = 0

Precond: ...
Effects: ...

Precond: ...
Effects: ...

Precond: ...
Effects: ...

Backtrack
Nondeterministic Planning Algorithm

- **find-plan**($s_0, T$)
  - return seek-plan($s_0, T, \langle \rangle$)

- **seek-plan**($s, T, \pi$)
  - if $T = \langle \rangle$ then return $\pi$
  - let $t_1, t_2, \ldots, t_k$ be the tasks in $T$
  - if $t_1$ is primitive then
    - if there is an action $a$ such that head($a$) matches $t_1$ and $a$ is applicable in $s$:
      - return seek-plan($\gamma(s,a), \langle t_2,\ldots,t_k \rangle, \pi . a$)
    - else: return failure
  - else  // $t_1$ is nonprimitive
    - Candidates ← $\{ m \in M \mid \text{task}(m) \text{ matches } t_1 \text{ and } m \text{ is applicable in } s \}$
    - if Candidates = $\emptyset$ then return failure
    - nondeterministically choose any $m \in$ Candidates
    - return seek-plan($s, \text{subtasks}(m). \langle t_2,\ldots,t_k \rangle, \pi$)
Depth-first Search Implementation

- find-plan($s_0$, $T$)
  - return seek-plan($s_0$, $T$, $\langle \rangle$)

- seek-plan($s$, $T$, $\pi$)
  - if $T = \langle \rangle$ then return $\pi$
  - let $t_1$, $t_2$, ..., $t_k$ be the tasks in $T$
  - if $t_1$ is primitive then
    - if there is an action $a$ such that
      head($a$) matches $t_1$ and $a$ is applicable in $s$:
        - return seek-plan($\gamma(s,a)$, $\langle t_2, ..., t_k \rangle$, $\pi.a$)
    - else: return failure
  - else // $t_1$ is nonprimitive
    - for each $m \in M$:
      - if task($m$) matches $t_1$ and $m$ is applicable in $s$ then
        - $\pi \leftarrow$ seek-plan($s$, subtasks($m$).$\langle t_2, ..., t_k \rangle$, $\pi$)
      - if $\pi \neq$ failure then return $\pi$
    - return failure

i.e., $T = \langle t_1, t_2, ..., t_k \rangle$

Apply action; call seek-plan

state $s$; $T = [a, t_2, ..., t_k]$
new state $\gamma(s,a)$; new $T = [t_2, ..., t_k]$

Apply method; call seek-plan

state $s$, $T = [\tau, t_2, ..., t_k]$ task method $m$
new $T = [u_1, ..., u_j, t_2, ..., t_k]$
Search Direction, Search Strategies

- Down, then forward
  - totally-ordered tasks: find-plan, SHOP, Pyhop
  - partially-ordered tasks: SHOP2, SHOP3
  - goals instead of tasks: GDP, GoDeL
  - acting, task refinement: RAE (Chap. 3)
  - Monte Carlo rollouts: UPOM

- Down and backward
  - plan-space planning: SIPE, O-Plan, UMCP

- Forward, then down (level 1, level 2, level 3, …)
  - AHA*: A* search
  - Bridge Baron 1997: game-tree generation
“First-person shooter” game, ≈ 2009
Special-purpose HTN planner for planning at the squad level
  - Method and action 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:
  - Need actions that appear believable and consistent to human users
  - Need them very quickly
Pyhop

- A simple HTN planner written in Python
  - Implements find_plan
  - Works in both Python 2.7 and 3.2
- Only 240 lines of code
  - ≈ 95 lines excluding comments and docstrings
- State: Python object that contains variable bindings
  - To say taxi is at park in state s, write
    - s.loc['taxi'] = 'park'
- Actions and methods: ordinary Python functions
- Some limitations compared to most other HTN planners
  - I’ll discuss later

- Open-source software, Apache license
  - http://bitbucket.org/dananau/pyhop
- Simple travel example
  - download Pyhop
  - import simple_travel_example
Actions (Pyhop Operators)

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

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

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

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

$a \in \text{Agents}; x, y \in \text{Location}$

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: \(\text{travel}(a, x, y)\)
  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: \(\text{cash}(a) \geq 1.5 + 0.5 \times \text{dist}(x, y)\)
  Sub: \(\text{call-taxi}(a, x), \text{ride-taxi}(a, x, y), \text{pay-driver}(a)\)

```python
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
def declare_methods('travel', travel_by_foot, travel_by_taxi)
```
Travel Planning Problem

Initial state:

\[
\text{loc}(\text{me}) = \text{home}, \quad \text{cash}(\text{me}) = 20, \quad \text{dist}(\text{home}, \text{park}) = 8
\]

\[
\begin{align*}
\text{state1} &= \text{State}('\text{state1}') \\
\text{state1}.\text{loc} &= \{'\text{me}':'\text{home}'\} \\
\text{state1}.\text{cash} &= \{'\text{me}':20\} \\
\text{state1}.\text{owe} &= \{'\text{me}':0\} \\
\text{state1}.\text{dist} &= \{'\text{home}':{'\text{park}':8}, \ '\text{park}':{'\text{home}':8}\}
\end{align*}
\]

Task:

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

\[
\text{pyhop(state1,[(}'\text{travel}', 'me', 'home', 'park')\'])
\]

Solution plan:

\[
\text{call-taxi}(\text{me}, \text{home}), \quad \text{ride-taxi}(\text{me}, \text{park}), \quad \text{pay-driver}(\text{me})
\]

\[
[(}'\text{call\_taxi}', 'me', 'home'), \ ('\text{ride\_taxi}', 'me', 'home', 'park'), \ ('\text{pay\_driver}', 'me')]
\]
• Methods in most HTN planners:
  ▶ method name and parameter list
  ▶ the task for which the method is relevant
  ▶ preconditions (logical expression)
  ▶ subtasks (list or partially ordered set)

Method m_transport\((r,x,c,y,z)\)

  Task: transport\((c,y,z)\)
  Pre: loc\((r) = x\), cargo\((r) = \text{nil}\), loc\((c) = y\)
  Sub: move\((r,x,y)\), take\((r,c,y)\), move\((r,y,z)\), put\((r,c,z)\)

• Can have method parameters that aren’t in the task
  ▶ Multiple instances may match a task
    ▶ e.g., different values of \(r\) and \(x\)
  ▶ If one leads to failure, backtrack and try another

Discussion (1)

def m_transport\((c,y,z)\):
  if loc\((r) == x\) and cargo\((r) == \text{nil}\) and loc\((c) == y\):
    \((r,x) = \text{find suitable robot}('\text{transport}',c,y,z)\)
    return [move\((r,x,y)\), take\((r,c,y)\), move\((r,y,z)\), put\((r,c,z)\)]
  else:
    return False

• Pyhop:
  ▶ Method’s parameter list = task’s parameter list

• Limitation:
  ▶ Only one method instance matches the task
  ▶ How to backtrack over multiple possibilities?
Discussion (2)

- Methods in most HTN planners:
  - method name and parameter list
  - the task for which the method is relevant
  - preconditions (logical expression)
  - subtasks (list or partially ordered set)

Method m_transport(r,x,c,y,z)
  Task: transport(c,y,z)
  Pre: loc(r) = x, cargo(r) = nil, loc(c) = y
  Sub: move(r,x,y), take(r,c,y), move(r,y,z), put(r,c,z)

- Subtasks are known in advance
  - Can reason about them to decide which method instance to use
  - e.g., heuristic functions

def m_transport(c,y,z):
    if loc(r) == x and cargo(r) == nil and loc(c) == y:
        (r,x) = find_suitable_robot('transport',c,y,z)
        return [move(r,x,y), take(r,c,y), move(r,y,z), put(r,c,z)]
    else:
        return False

- Pyhop:
  - Subtasks are produced by code execution
- Limitation: subtasks not known in advance
  - How to implement heuristic functions?
- Strength:
  - Easier to reason about what the subtasks should be
GTPyhop

- GTPyhop (2021):
  - Combines HTN planning with HGN (hierarchical goal network) planning
- Like Pyhop, but plans for both tasks and goals
  - declare some methods for accomplishing tasks
  - declare other methods for achieving goals
- Open-source: https://github.com/dananau/GTPyhop

- Mostly backward-compatible with Pyhop
  - Example in the GTPyhop software distribution:
    - Examples/pyhop_simple_travel_example
  - Near-verbatim version of the Pyhop simple travel example
    - Documentation tells what the differences are
GTPyhop pseudocode

- **find_plan**($s_0, T$)
  - return seek_plan($s_0, T, []$)

- **seek_plan**($s, T, \pi$)
  - if $T = []$ then return $\pi$
  - $t \leftarrow$ first element of $T$; $T' \leftarrow$ the rest of $T$
  - case($t$):
    - action:
      - return apply_action_and_continue($s, t, T', \pi$)
    - task:
      - return refine_task_and_continue($s, t, T', \pi$)
    - goal:
      - return refine_goal_and_continue($s, t, T', \pi$)

- **refine_task_and_continue**($s, \tau, T', \pi$)
  - $M \leftarrow \{\text{task methods that were declared relevant for } \tau\}$
  - for each $m \in M$ that is applicable in $s$:
    - $T_{sub} \leftarrow \text{decomp}(s, m)$
    - $\pi \leftarrow \text{seek_plan}(s, T_{sub} + T', \pi)$
    - if $\pi \neq \text{failure}$ then return $\pi$
  - return failure

- **apply_action_and_continue**($s, a, T', \pi$)
  - if $a$ is applicable in $s$
    - return seek_plan($\gamma(s,a), T', \pi + [a]$)
  - else return failure

- **Actions and tasks:**
  - Handled the same way as in Pyhop
  - (pseudocode is reorganized but still does the same thing)

state $s$; $T = [a, t_2, \ldots, t_k]$ new state $\gamma(s,a)$; new $T = [t_2, \ldots, t_k]$

state $s$, $T = [\tau, t_2, \ldots, t_k]$ task method $m$ new $T = [u_1, \ldots, u_j, t_2, \ldots, t_k]$
GTPyhop pseudocode

- **find_plan**($s_0, T$)
  - return **seek_plan**($s_0, T, []$)

- **seek_plan**($s, T, \pi$)
  - if $T = []$ then return $\pi$
  - $t \leftarrow$ first element of $T$; $T' \leftarrow$ the rest of $T$
  - **case**($t$):
    - **action**:
      - return **apply_action_and_continue**($s, t, T', \pi$)
    - **task**:
      - return **refine_task_and_continue**($s, t, T', \pi$)
    - **goal**:
      - return **refine_goal_and_continue**($s, t, T', \pi$)

- **Goal methods**:
  - Analogous to task methods but match goals instead
  - To ensure soundness:
    - Verify whether the goal is actually achieved

- **verify**($g$):
  - **Pre**: $g$
  - **Eff**: —

- **refine_goal_and_continue**($s, g, T', \pi$)
  - $M \leftarrow \{\text{goal methods that were declared relevant for } g\}$
  - for each $m \in M$ that is applicable in $s$:
    - $T_{sub} \leftarrow \text{decomp}(s, m) + [\text{verify}(g)]$
    - $\pi \leftarrow \text{seek_plan}(s, T_{sub} + T', \pi)$
    - if $\pi \neq \text{failure}$ then return $\pi$
  - return failure

- New $T = [g, t_2, \ldots, t_k, \text{verify}(g), t_2, \ldots, t_k]$

State $s$, $T = [g, t_2, \ldots, t_k]$

Task method $m$

New $T = [u_1, \ldots, u_j, \text{verify}(g), t_2, \ldots, t_k]$

- **Dummy action**
  - if $g$ is true in the current state, no effect
  - if $g$ is false, GTPyhop must backtrack

Nau – Lecture slides for Automated Planning and Acting
Example: Blocks World

Simple classical planning domain

- **pickup(x)**
  - pre: loc(x)=table, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x

- **putdown(x)**
  - pre: holding=x
  - eff: holding=nil, loc(x)=table, clear(x)=T

- **unstack(x,y)**
  - pre: loc(x)=y, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x, clear(y)=T

- **stack(x,y)**
  - pre: holding=x, clear(y)=T
  - eff: holding=nil, clear(y)=F, loc(x)=y, clear(x)=T

- The “Sussman anomaly”
  - Planning problem that caused problems for early classical planners

\[
\pi = \langle \text{unstack}(c,b), \text{putdown}(c), \text{pickup}(b), \text{stack}(b,c), \text{pickup}(a), \text{stack}(a,b) \rangle
\]

\[
s_0 = \{ \text{clear}(a)=\text{F}, \text{clear}(b)=\text{T}, \text{clear}(c)=\text{T}, \text{loc}(a)=\text{table}, \text{loc}(b)=\text{table}, \text{loc}(c)=a, \text{holding}(\text{hand})=\text{nil} \}
\]

\[
g = \{ \text{loc}(a)=b, \text{loc}(b)=c \}
\]
Domain-Specific Algorithm

**loop**

- if there’s a clear block that needs to be moved
  and it can immediately be moved to a place
  where it won’t need to be moved again
  then move it there

- else if there’s a clear block that needs to be moved
  then move it to the table

- else if the current state satisfies the goal
  then return success

- else return failure

- Situations in which \(c\) needs to be moved:
  - \(\text{loc}(c)=d\), goal contains \(\text{loc}(c)=e\), and \(d \neq e\)
  - \(\text{loc}(c)=d\), \(d\) is a block, goal contains \(\text{loc}(b)=d\), and \(b \neq c\)
  - \(\text{loc}(c)=d\) and \(d\) is a block that needs to be moved

- Can extend this to include situations involving clear and holding

\[
\pi = \langle \text{unstack}(c,b), \text{putdown}(c), \text{pickup}(b), \text{stack}(b,c), \text{pickup}(a), \text{stack}(a,b) \rangle
\]

\[
s_0 = \{\text{clear}(a)=F, \text{clear}(b)=T, \text{clear}(c)=T, \text{loc}(a)=\text{table}, \text{loc}(b)=\text{table}, \text{loc}(c)=a, \text{holding}(\text{hand})=\text{nil}\}
\]

\[
g = \{\text{loc}(a)=b, \text{loc}(b)=c\}
\]
Properties of the Algorithm

- Sound, complete, guaranteed to terminate on all block-stacking problems
- Runs in time $O(n^3)$
  - Can be modified (Slaney & Thiébaux) to run in time $O(n)$
- Often finds optimal (shortest) solutions
- But sometimes only near-optimal
  - For block-stacking problems, \texttt{PLAN-LENGTH} is NP-complete
- Can be implemented as GTPython methods
States and goals

- Sussman anomaly initial state:

  - A State object to hold all the state-variable bindings:

    ```python
    sus_s0 = gtpyhop.State('Sussman initial state')
    sus_s0.pos = {'a':'table', 'b':'table', 'c':'a'}
    sus_s0.clear = {'a':False, 'b':True, 'c':True}
    sus_s0.holding = {'hand':False}
    ```

    - Python dictionary notation for
      ```python
      sus_s0.pos['a'] = 'table', etc.
      ```

- Sussman anomaly goal:

  ```python
  sus_g = gtpyhop.Multigoal('Sussman goal')
  sus_g.pos = {'a':'b', 'b':'c'}
  ```

- Two kinds of goals:
  - **Unigoal**: a triple \((\text{name}, \text{arg}, \text{value})\)
    - represents a desired state-variable binding
    - e.g., unigoal \((\text{pos}, 'a', 'b')\)
      - find a state \(s\) in which \(s.\text{pos}[\text{a}] == 'b'\)
  - **Multigoal**: state-like object
    - represents a conjunction of unigoals
    - \(\text{sus_g}\): find a state \(s\) in which
      - \(s.\text{pos}[\text{a}] == 'b'\) and \(s.\text{pos}[\text{a}] == 'c'\)
Actions

- Blocks-world pickup action
  - if $x$ is on table, $x$ is clear, robot hand is empty
  - then modify $s$ and return it
  - else return nothing
    - Tells GTPyhop the action isn’t applicable
    - (could instead return false like Pyhop does)

- putdown action – similar

- Last line declares them to be actions

- Easy to write similar “stack” and “unstack” actions

```python
def pickup(s,x):
    if s.pos[x] == 'table' \
        and s.clear[x] == True \
        and s.holding['hand'] == False:
        s.pos[x] = 'hand'
        s.clear[x] = False
        s.holding['hand'] = x
    return s

def putdown(s,x):
    if s.holding['hand'] = x:
        s.pos[x] = 'table'
        s.clear[x] = True
        s.holding['hand'] = False
    return s

gtpyhop.declare_actions(pickup,putdown)
```
**Task methods**

- **m_take**: method to pick up a clear block $x$, regardless of what it’s on
  - Args: current state $s$, block $x$.
  - If $x$ is clear:
    - Return one to-do list if $x$ is on the table, another to-do list if $x$ isn’t on the table
  - Else return nothing
    - means method is inapplicable
    - (also OK to return false like Pyhop does)

- Last line says **m_take** is a task method
  - relevant for all tasks of the form (take, ...)

- **m_put**: similar, for all tasks of the form (put, ...)

```python
def m_take(s, x):
    if s.clear[x] == True:
        if s.pos[x] == 'table':
            return [('pickup', x)]
        else:
            return [('unstack', x, s.pos[x])]
    else:
        return []
gtpyhop.declare_task_methods('take', m_take)

def m_put(s, x, y):
    if s.holding['hand'] == x:
        if y == 'table':
            return [('putdown', x)]
        else:
            return [('stack', x, y)]
    else:
        return False
gtpyhop.declare_task_methods('put', m_put)
```
def m_moveblocks(s, mgoal):
    for x in all_clear_blocks(s):
        stat = status(x, s, mgoal)
        if stat == 'move-to-block':
            where = mgoal.pos[x]
            return [('take', x), ('put', x, where), mgoal]
        elif stat == 'move-to-table':
            return [('take', x), ('put', x, 'table'), mgoal]
            for x in all_clear_blocks(s):
                if status(x, s, mgoal) == 'waiting' and s.pos[x] != 'table':
                    return [('take', x), ('put', x, 'table'), mgoal]
            return []

gtpyhop.declare_multigoal_methods(m_moveblocks)

find_plan(sus_s0, sus_g)
returns [('unstack', 'c', 'a'), ('putdown', 'c'), ('pickup', 'b'), ('stack', 'b', 'c'), ('pickup', 'a'), ('stack', 'a', 'b')]

• s = current state
• mgoal = a multigoal
• red = helper functions

• s = current state
• mgoal = a multigoal
• red = helper functions
Discussion

● Earlier we discussed limitations/strengths of Pyhop compared to most other HTN planners
  ▸ Same discussion also applies to GTPyhop

● Similar comparison for GTPyhop vs. most HGN planners
Acting and Planning

- run_lazy_lookahead(state, todo_list)
  - loop:
    - plan = find_plan(state, todo_list)
    - if plan = []:
      - return state // the new current state
    - for each action in plan:
      - execute the corresponding command
      - if the command fails:
        - continue the outer loop

- Simple Travel Problem:
  - run_lazy_lookahead calls
    - find_plan(s_0, [(travel,me,home,park)])
  - find_plan returns
    - [(call_taxi,me,home),
      (ride_taxi,me,home,park),
      (pay_driver,me)]
  - run_lazy_lookahead executes
    - c_call_taxi(me,home)
    - c_ride_taxi(me,home,park)
    - c_pay_driver(me)

- If everything executes correctly, I get to the park
  - But suppose the taxi breaks down …
Acting and Planning

- For planning and acting, need to HTN methods that can recover from unexpected problems

- Example:
  - run_lazy_lookahead executes
    - c_call_taxi(me,home)
    - c_ride_taxi(me,home,park)
      - Suppose the 2nd command fails
  - run_lazy_lookahead calls
    - find_plan($s_1$, [(travel,me,home,park)])
      - Error: tries to use an undefined value

- To run this example in GTPyhop:
  - import Examples.simple_htn_acting_error

**Current State**

\[ s_0 \]
- c_call_taxi(me,home)
- loc(me) = home
- cash(me) = 20
- owe(me) = 0
- dist(home,park) = 8

\[ s_1 \]
- c_ride_taxi(me,home,park)
- loc(me) = taxi
- cash(me) = 20
- owe(me) = 0
- dist(home,park) = 8
- loc(taxi) = home

**Error Case**

- (travel,me,home,park)
- (travel_by_foot, me, home, park)
- Pre:
  - ✓ loc(me,taxi)
  - X dist(taxi,park) undefined

**Program Error**

- Command failure, current state is still $s_1$
Summary

- Total-order HTN planning
  - Planning problem: initial state, list of tasks
  - Apply HTN methods to tasks to get subtasks (smaller tasks)
    - Do this recursively to get smaller and smaller subtasks
      - At the bottom: primitive tasks that correspond to actions
  - Search goes down and forward

- Pyhop: Python implementation of total-order HTN planning
  - Open source: [http://bitbucket.org/dananau/pyhop](http://bitbucket.org/dananau/pyhop)

- GTPyhop: Python implementation of HTN + HGN planning
  - Open source: [https://github.com/dananau/GTPyhop](https://github.com/dananau/GTPyhop)

- Examples: simple travel, blocks world

- To integrate planning and acting, need to make sure the HTN methods can handle unexpected events
  - One way: make GTPyhop re-entrant