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, GNPyhop: 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

\[ s_0 = \{ \text{loc(me)=home,} \]
\[ \quad \text{cash(me)=20,} \]
\[ \quad \text{dist(home,park)=8,} \]
\[ \quad \text{loc(taxi)=elsewhere} \} \]

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

Methods:

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

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

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

- Left-to-right backtracking search

**Solution plan:** \( \langle \text{call-taxi}(\text{me}, \text{home}), \text{ride-taxi}(\text{me}, \text{home}, \text{park}), \text{pay-driver}(\text{me}) \rangle \)
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 $t_1$, $t_2$, ..., $t_k$ be the tasks in $T$  
    i.e., $T = \langle t_1, t_2, ..., t_k \rangle$
  - 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
    - $Candidates \leftarrow \{ m \in M \mid \text{task}(m) \text{ matches } t_1 \text{ and } m \text{ is applicable in } s \}$
    - if $Candidates = \emptyset$ then return failure
    - nondeterministically choose any $m \in Candidates$
    - return **seek-plan**($s$, 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 $t_1$, $t_2$, $\ldots$, $t_k$ be the tasks in $T$  
    i.e., $T = \langle t_1, t_2, \ldots, t_k \rangle$
  - 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($\gamma(s,a)$, $\langle t_2, \ldots, t_k \rangle$, $\pi \cdot a$)
    - 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($m$) $\cdot \langle t_2, \ldots, t_k \rangle$, $\pi$)
        - if $\pi \neq$ failure then return $\pi$
      - return failure
Comparison to Forward and Backward Search

- More possibilities than just forward or backward
  - A little like the choices to make in parsing algorithms

- SHOP, Pyhop, (total-order HTN planning), SHOP2 (partial-order HTN planning), GDP, GoDeL (HGN planning), RAE (refinement acting, Chap. 3):
  - down, then forward

- SIPE, O-Plan, UMCP
  - plan-space HTN planning (down and backward)

- AHA*
  - search in layers:
    - forward A*, at the top level
    - forward A*, one level down
    - …
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 ⇔ standard bridge card plays (finesse, ruff, cash out, …)
  - Much smaller game tree: can search the entire tree, 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

- "First-person shooter" game, ≈ 2009
- 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
SHOP, SHOP2, SHOP3

- SHOP (released 1999)
  - Uses the algorithm I showed you
  - Instead of state variables, “classical, plus functions”
  - Method and operator syntax based on Lisp
- SHOP2 (released 2001)
  - Allows partially-ordered tasks
  - Won an award in the AIPS-2002 Planning Competition
- Freeware, open source
  - As of Feb 2013, downloaded more than 20,000 times
  - Has been used in many projects worldwide
- SHOP3 (developed at SIFT, LLC, released 2019)
Pyhop and GNPyhop

- Pyhop: a simple HTN planner written in Python
  - Released in 2013
- Planning algorithm is like the one in SHOP, except:
  - 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 \( \text{c is on a} \), write \( s.\text{loc['c']} = 'a' \) where \( s \) is the current state
  - Easy to implement and understand
    - 240 lines
    - \( \approx 95 \) excluding comments and docstrings
- Open-source: [http://bitbucket.org/dananau/pyhop](http://bitbucket.org/dananau/pyhop)

- GNPyhop: enhanced version of Pyhop
- Main differences:
  - Can plan for both tasks and goals
  - Can hold multiple planning domains in memory at the same time
    - Give a different name to each one
  - \( \approx 5 \) times as large as Pyhop
- Open-source: pending
  - (will post link when U of Md approves open-source license)
GNPyhop (tasks)

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

- **seek_plan**\((s, T, \pi)\)
  - if \(T = [ ]\) then return \(\pi\)
  - let \(t_1, t_2, \ldots, t_k\) be the tasks/goals/multigoals in \(T\)
  - if \(t_1\) is an action:
    - return **apply_action**\((s, t_1, [t_2,\ldots,t_k], \pi)\)
  - else if \(t_1\) is a task:
    - return **find_task_method**\((s, t_1, [t_2,\ldots,t_k], \pi)\)
  - else if \(t_1\) is a goal:
    - return **find_goal_method**\((s, t_1, [t_2,\ldots,t_k], \pi)\)
  - else if \(t_1\) is a multigoal:
    - return **find_multigoal_method**\((s, t_1, [t_2,\ldots,t_k], \pi)\)
  - else error

- **apply_action**\((s, a, [t_2,\ldots,t_k], \pi)\)
  - if \(a\) is applicable in \(s\):
    - return **seek_plan**\((\gamma(s,a), [t_2,\ldots,t_k], \pi . a)\)
  - else return failure

- **find_task_method**\((s, t, [t_2,\ldots,t_k], \pi)\)
  - for every task method \(m\) such that \(\text{name}(t)\) matches \(\text{taskname}(m)\) and \(m\) is applicable to \(t\) in \(s\):
    - \(\pi \leftarrow \text{seek_plan}(s, \text{subtasks}(m) . [t_2,\ldots,t_k], \pi)\)
    - if \(\pi \neq \text{failure}\) then return \(\pi\)
  - return failure

\[
\begin{align*}
\text{state } s, \text{ action } a, T &= [t_2, \ldots, t_k] \\
\text{state } \gamma(s,a); T &= [t_2, \ldots, t_k] \\
\text{state } s, \text{ task } t, T &= [t_2, \ldots, t_k] \\
\text{method } m \\
\text{state } s; T &= [u_1, \ldots, u_j, t_2, \ldots, t_k]
\end{align*}
\]
GNPyhop (goals)

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

- **seek_plan**($s, T, \pi$)
  - if $T = [ ]$ then return $\pi$
  - let $t_1, t_2, \ldots, t_k$ be the tasks/goals/multigoals in $T$
    - if $t_1$ is an action:
      - return apply_action($s, t_1, [t_2, \ldots, t_k], \pi$)
    - else if $t_1$ is a task:
      - return find_task_method($s, t_1, [t_2, \ldots, t_k], \pi$)
    - else if $t_1$ is a goal:
      - return find_goal_method($s, t_1, [t_2, \ldots, t_k], \pi$)
    - else if $t_1$ is a multigoal:
      - return find_multigoal_method($s, t_1, [t_2, \ldots, t_k], \pi$)
    - else error

**multigoal**: a data structure that represents a conjunction of goals

- **find_goal_method**($s, g, T, \pi$)
  - if $s \models g$ then return $\pi$
  - for every goal method $m$ such that name($g$) matches goalname($m$) and $m$ is applicable to $g$ in $s$:
    - $\pi \leftarrow$ seek_plan($s$, subtasks($m$) + verify($g$) + $T$, $\pi$)
    - if $\pi \neq$ failure then return $\pi$
  - return failure

- **find_multigoal_method**($s, g, T, \pi$)
  - if $s \models g$ then return $\pi$
  - for every multigoal method $m$ that is applicable to $g$ in $s$:
    - $\pi \leftarrow$ seek_plan($s$, subtasks($m$) + verify($g$) + $T$, $\pi$)
    - if $\pi \neq$ failure then return $\pi$
  - return failure

optional

$g = (\text{name}, \text{arg}, \text{value})$

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

state $s; T = [u_1, \ldots, u_j, t_2, \ldots, t_k]$
GNPyhop version of the Simple Travel Problem

- Launch Python 3; load `simple_tasks1.py`

### GNPypihop 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($a, x, y$)

```python
def travel_by_foot(state, a, x, y):
    if state.dist[x][y] <= 4:
        return [('walk', a, x, y)]
gnpyhop.declare_task_methods('travel', travel_by_foot)
```

#### travel-by-taxi($a, x, y$)
- **Task:** travel($a, x, y$)
- **Pre:** $\text{cash}(a) \geq 1.5 + 0.5*\text{dist}(x, y)$
- **Sub:** call-taxi ($a, x$),
  ride-taxi ($a, x, y$),
  pay-driver($a$)

```python
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)]
gnpyhop.declare_task_methods('travel', travel_by_taxi)
```
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: \(\text{cash}(a) \leftarrow \text{cash}(a) - \text{owe}(a), \text{owe}(a) \leftarrow 0, \text{loc}(a) = y\)

gnpyhop.declare_actions(walk, call_taxi, ride_taxi, pay_driver)
Travel Planning Problem

Initial state:

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

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

Task:

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

\[
\text{gnpyhop}\text{.find_plan(state1,[(‘travel’,’me’,’home’,’park’)])}
\]

Solution plan:

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

\[
[(\text{‘call\_taxi’, ’me’, ’home’}), (\text{‘ride\_taxi’, ’me’, ’home’, ’park’}), (\text{‘pay\_driver’, ’me’})]
\]

To run this example in GNPyhop:

\[
\text{import simple_tasks1.py}
\]
Travel-Planning Problem

- Left-to-right backtracking search

\[
\begin{align*}
\text{travel} & \text{by foot}(\text{me, home, park}) \\
\text{Pre:} & \quad \checkmark \ \text{loc}(\text{me, home}) \\
& \quad \times \ \text{dist}(\text{home, park}) \leq 4
\end{align*}
\]

\[
\begin{align*}
\text{travel} & \text{by taxi}(\text{me, home, park}) \\
\text{Pre:} & \quad \checkmark \ \text{loc}(\text{me, home}) \\
& \quad \checkmark \ \text{cash}(\text{me}) \geq 1.5 + 0.5 \times \text{dist}(\text{home, park})
\end{align*}
\]

\[
\begin{align*}
\text{loc}(\text{me}) & = \text{taxi} \\
\text{cash}(\text{me}) & = 20 \\
\text{dist}(\text{home, park}) & = 8 \\
\text{loc}(\text{taxi}) & = \text{home}
\end{align*}
\]

\[
\begin{align*}
\text{loc}(\text{me}) & = \text{park} \\
\text{cash}(\text{me}) & = 14.5 \\
\text{dist}(\text{home, park}) & = 8 \\
\text{loc}(\text{taxi}) & = \text{park}
\end{align*}
\]

\[
\begin{align*}
\text{owe}(\text{me}) & = 5.50 \\
\text{loc}(\text{me}) & = \text{park} \\
\text{cash}(\text{me}) & = 14.5 \\
\text{dist}(\text{home, park}) & = 8 \\
\text{loc}(\text{taxi}) & = \text{park} \\
\text{owe}(\text{me}) & = 0
\end{align*}
\]

Solution plan: \([(\text{call_taxi, me, home}), (\text{ride_taxi, me, home, park}), (\text{pay_driver, me})]\)
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(s₀, [(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

- `run_lazy_lookahead` calls `find_plan(s₀, [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)`
- Suppose `c_ride_taxi(me, home, park)` fails:

  - Next, `run_lazy_lookahead` calls `find_plan(s₁, [(travel, me, home, park)])`
    - `find_plan(s₁, [(travel, me, home, park)])` returns
      - `[(travel_by_foot, me, home, park)]`
      - `loc(me, taxi)` undefined
      - `dist(taxi, park)` undefined
      - Program error

- To run this example in GNPyhop:
  - `import simple_tasks2.py`

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

\[s₀\]

\[s₁\]

- `loc(me) = home`
- `cash(me) = 20`
- `owe(me) = 0`
- `dist(home, park) = 8`
- `loc(taxi) = station`

- `loc(me) = taxi`
- `cash(me) = 20`
- `owe(me) = 0`
- `dist(home, park) = 8`
- `loc(taxi) = home`

- Command failure, state \[s₁\] unchanged
Motivation

- Sometimes we can write highly efficient planning algorithms for a specific domain
  - Use special properties of the domain
- Example: the “blocks world”

$\text{pickup}(x)$
  - $\text{pre: } \text{loc}(x)=\text{table}, \text{clear}(x)=\text{T}, \text{holding}=\text{nil}$
  - $\text{eff: } \text{loc}(x)=\text{hand}, \text{clear}(x)=\text{F}, \text{holding}=x$

$\text{putdown}(x)$
  - $\text{pre: } \text{holding}=x$
  - $\text{eff: } \text{holding}=\text{nil}, \text{loc}(x)=\text{table}, \text{clear}(x)=\text{T}$

$\text{stack}(x, y)$
  - $\text{pre: } \text{holding}=x, \text{clear}(y)=\text{T}$
  - $\text{eff: } \text{holding}=\text{nil}, \text{clear}(y)=\text{F}, \text{loc}(x)=y, \text{clear}(x)=\text{T}$

$\text{unstack}(x, y)$
  - $\text{pre: } \text{loc}(x)=y, \text{clear}(x)=\text{T}, \text{holding}=\text{nil}$
  - $\text{eff: } \text{loc}(x)=\text{hand}, \text{clear}(x)=\text{F}, \text{holding}=x, \text{clear}(y)=\text{T}$

clear(a)=F, clear(b)=T, clear(c)=T, clear(d)=F, clear(e)=T, loc(a)=table, loc(b)=table, loc(c)=a, loc(d)=table, loc(e)=d, holding=nil

clear(a)=T, clear(b)=F, clear(c)=T, clear(d)=F, clear(e)=T, loc(a)=b, loc(b)=table, loc(c)=d, loc(d)=table, loc(e)=table, holding=nil
The Blocks World

- For block-stacking problems with n blocks, easy to get a solution of length $O(n)$
  - Move all blocks to the table, then build up stacks from the bottom

- With more domain knowledge, can do even better

**pickup**($x$)
  
  **pre:** $\text{loc}(x)=$table, $\text{clear}(x)=$T, $\text{holding}=$nil
  
  **eff:** $\text{loc}(x)=$hand, $\text{clear}(x)=$F, $\text{holding}=$x

**putdown**($x$)
  
  **pre:** $\text{holding}=$x
  
  **eff:** $\text{holding}=$nil, $\text{loc}(x)=$table, $\text{clear}(x)=$T

**stack**($x,y$)
  
  **pre:** $\text{holding}=$x, $\text{clear}(y)=$T
  
  **eff:** $\text{holding}=$nil, $\text{clear}(y)=$F, $\text{loc}(x)=$y, $\text{clear}(x)=$T

**unstack**($x,y$)
  
  **pre:** $\text{loc}(x)=$y, $\text{clear}(x)=$T, $\text{holding}=$nil
  
  **eff:** $\text{loc}(x)=$hand, $\text{clear}(x)=$F, $\text{holding}=$x, $\text{clear}(y)=$T

For block-stacking problems with n blocks, easy to get a solution of length $O(n)$

- Move all blocks to the table, then build up stacks from the bottom

- With more domain knowledge, can do even better

**pickup**($x$)
  
  **pre:** $\text{loc}(x)=$table, $\text{clear}(x)=$T, $\text{holding}=$nil
  
  **eff:** $\text{loc}(x)=$hand, $\text{clear}(x)=$F, $\text{holding}=$x

**putdown**($x$)
  
  **pre:** $\text{holding}=$x
  
  **eff:** $\text{holding}=$nil, $\text{loc}(x)=$table, $\text{clear}(x)=$T

**stack**($x,y$)
  
  **pre:** $\text{holding}=$x, $\text{clear}(y)=$T
  
  **eff:** $\text{holding}=$nil, $\text{clear}(y)=$F, $\text{loc}(x)=$y, $\text{clear}(x)=$T

**unstack**($x,y$)
  
  **pre:** $\text{loc}(x)=$y, $\text{clear}(x)=$T, $\text{holding}=$nil
  
  **eff:** $\text{loc}(x)=$hand, $\text{clear}(x)=$F, $\text{holding}=$x, $\text{clear}(y)=$T
Block-Stacking Algorithm

loop
  if \( \exists \) a clear block \( c \) that needs moving
    & we can move \( c \) to a position \( d \)
    where it won’t need to be moved again
    then move \( c \) to \( d \)
  else if \( \exists \) a clear block \( c \) that needs to be moved
    then move \( c \) to the table
  else if the goal is satisfied
    then return success
  else return failure
repeat

- Cases in which \( c \) needs to be moved:
  - \( s \) contains \( \text{loc}(c)=d \) and \( g \) contains \( \text{loc}(c)=e \), where \( d \neq e \)
  - \( s \) contains \( \text{loc}(c)=d \) and \( g \) contains \( \text{loc}(b)=d \), where \( b \neq c \) and \( d \neq \text{table} \)
  - \( s \) contains \( \text{loc}(c)=d \) and \( d \) needs to be moved

\[ s_0: \quad \begin{array}{c}
  \text{e} \\
  \text{c} \\
  \text{d} \\
  \text{a} \\
  \text{b}
\end{array} \quad g: \quad \begin{array}{c}
  \text{a} \\
  \text{b} \\
  \text{c} \\
  \text{d} \\
  \text{e}
\end{array} \]

\( \langle \text{unstack(e,a)}, \text{putdown(e)}, \text{unstack(d,c)}, \text{stack(d,e)}, \text{unstack(c,b)}, \text{putdown(c)}, \text{pickup(b)}, \text{stack(b,c)}, \text{pickup(a)}, \text{stack(a,b)} \rangle \)
Properties of the Algorithm

● Sound, complete, guaranteed to terminate on all block-stacking problems

● Runs in time $O(n^3)$
  ▶ Can be modified (Slaney & Thiebaux) to run in time $O(n)$

● Often finds optimal (shortest) solutions, but sometimes only near-optimal
  ▶ For block-stacking problems, the question
    “does there exist a solution of length $\leq k$?”
    is NP-complete

● Some ways to implement it:
  ▶ As a domain-specific algorithm
  ▶ Using HTN planning (SHOP, PyHop - Section 2.7.7)
  ▶ Using refinement methods (RAE and SeRPE - Chapter 3)
  ▶ Using control rules (Section 2.7.8)

● To run it in GNPyhop:
  ▶ import blocks_tasks
GNPyhop Implementation

- task (move_blocks, g)
- method m_moveb(s, g)
  - if ∃ a clear block c that needs moving, and we can move c to a location d where it won’t need to be moved again
  - then return [(move_one, c, d), (move_blocks, g)]
  - else if ∃ a clear block c that needs to be moved
  - then return [(move_one, c, table), (move_blocks, g)]
  - else if s satisfies g then return []
  - else return False
- task (move_one, c, d)
  - methods that reduce it to
    - pickup(c) or unstack(c, d) followed by
    - putdown(c) or stack(c, d)

Cases in which c needs to be moved:

- s contains loc(c)=d and g contains loc(c)=e, where d ≠ e
- s contains loc(c)=d and g contains loc(b)=d, where b ≠ c and d ≠ table
- s contains loc(c)=d and d needs to be moved

$S_0$:  

\begin{align*}
\text{g:} & \\
\text{[(unstack,e,a), (putdown,e), (unstack,d,c), (stack,d,e), (unstack,c,b), (putdownc), (pickup,b), (stack,b,c), (pickup,a), (stack,a,b)]}
\end{align*}
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

- Unlike most HTN planners, Pyhop and GNPyhop use state-variable representation
  - Makes it easier to integrate them into ordinary programming
  - Written in Python
  - Open source
    - Pyhop at [http://bitbucket.org/dananau/pyhop](http://bitbucket.org/dananau/pyhop)
    - GNPyhop: (will post link when U of Md approves open-source license)

- Examples: simple travel, blocks world

- To integrate planning and acting, need to make sure the HTN methods can handle unexpected events