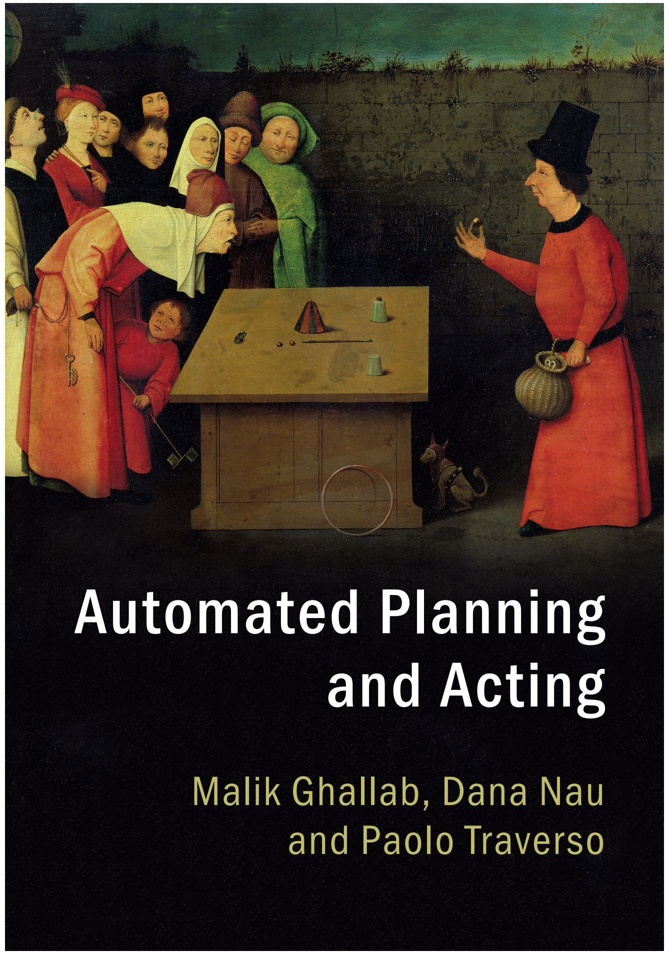
Section 2.7.7
HTN Planning

Dana S. Nau
University of Maryland



http://www.laas.fr/planning

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

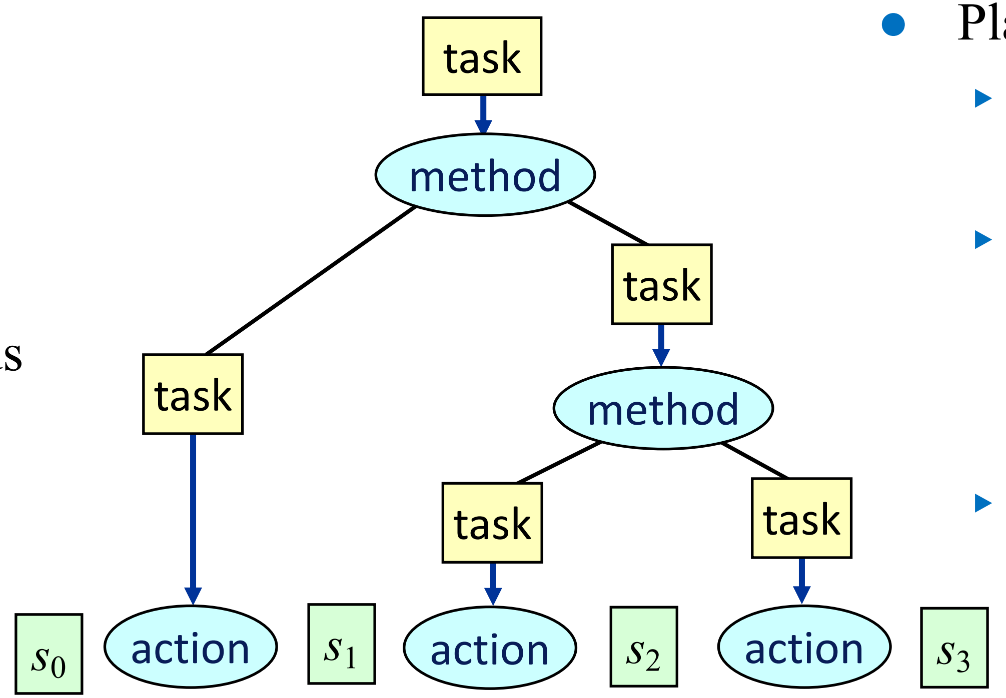
- Several ways to do it
  - Domain-specific algorithm
  - Refinement methods (RAE and SeRPE: Chapter 3)
  - HTN planning (SHOP, PyHop 2: Section 2.7.7)
  - Control rules (TLPlan: Section 2.7.8)

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 (Σ, M)
  - Σ: state-variable planning domain (states, actions)
  - M: set of methods

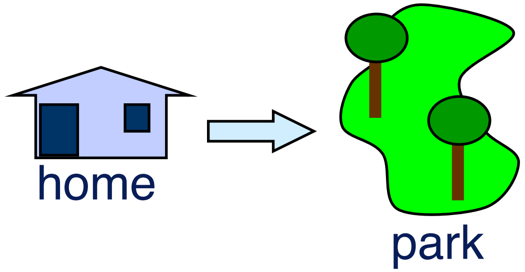
- Planning problem: $P = (\Sigma, M, s_0, T)$
  - T: list of tasks $\langle t_1, t_2, \ldots, t_k \rangle$
- Solution: any executable plan that can be generated by applying
  - methods to nonprimitive tasks
  - actions to primitive tasks



- 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: loc$(a) = x$
  Eff: loc$(a) \leftarrow y$

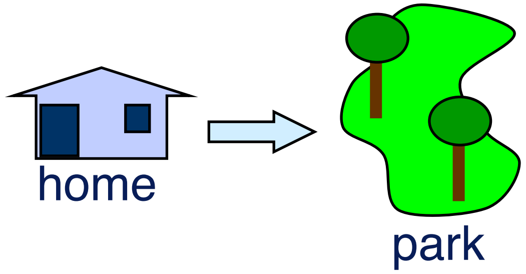
call-taxi (a,x)
  Pre: —
  Eff: loc(taxi) $\leftarrow x$,
    loc$(a) \leftarrow$ taxi

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

pay-driver(a,y)
  Pre: owe$(a) \leq$ cash(a)
  Eff: cash$(a) \leftarrow$ cash(a) – owe(a),
    owe$(a) \leftarrow 0$,
    loc$(a) = y$

- Action parameters
  - $a \in Agents$
  - $x,y \in Locations$

Simple Travel-Planning Problem



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

- *Methods*:

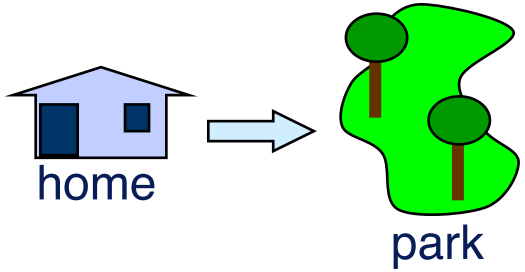
  travel-by-foot(a,x,y)
  Task: travel(a,x,y)
  Pre: loc(a,x), distance(x, y) ≤ 4
  Sub: walk(a,x,y)

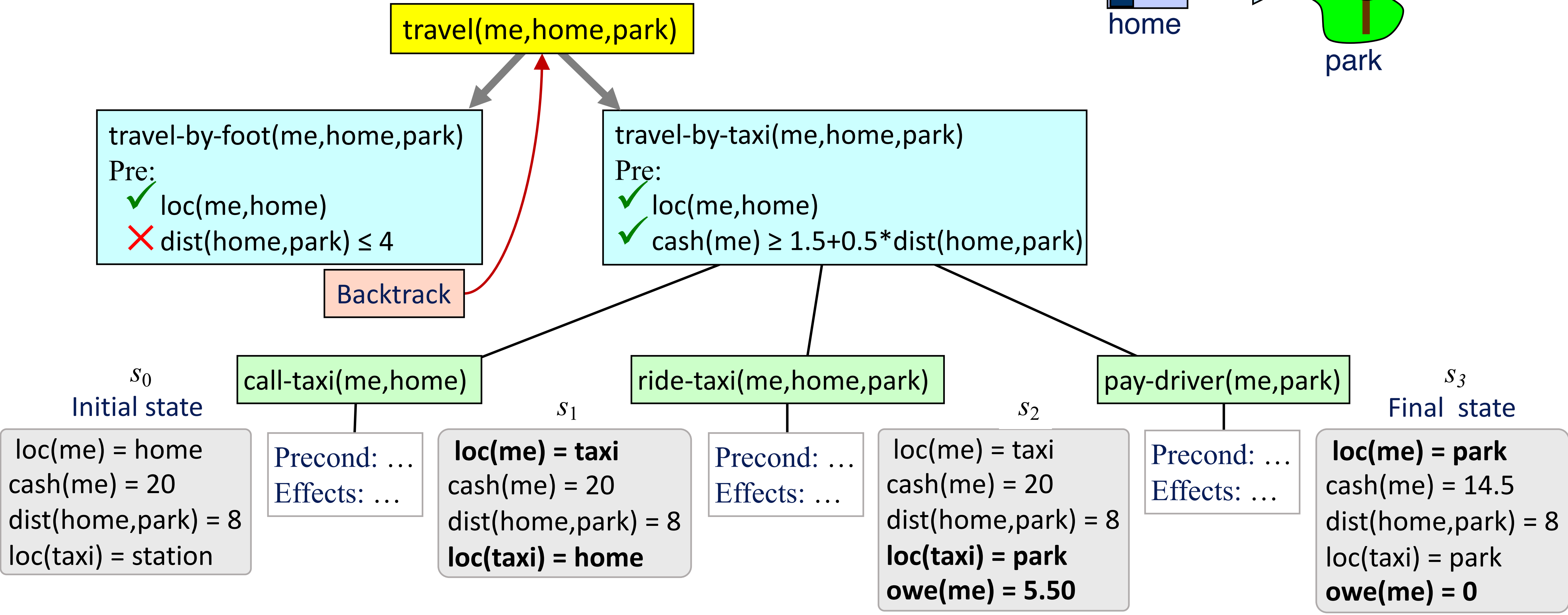
  travel-by-taxi(a,x,y)
  Task: travel(a,x,y)
  Pre: loc(a,x),
  cash(a) $\geq 1.50 + ½$ dist(x,y)
  Sub: call-taxi (a,x),
  ride-taxi (a,x,y),
  pay-driver(a,y)

- Method parameters
  - $a \in Agents$
  - $x,y \in Locations$

Simple Travel-Planning Problem

- Left-to-right backtracking search





Solution plan: ⟨call-taxi(me,home), ride-taxi(me,home,park), pay-driver(me)⟩

Nondeterministic Planning Algorithm

- find-plan(s_0, T)
  - return seek-plan(s_0, T, $\langle\rangle$)
- seek-plan(s, T, π)
  - if $T = \langle\rangle$ then return π
  - 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* $\leftarrow \{m \in M \mid$ task(m) matches t_1 and m is applicable in $s\}$
    - if *Candidates* = ∅ then return failure
    - nondeterministically choose any $m \in$ *Candidates*
    - return seek-plan(s, subtasks(m) . $\langle t_2,\ldots,t_k\rangle$, π)

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

state s, task list $T=\langle t_1, t_2, \ldots, t_k\rangle$
action a

state $\gamma(s,a)$, task list $T=\langle t_2, \ldots, t_k\rangle$

state s, task list $T=\langle t_1, t_2,\ldots, t_k\rangle$
method instance m

state s, task list $T=\langle u_1, \ldots, u_j, t_2, \ldots, t_k\rangle$

Depth-first Search Implementation

- find-plan(s_0, T)
  - return seek-plan(s_0, T, $\langle\rangle$)
- seek-plan(s, T, π)
  - if $T = \langle\rangle$ then return π
  - 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($\gamma(s,a)$, $\langle t_2,\ldots,t_k \rangle$, $\pi . a$)
    - 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) . $\langle t_2,\ldots,t_k \rangle$, π)
        - if $\pi \neq$ failure then return π
    - return failure

state s, task list $T=\langle t_1, t_2, \ldots, t_k \rangle$

action a

state $\gamma(s,a)$, task list $T=\langle t_2, \ldots, t_k \rangle$

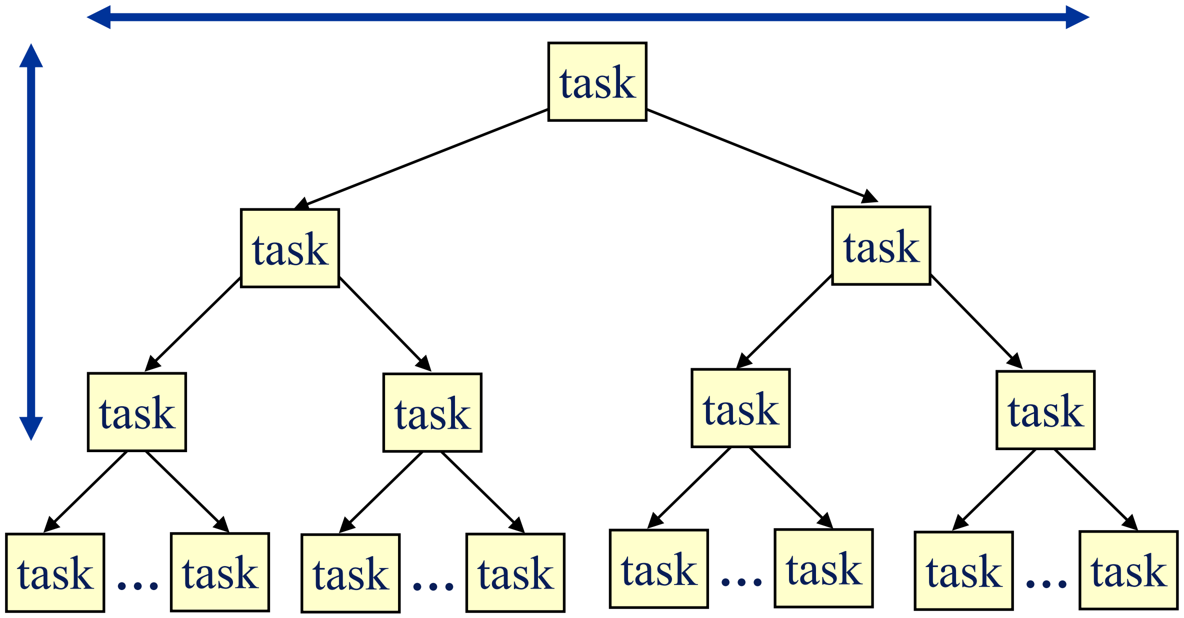
state s, task list $T=\langle t_1, t_2, \ldots, t_k \rangle$

method instance m

state s, task list $T=\langle u_1, \ldots, u_j, t_2, \ldots, t_k \rangle$

Comparison to Forward and Backward Search

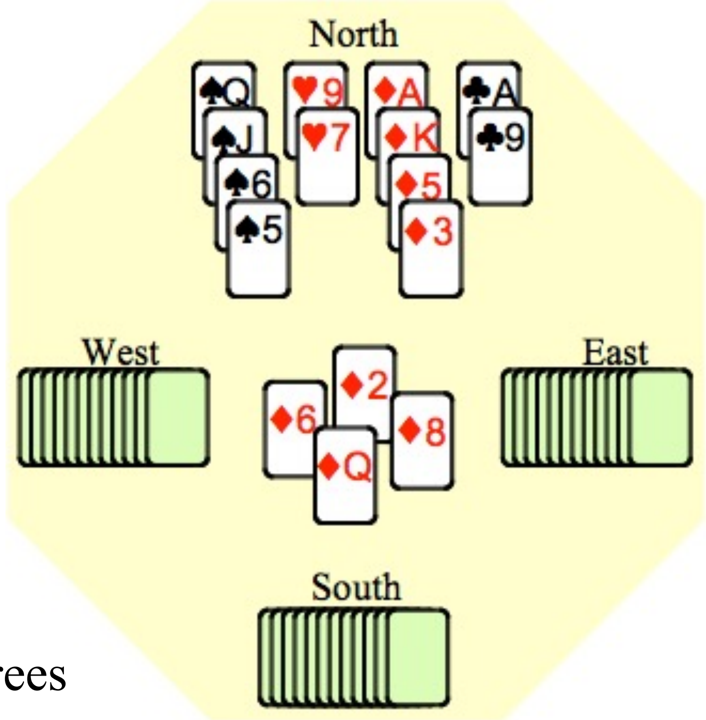
- More possibilities than just forward or backward
    - A little like the choices to make in parsing algorithms
- SHOP, Pyhop, (total-order HTN planning),
  SHOP2 (partial-order HTN planning),
  GDP, GoDeL (HGN planning),
  RAE (refinement acting, Chap. 3):
    - down, then forward
- SIPE, O-Plan, UMCP
    - 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 6×10^{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

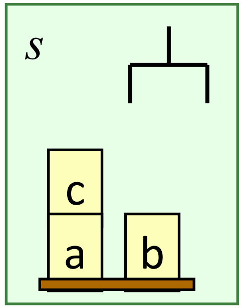
- “First-person shooter” game, ≈ 2009
- Special-purpose HTN planner for planning at the squad level
  - Method and operator syntax similar to SHOP’s and SHOP2’s
  - Quickly generates a linear plan that would work if nothing interferes
  - Replan several times per second as the world changes
- **Why it 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 Pyhop 2

- 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 `c` is on `a`, write `s.loc['c'] = 'a'` where `s` is the current state
  - Easy to implement and understand
    - 240 lines
    - ≈ 95 excluding comments and docstrings
- Open-source: http://bitbucket.org/dananau/pyhop



- Pyhop 2: 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
  - ≈ 5 times as large as Pyhop
- Open-source: pending
  - (will post link when U of Md approves open-source license)

Pyhop 2 (tasks)

state s, action a, $T = [t_2, \ldots, t_k]$

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

- find_plan(s_0, T)
  - return seek_plan(s_0, T, [ ])
- seek_plan(s, T, π)
  - if $T = [\,]$ then return π
  - 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, $[t_2,\ldots,t_k]$, π)
  - else if t_1 is a task:
    - return find_task_method(s, t_1, $[t_2,\ldots,t_k]$, π)
  - else if t_1 is a goal:
    - return find_goal_method(s, t_1, $[t_2,\ldots,t_k]$, π)
  - else if t_1 is a multigoal:
    - return find_multigoal_method(s, t_1, $[t_2,\ldots,t_k]$, π)
  - else error

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

$t = (name, arg_1, arg_2, \ldots, arg_j)$

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

state s, task t, $T = [t_2,\ldots, t_k]$
method m

state s; $T = [u_1, \ldots, u_j, t_2, \ldots, t_k]$

Pyhop 2 (goals)

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

state s; $T = [u_1, \ldots, u_j, t_2, \ldots, t_k]$

- find_plan(s_0, T)
  - return seek_plan(s_0, T, [ ])
- seek_plan(s, T, π)
  - if $T = [\,]$ then return π
  - 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, $[t_2,\ldots,t_k]$, π)
  - else if t_1 is a task:
    - return find_task_method(s, t_1, $[t_2,\ldots,t_k]$, π)
  - else if t_1 is a goal:
    - return find_goal_method(s, t_1, $[t_2,\ldots,t_k]$, π)
  - else if t_1 is a multigoal:
    - return find_multigoal_method(s, t_1, $[t_2,\ldots,t_k]$, π)
  - else error

multigoal: a data structure that represents a conjunction of goals

- find_goal_method(s, g, T, π)
  - if $s \vDash g$ then return π
  - 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, π)
    - if $\pi \neq$ failure then return π
  - return failure

g = (name, arg, value)

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

optional

Pyhop 2 version of the Simple Travel Problem

- Launch Python 3; load `simple_tasks1.py`

Pyhop 2 Methods

travel-by-foot(a, x, y)
    Task: travel(a,x,y)
    Pre: loc(a,x), distance(x,y) ≤ 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)]

pyhop2.declare_task_methods('travel', travel_by_foot)
```

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

```
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)]

pyhop2.declare_task_methods('travel', travel_by_taxi)
```

Pyhop 2 Actions

walk (a,x,y)

Pre: loc$(a) = x$

Eff: loc$(a) \leftarrow y$

call-taxi (a,x)

Pre: —

Eff: loc(taxi) $\leftarrow x$, loc$(a) \leftarrow$ taxi

ride-taxi (a,x,y)

Pre: loc$(a) =$ taxi, loc(taxi) $= x$

Eff: loc(taxi) $\leftarrow y$,

owe$(a) \leftarrow 1.50 + ½$ dist(x,y)

pay-driver(a,y)

Pre: owe$(a) \leq$ cash(a)

Eff: cash$(a) \leftarrow$ cash(a) – owe(a),

owe$(a) \leftarrow 0$,

loc$(a) = y$

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

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

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

def pay_driver(state,a,y):
   if state.cash[a] >= state.owe[a]:
      state.cash[a] = state.cash[a] – state.owe[a]
      state.owe[a] = 0
      state.loc[a] = y
      return state

pyhop2.declare_actions(walk,call_taxi,ride_taxi,pay_driver)
```

Travel Planning Problem

Initial state:

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

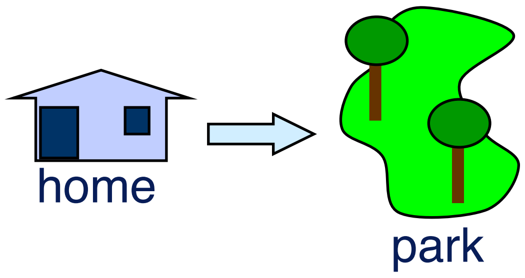
```
state1 = pyhop2.State('state1')
state1.loc = {'me':'home'}
state1.cash = {'me':20}
state1.owe = {'me':0}
state1.dist = {'home':{'park':8}, 'park':{'home':8}}
```

why not this instead?
```
state1.loc['me'] = 'home'
```

Task:

travel(me,home,park)

```
pyhop2.find_plan(state1,[('travel','me','home','park')])
```



Solution plan:

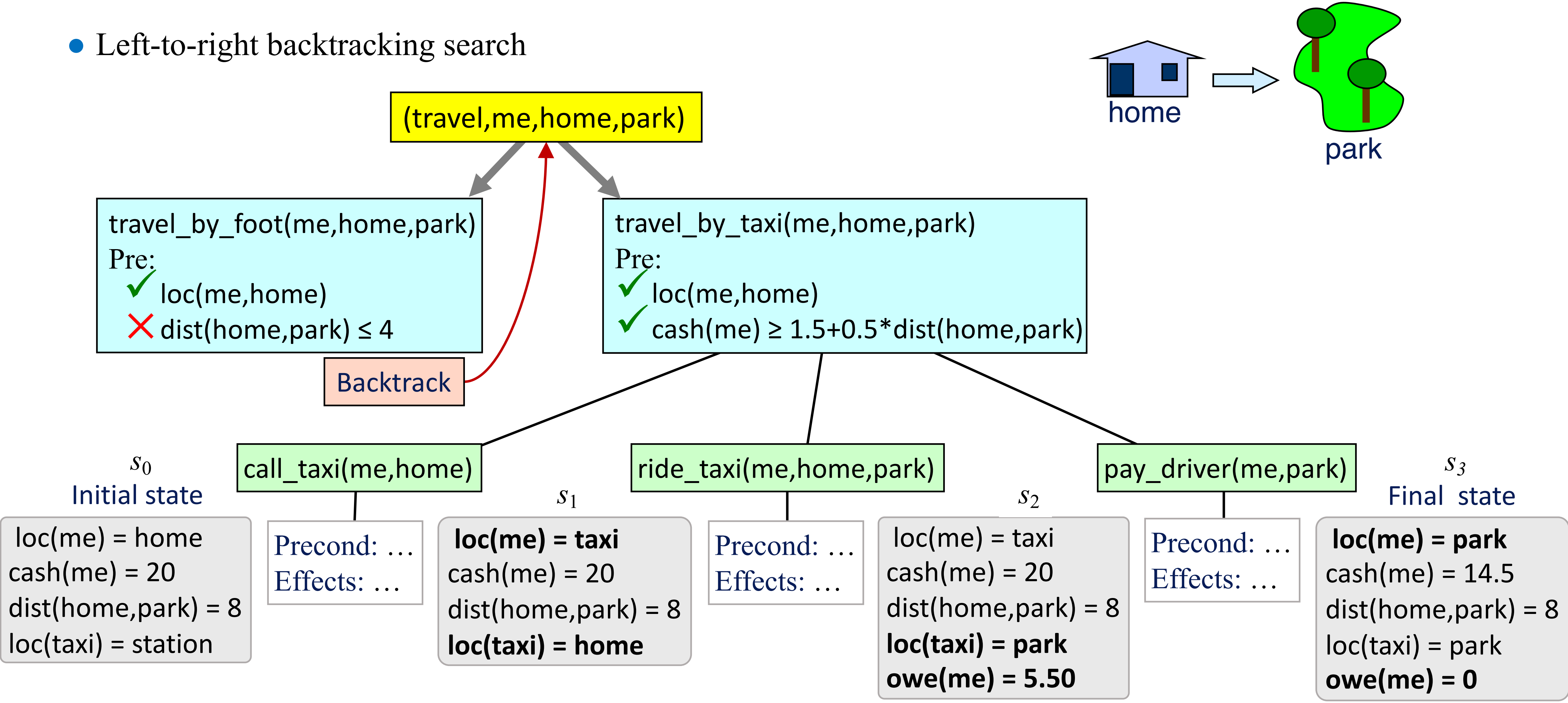
call-taxi(me,home), ride-taxi(me,park), pay-driver(me)

```
[('call_taxi', 'me', 'home’),
 ('ride_taxi', 'me', 'home', 'park’),
 ('pay_driver', 'me')]
```

To run this example in Pyhop 2:
```
import simple_tasks1.py
```

Travel-Planning Problem

- Left-to-right backtracking search



Solution plan: [(call_taxi,me,home), (ride_taxi,me,home,park), (pay_driver,me)]

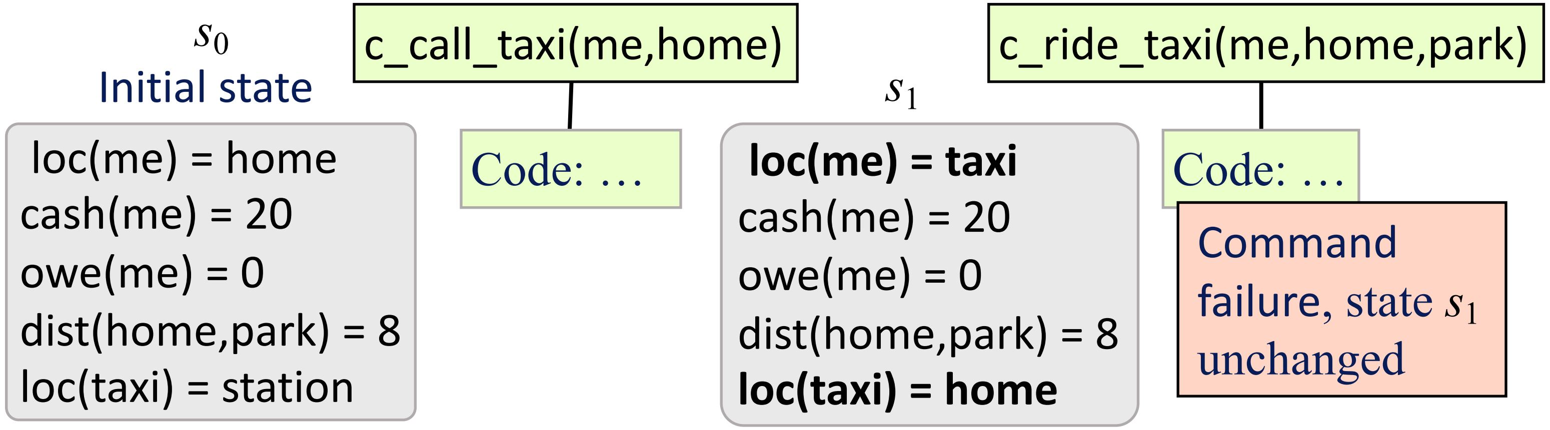
Acting and Planning

- 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

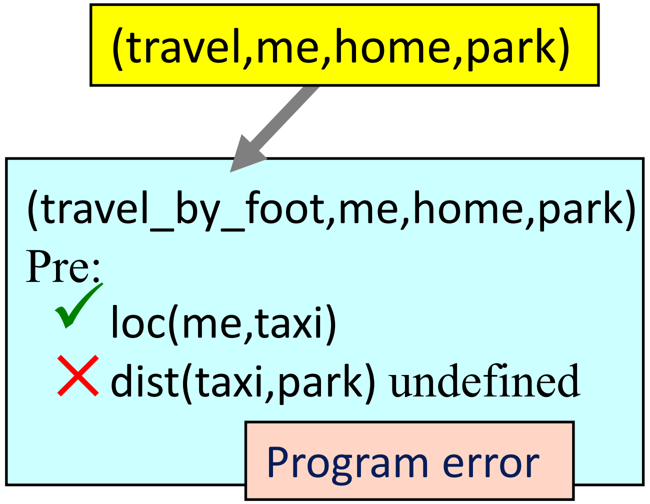
- Simple Travel Problem:
  - run_lazy_lookahead calls
    - find_plan(s_0, [(travel,me,home,park)])
  - find_plan returns
    - [(call_taxi,me,home),
      (ride_taxi,me,home,park),
      (pay_driver,me)]
  - run_lazy_lookahead executes
    - c_call_taxi(me,home)
    - c_ride_taxi(me,home,park)
    - c_pay_driver(me)
- If everything executes correctly, I get to the park
  - But suppose the taxi breaks down …

Acting and Planning

- 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)
- Suppose c_ride_taxi(me,home,park) fails:



- Next, run_lazy_lookahead calls
  - find_plan(s_1,[(travel,me,home,park)])



- To run this example in Pyhop 2:
  - `import simple_tasks2.py`
- For planning and acting, need to write HTN methods that can recover from unexpected problems

Motivation

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

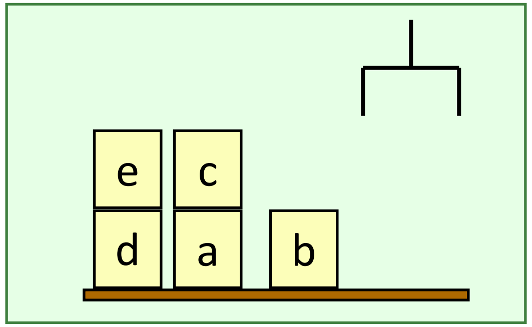
pickup(x)
pre: loc(x)=table, clear(x)=T, holding=nil
eff: loc(x)=hand, clear(x)=F, holding=x

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

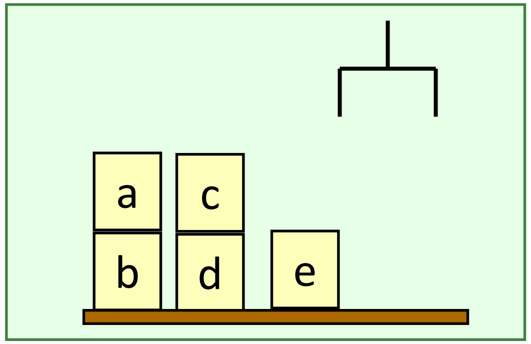
stack(x,y)
pre: holding=x, clear(y)=T
eff: holding=nil, clear(y)=F, loc(x)=y, clear(x)=T

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

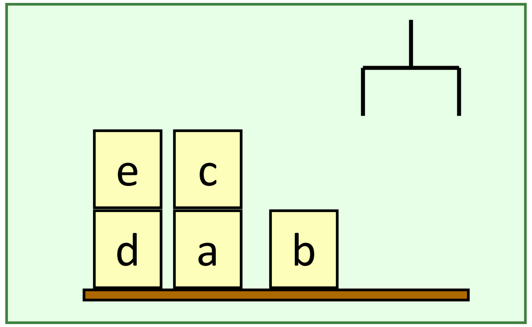


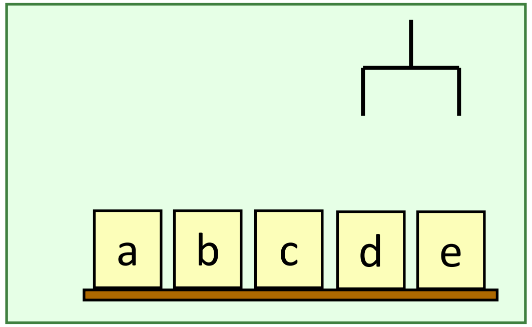
clear(a)=T, clear(b)=F, clear(c)=T, clear(d)=F, clear(e)=T, loc(a)=b, loc(b)=table, loc(c)=d, loc(d)=table, loc(e)=table, holding=nil

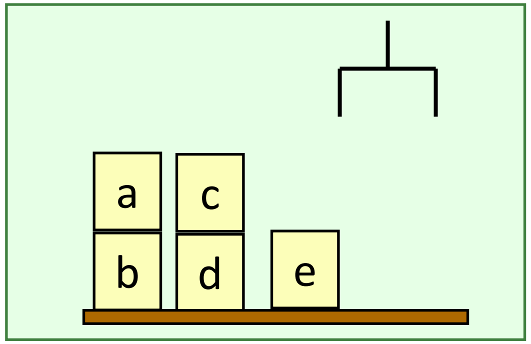


The Blocks World

- 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: loc(x)=table, clear(x)=T, holding=nil
  eff: loc(x)=hand, clear(x)=F, holding=x

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

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

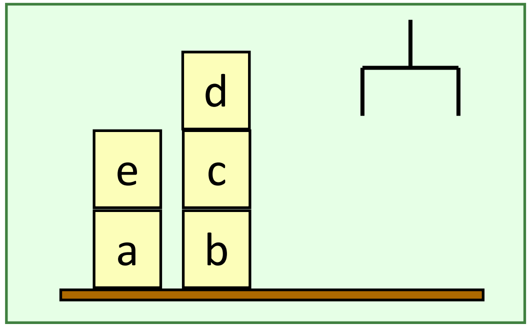
unstack(x,y)
  pre: loc(x)=y, clear(x)=T, holding=nil
  eff: loc(x)=hand, clear(x)=F, holding=x, clear(y)=T

Block-Stacking Algorithm

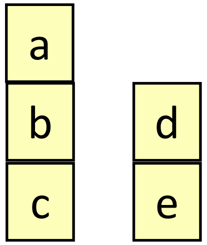
loop
    if ∃ a clear block c that needs moving
            & we can move c to a position d
                where it won't need to be moved again
        then move c to d
    else if ∃ a clear block c that needs to be moved
            then move c to the table
    else if the goal is satisfied
     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

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

s_0:



g:



⟨unstack(e,a), putdown(e), unstack(d,c), stack(d,e), unstack(c,b), putdown(c), pickup(b), stack(b,c), pickup(a), stack(a,b)⟩

Properties of the Algorithm

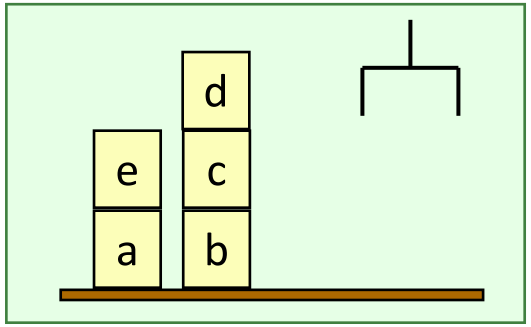
- Sound, complete, guaranteed to terminate on all block-stacking problems
- Runs in time $O(n^3)$
  - Can be modified (Slaney & Thiébaux) to run in time $O(n)$
- Often finds optimal (shortest) solutions, but sometimes only near-optimal
  - For block-stacking problems, the question "does there exist a 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 Pyhop 2:
  - `import blocks_tasks`

Pyhop 2 Implementation

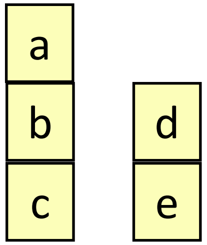
- task (move_blocks,g)
- method m_moveb(s,g)
  - if ∃ a clear block c that needs moving, and we can move c to a location d where it won't need to be moved again
  - then return [(move_one,c,d), (move_blocks,g)]
  - else if ∃ a clear block c that needs to be moved
  - then return [(move_one,c,table), (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 $d \neq e$
  - s contains loc(c)=d and g contains loc(b)=d, where $b \neq c$ and $d \neq$ table
  - s contains loc(c)=d and d needs to be moved

s_0:



g:



[(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)]

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 Pyhop 2 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`
    - Pyhop 2 at `https://github.com/patras91/pyhop2`
- Examples: simple travel, blocks world
- To integrate planning and acting, need to make sure the HTN methods can handle unexpected events