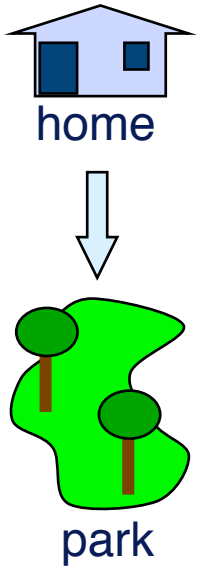
HTN Planning

- 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 ways to combine 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
  - HTN planners use such recipes to generate the search space
- Ingredients
  - states, tasks, operators, methods, planning algorithm

States and Tasks



- **State**: description of the current situation
  - » I'm at home, I have €20, there's a park 8 km away
- **Task**: description of an activity to perform
  - » Travel to the park
- Two kinds of tasks
  - » ***Primitive*** task: a task that corresponds to a basic action
  - » ***Compound*** task: a task that is composed of other simpler tasks

Operators

- **Operators**: parameterized descriptions of what the basic actions do
  - ***walk*** from location x to location y
    - Precond: agent is at x
    - Effects: agent is at y
  - ***call taxi*** to location x
    - Precond: (none)
    - Effects: taxi is at x
  - ***ride taxi*** from location x to location y
    - Precond: agent and taxi are at x
    - Effects: agent and taxi at y, agent owes $1.50 + \frac{1}{2}$ distance(x,y)
  - ***pay driver***
    - Precond: agent owes amount of money r, agent has money $m \geq r$
    - Effects: agent owes nothing, agent has money $m - r$
- **Actions**: operators with arguments

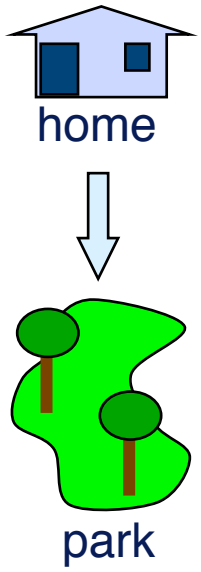
Methods

- Method: parameterized description of a possible way to perform a compound task by performing a collection of subtasks
- There may be more than one method for the same task

  - ***travel by foot*** from *x* to *y*
    - Task: travel from *x* to *y*
    - Precond: agent is at *x*, distance to *y* is ≤ 4 km
    - Subtasks: walk from *x* to *y*
  - ***travel by taxi*** from *x* to *y*
    - Task: travel from *x* to *y*
    - Precond: agent is at *x*, agent has money ≥ 1.5 + ½ distance(*x*,*y*)
    - Subtasks: call taxi to *x*,
      ride taxi from *x* to *y*,
      pay driver

Simple Travel-Planning Problem

- Left-to-right backtracking search (SHOP)



travel from home to park

Travel by foot

Precond:
✓ I'm at home
✗ home to park ≤ 4 km

Travel by taxi

Precond:
✓ I'm at home
✓ I have ≥ €5.50

Backtrack

s_0

call taxi to home

s_1

ride taxi to park

s_2

pay driver

s_3

Precond: …
Effects: …

Precond: …
Effects: …

Precond: …
Effects: …

Initial state

I'm at home
I have €20
home to park is 8 km

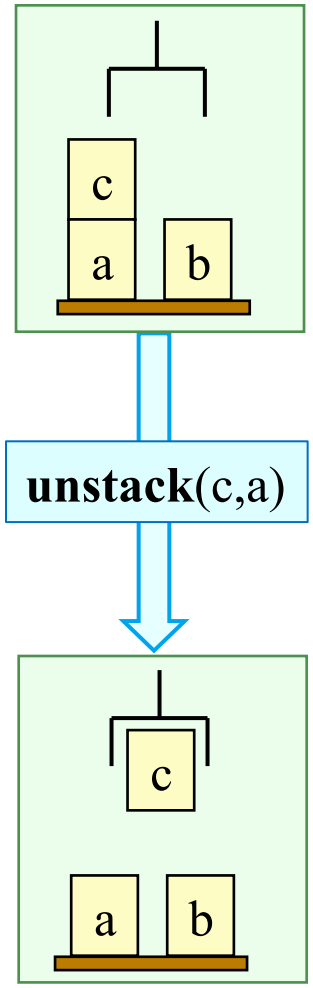
I'm at home
I have €20
home to park is 8 km
Taxi is at home

I'm at the park
I have €20
home to park is 8 km
I owe €5.50
Taxi is at the park

Final state

I'm at the park
I have €14.50
home to park is 8 km
I owe nothing
Taxi is at the park

Propositions Versus State Variables



{ontable(a), on(c,a),
clear(c), ontable(b),
clear(b), handempty}

{loc(a)=table, clear(a)=0, loc(c)=a,
clear(c)=1, loc(b)=table,
clear(b)=1, holding=nothing}

unstack(*x*,*y*)
Precond: on(*x*,*y*), clear(*x*), handempty
Effects: ¬on(*x*,*y*), ¬clear(*x*), clear(*y*), holding(*x*), ¬handempty

unstack(*x*,*y*)
Precond: loc(*x*) = *y*, *y* ≠ table, clear(*x*) = 1, holding = nothing
Effects: loc(*x*) = hand, clear(*x*) = 0, clear(*y*) = 1, holding = *x*

- Classical representation:
  - » State: set of propositions
  - » Actions add/delete them
- PDDL is based on this
- Reason is largely historical
  - » AI planning evolved out of AI theorem proving

- State-variable representation:
  - » State: variable bindings
  - » Actions change the values
- Same expressive power
- More compatible with conventional computer programming

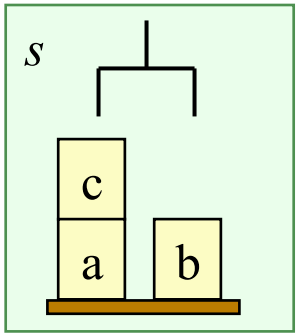
Pyhop

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

- Planning algorithm is like the one in SHOP
- Main differences:
  - » 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



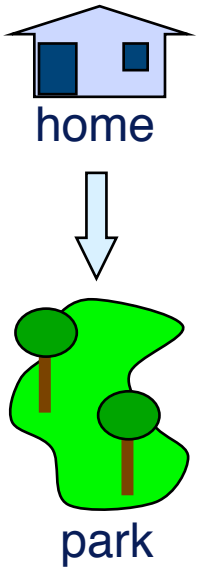
Travel-Planning Methods

travel by foot from x to y

Task: travel from x to y

Precond: agent is at x, distance to y is ≤ 4 km

Subtasks: walk from x to y




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

travel by taxi from x to y

Task: travel from x to y

Precond: agent is at x, agent has money $\geq 1.5 + ½$ distance(x,y)

Subtasks: call taxi to x, ride taxi from x to y, pay driver

```
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 Operators (1)

walk from x to y

Precond: agent is at location x

Effects: agent is at location y

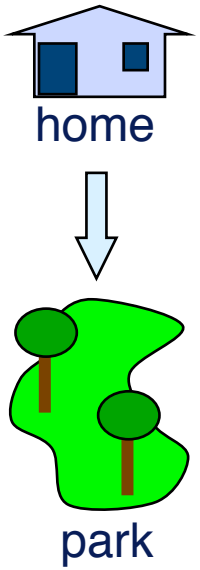
```
def walk(state,a,x,y):
    if state.loc[a] == x:
        state.loc[a] = y
        return state
    else: return False
```

call taxi to location x

Precond: (none)

Effects: taxi is at location x

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

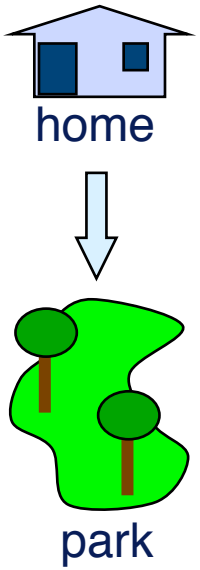


Travel-Planning Operators (2)

ride taxi from *x* to *y*

Precond: agent and taxi are at *x*

Effects: agent and taxi are at *y*, agent owes 1.5 + ½ distance(*x*,*y*)




```
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
```

pay driver

Precond: agent owes money, and has at least as much as what's owed

Effects: agent owes nothing, agent's money reduced by what was owed

```
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)
```