Section 11.2: Hierarchical Planning

Dana S. Nau

CMSC 421, Fall 2012

Motivation

- For some planning problems, we may already have ideas about good ways to solve them
- Example: travel to a destination that's far away:
  - Domain-independent planner:
    - » many combinations 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 to get planning systems to use such recipes?
  - General approach: Hierarchical Task Network (HTN) planning
  - We'll look at a simpler special case: *Task-List Planning*

Task-List Planning

- States and operators: same as in classical planning
- Instead of achieving a *goal*, we will want to accomplish a list of *tasks*
  - Recursively decompose tasks into smaller and smaller subtasks
  - At the bottom, actions that we know how to accomplish directly

- *Task*: an expression of the form $t(u_1,\ldots,u_n)$
  - t is a *task symbol*, and each u_i is a term

- Two kinds of task symbols (and tasks):
  - *primitive*: tasks that we know how to execute directly
    - » task symbol is the head of an operator
  - *nonprimitive*: tasks that must be decomposed into subtasks
    - » use *methods* (next slide)

Methods

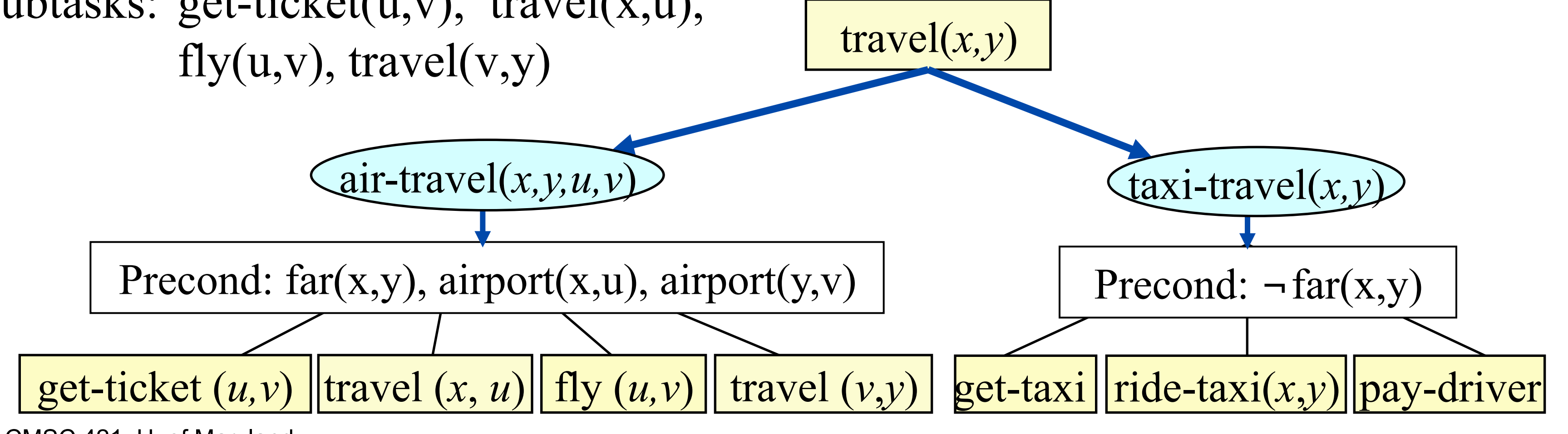
- Method: a 4-tuple $m = (head, task, precond, subtasks)$
  - *head*: the method's *name*, followed by list of variable symbols $(x_1,\ldots,x_n)$
  - *task*: a nonprimitive task
  - *precond*: preconditions (literals)
  - *subtasks*: a sequence of tasks $\langle t_1, \ldots, t_k \rangle$

air-travel(x,y,u,v)

task: travel(x,y)

precond: far(x,y), airport(x,u), airport(y,v)

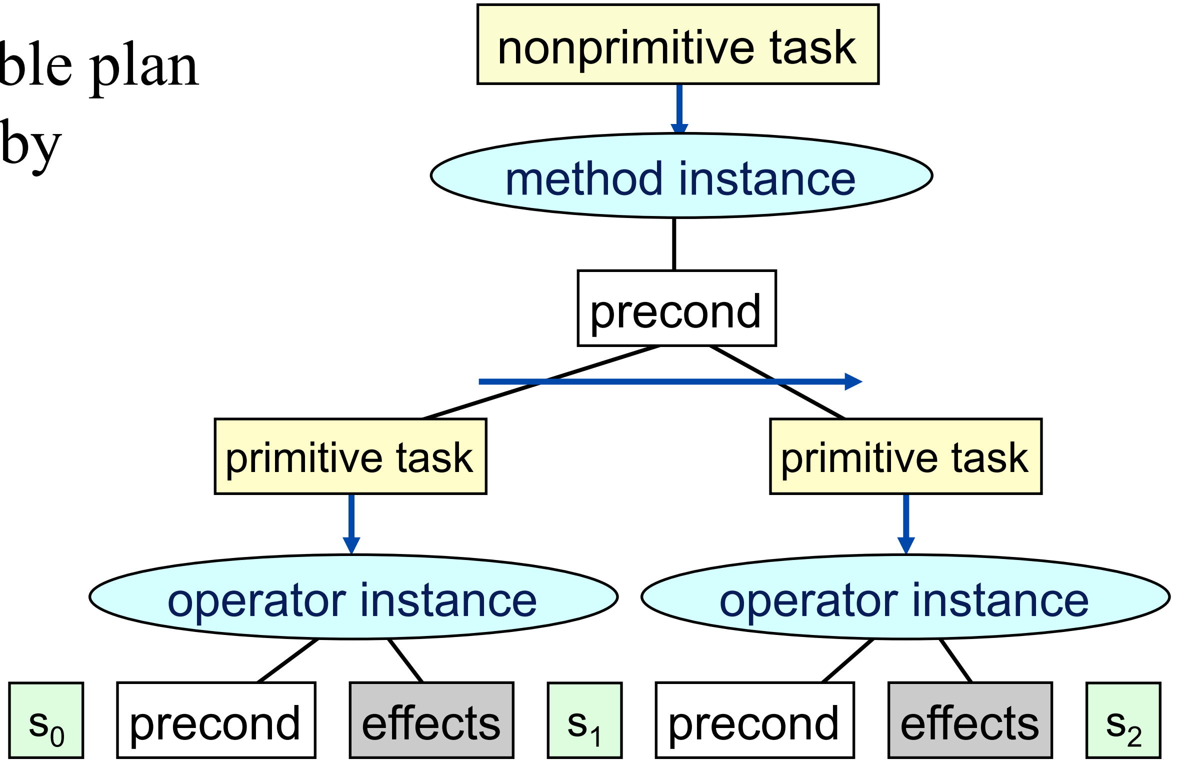
subtasks: get-ticket(u,v), travel(x,u),
fly(u,v), travel(v,y)



Domains, Problems, Solutions

- Task-list planning domain: methods, operators
- Task-list planning problem: methods, operators, initial state, initial task list

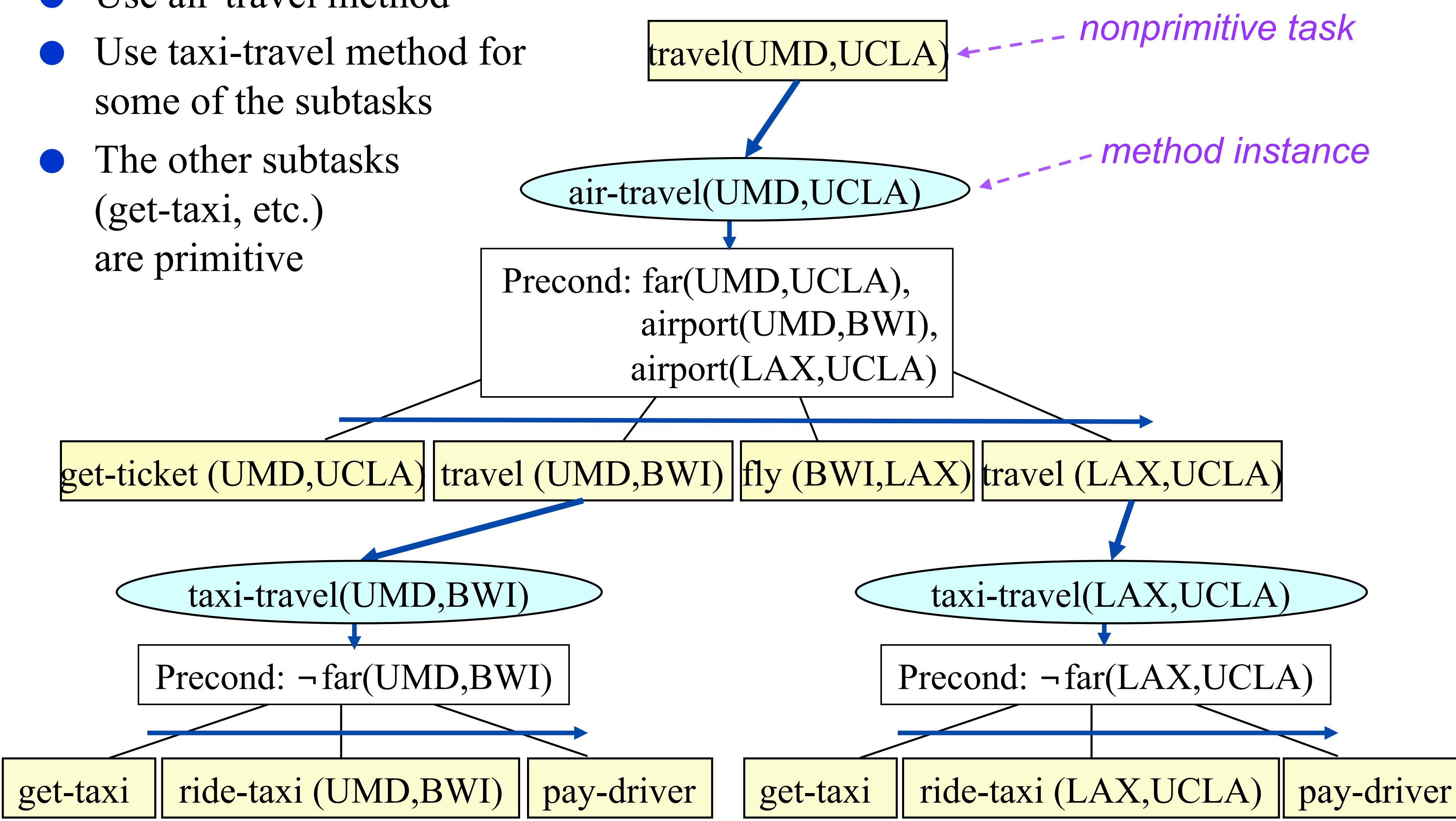
- Solution: any executable plan that can be generated by recursively applying
  - methods to nonprimitive tasks
  - operators to primitive tasks



Example

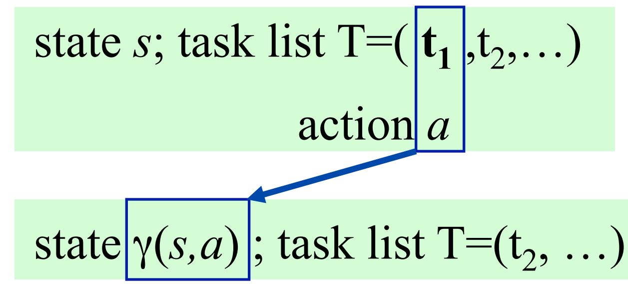
Task: travel from UMD to UCLA

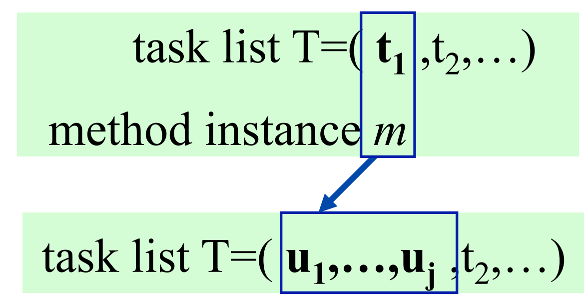
- Use air-travel method
- Use taxi-travel method for some of the subtasks
- The other subtasks (get-taxi, etc.) are primitive



Solving Task-List Planning Problems

- TFD(s,(t_1,…,t_k))
  - if k=0 (i.e., no tasks) then return the empty plan
  - else if there is an action a such that head(a) = t_1 then
    - » if s satisfies precond(a) then
      - • return TFD(γ(s,t_1),(t_2,…,t_k))
    - » else return failure
  - else
    - » A = {m : m is a method instance such that task(m)=t_1, and s satisfies precond(m)}
    - » if *active* is empty then return failure
    - » nondeterministically choose m in A
    - » let u_1…, u_j be m's subtasks
    - » return TFD(s, (u_1…, u_j, t_2, …, t_k))





Example

- TFD(s,(t_1,…,t_k))
  - if k=0 (i.e., no tasks) then return the empty plan
  - else if there is an action a such that head(a) = t_1 then
    - » if s satisfies precond(a) then
      - return TFD($\gamma(s,t_1)$,(t_2,…,t_k))
    - » else return failure
  - else
    - » A = {m : m is a method instance such that task(m)=t_1, and s satisfies precond(m)}
    - » if *active* is empty then return failure
    - » nondeterministically choose m in A
    - » let u_1…, u_j be m's subtasks
    - » return TFD(s, (u_1…, u_j, t_2, …, t_k))

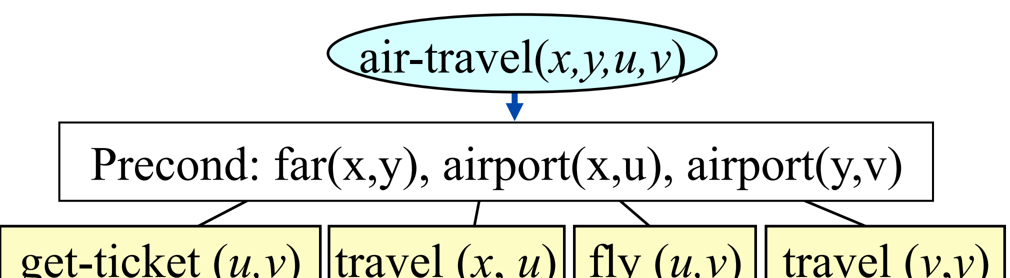
s_0: far(UMD,UCLA), airport(UMD,BWI), airport(UCLA,LAX)

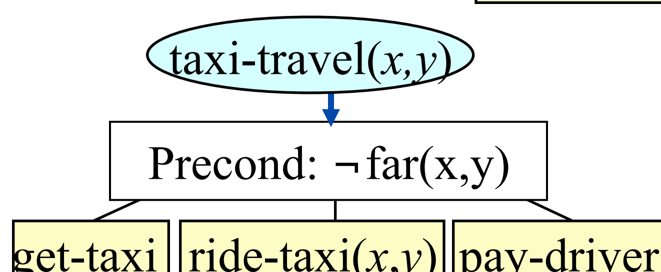
task list: ⟨travel(UMD,UCLA)⟩

apply air-travel method: ⟨get-ticket (UMD,UCLA) travel (UMD,BWI) fly (BWI,LAX) travel (LAX,UCLA)⟩

apply get-ticket action: far(UMD,UCLA), airport(UMD,BWI), airport(UCLA,LAX) ticket(UCLA,LAX)

apply taxi-travel method: ⟨get-taxi ride-taxi(UMD,BWI) pay-driver fly (BWI,LAX) travel (LAX,UCLA)⟩





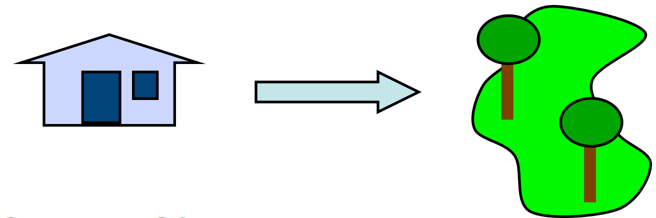
Comparison to Classical Planners

- Advantages:
  - Can encode "recipes" (standard ways do planning in a given domain) as collections of methods and operators
    - Helps the planning system do more-intelligent search - can speed up planning by many orders of magnitude (e.g., polynomial time versus exponential time)
    - Produces plans that correspond to how a human might solve the problem
  - Greater expressive power
    - Preconditions and effects aren't limited to just sets of literals
- Disadvantages:
  - More complicated than just writing classical operators
  - The author needs knowledge about planning in the given domain

SHOP and SHOP2

- SHOP and SHOP2:
  - `http://www.cs.umd.edu/projects/shop`
  - Generalized versions of TFD
  - SHOP2 an award in the AIPS-2002 Planning Competition
- Freeware, open source
  - Downloaded more than 13,000 times – I stopped keeping track
  - Used in hundreds (thousands?) of projects worldwide

HTN planning with state variables



- Simple travel-planning domain
  - Go from one location to another
- Represent states as collections of variables
  - Equivalent expressive power, but easier to understand

method travel-by-foot (a,x,y)
precond: $distance(x,y) \leq 2$
task: travel(a,x,y)
subtasks: walk(a,x,y)

method travel-by-taxi (a,x,y)
task: travel(a,x,y)
precond: $cash(a) \geq 1.5 + 0.5 \times distance(x,y)$
subtasks: $\langle$call-taxi(a,x), ride(a,x,y), pay-driver$(a,x,y)\rangle$

operator walk (a,x,y)
precond: $location(a) = x$
effects: $location(a) \leftarrow y$

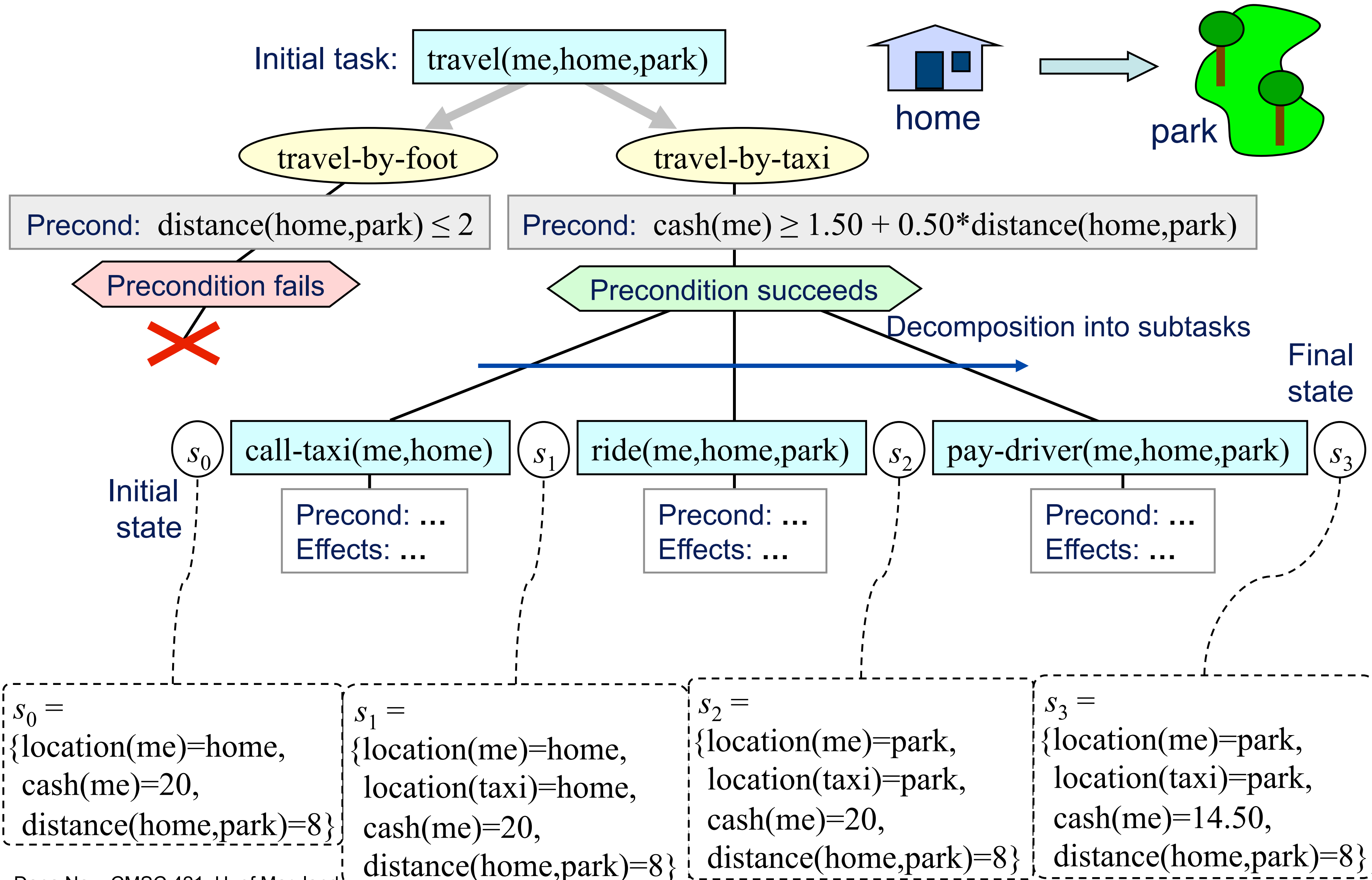
operator call-taxi(a,x)
effects: $location(taxi) \leftarrow x$

operator ride-taxi(a,x,y)
precond: $location(taxi) = x$, $location(a) = x$
effects: $location(taxi) \leftarrow y$, $location(a) \leftarrow y$

operator pay-driver(a,x,y)
precond: $cash(a) \geq 1.5 + 0.5 \times distance(x,y)$
effects: $cash(a) \leftarrow cash(a) - 1.5 + 0.5 \times distance(x,y)$

Planning Problem:

I am at home, I have $20,
I want to go to a park 8 miles away



Pyhop

- A simple HTN planner written in Python; works in Python 2.7 and 3.2
- Somewhat similar to SHOP
- The main differences:
  - HTN operators and methods are Python functions
  - States are collections of **variables**, not logical atoms.
    - Instead of writing on(a,b), you might write something like loc[a] = b
  - The current state is a python object; must refer to it explicitly in the operator and method definitions
    - In the above example, what you would **really** write is state.loc[a] = b
  - You can define a goal as a python object
    - You might write goal.loc[a] = b to specify that your goal of having block a on block b
    - Pyhop doesn't explicitly check to see if the goal is achieved, but you can use it to hold information that you might want to use in your operators and methods