Lecture slides for

Automated Planning: Theory and Practice

Chapter 11
Hierarchical Task Network Planning

Dana S. Nau

University of Maryland

12:50 PM September 27, 2013

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:

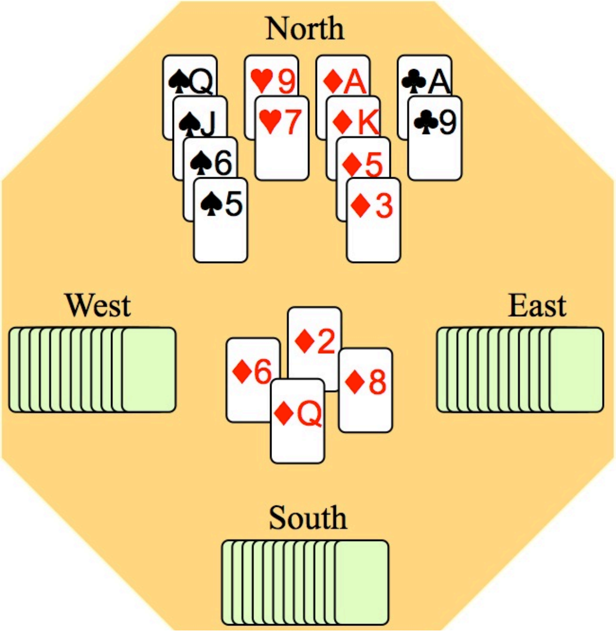
    1. buy ticket from local airport to remote airport
    2. travel to local airport
    3. fly to remote airport
    4. travel to final destination
- How to provide such information to a planning system?

Two Approaches

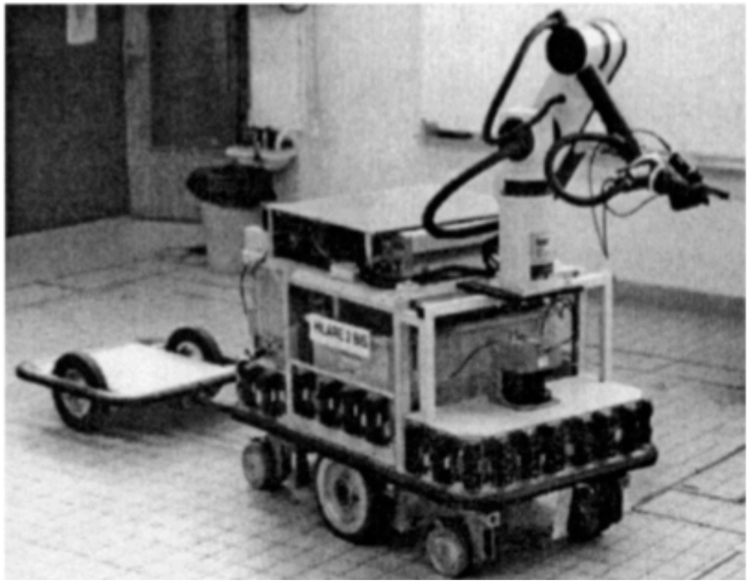
- Control rules (chapter 10):
  - Write rules to prune actions that *don't* fit the recipe
- Hierarchical Task Network (HTN) planning:
  - Describe how to consider only the actions that *do* fit the recipe

```
depth-first-search(D, s0, g)
  π ← ⟨⟩; s ← s0
  loop
    if s satisfies g then return π
    A′ ← {a | s satisfies Pre(a)}
    let Act ⊆ A′
    while Act ≠ ∅ do
      select a ∈ Act
      remove a from Act
      s ← γ(s, a)
      π ← π · a
    return failure
```

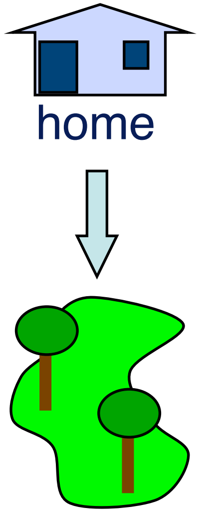
HTN Planning



- Ingredients
  - states and actions
  - tasks: activities to perform
  - methods: ways to perform the activities
  - planning algorithm
- HTN planners may be domain-specific
  - Chapter 20 (robotics)
  - Chapter 23 (bridge)
- Or domain-configurable
  - Domain-independent planning algorithm
  - Domain model includes definitions of tasks and methods
  - Planner needs to be able to read and understand them



States and Tasks



- **State**: description of the current situation
  - I'm at home, I have $20, there's a park 8 miles 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
- This time I won't require everything to be function-free
  - That was needed in Chapters 4 and 5, but not here
- Formulas can include functions and state variables
- Not every variable needs to be an argument

Parameterized actions

- walk (a: *Agents*, x: *Locations*, y: *Locations*)
  - Pre: loc(a) = x
  - Eff: loc(a) $\leftarrow y$

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

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

- pay-driver(a: *Agents*)
  - Pre: owe(a) = r, cash(a) $\geq r$
  - Pre: owe(a) $\leftarrow 0$, cash(a) $\leftarrow$ cash(a) $- r$

Methods

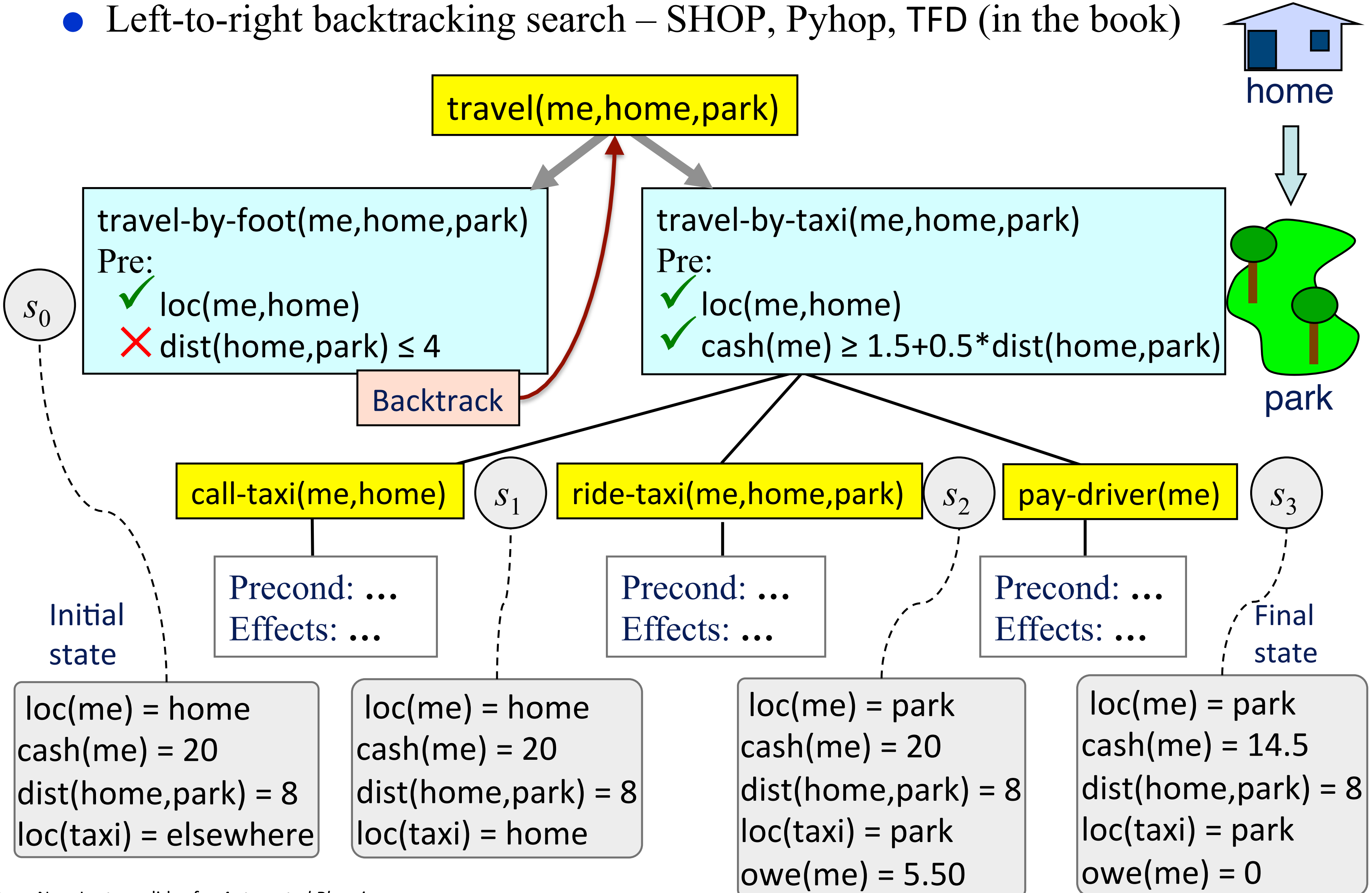
- Method: parameterized description of a possible way to perform a compound task by performing a collection of subtasks
- There may be more than one method for the same task

  - travel-by-foot(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) $\geq 1.50 + \frac{1}{2}$ dist(x,y)
    - Sub: call-taxi (a,x), ride-taxi (a,x,y), pay-driver(a)

Simple Travel-Planning Problem

- Left-to-right backtracking search – SHOP, Pyhop, TFD (in the book)



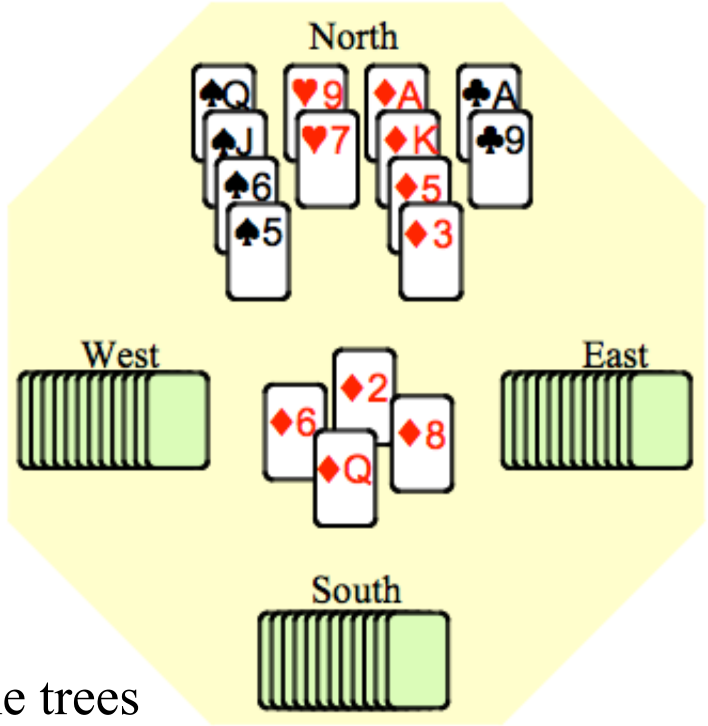
SHOP and SHOP2

- SHOP and SHOP2:
  - http://www.cs.umd.edu/projects/shop
  - HTN planning systems
  - SHOP2 an award in the AIPS-2002 Planning Competition
- Instead of state variables, used "classical plus functions"
- Freeware, open source
  - Downloaded more than 20,000 times
  - Used in many hundreds of projects worldwide
    - Government labs, industry, academia

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 it and 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

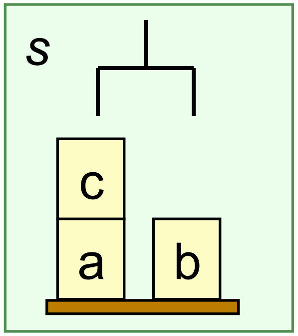
KILLZONE 2

- 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

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

Actions

walk(a: *Agents*, x: *Locations*, y: *Locations*)
  Pre: loc(a) = x
  Eff: loc(a) = y

call-taxi(a: *Agents*, x: *Locations*)
  Pre: —
  Eff: loc(taxi) = x

ride-taxi(a: *Agents*, x: *Locations*,
      y: *Locations*)
  Pre: loc(a) = x, loc(taxi) = x
  Eff: loc(a) = y, loc(taxi) = y,
    owe(a) = 1.50 + ½ distance(x,y)

pay-driver(a: *Agents*)
  Pre: owe(a) = r, cash(a) ≥ r
  Pre: owe(a) = r,
    cash(a) = cash(a) – r

```
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: 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: 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_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
```

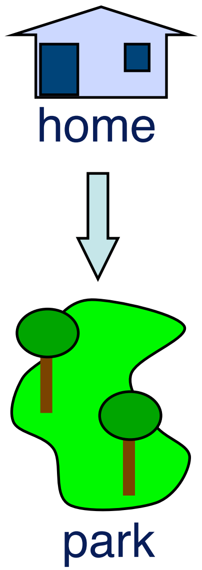
```
declare_methods('travel', travel_by_foot, travel_by_taxi)
```

Travel Planning Problem

Initial state:

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

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



Task:

travel(me,home,park)

```
# Invoke the planner
pyhop(state1,[('travel','me','home','park')])
```

Solution plan:

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

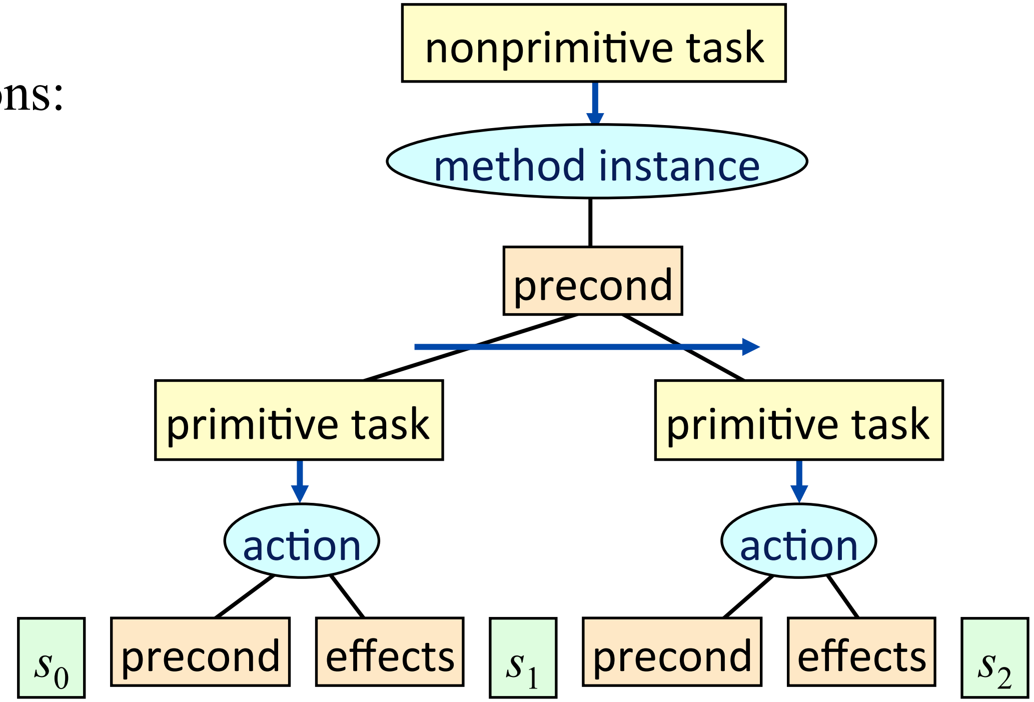
```
[('call_taxi', 'me', 'home'), ('ride_taxi', 'me', 'home', 'park'), ('pay_driver', 'me')]
```

Total-Order HTN Planning

- State-variable version of what the book calls STN planning
- Planning domain: a pair (Σ,*M*)
  - Σ: state-transition system
    - parameterized PE-specification
  - *M*: set of methods
    - Parameterized specifications:

      method-name(*args*)
      Task: *task-name*(*args*)
      Pre: *preconditions*
      Sub: *list of subtasks*
- Planning problem: (Σ, M, s_0, T)
  - $T = \langle t_1, t_2, \ldots, t_k \rangle$
- Task specification:
  - *task-name*(*args*)

- Solution: any executable plan that can be generated by applying
  - methods to monprimitive tasks
  - actions to primitive tasks



Planning Algorithm

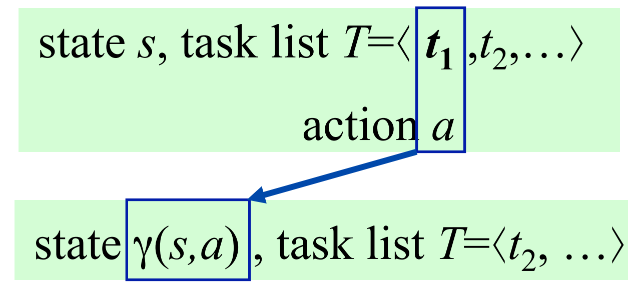
- TFD(Σ, M, s, T)

  state-variable version of the algorithm in the book

  - if $T = \langle\ \rangle$ then return $\langle\ \rangle$
  - let the tasks in T be $t_1, t_2, \ldots, t_k$

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

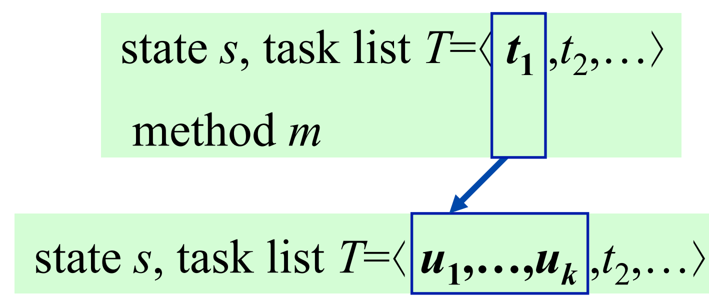
  - if t_1 is primitive then
    - $Act = \{a \mid \text{head}(a) \text{ matches } t_1 \text{ and } a \text{ is applicable in } s\}$
    - if $Act = \varnothing$ then return failure
    - nondeterministically choose any $a \in Act$
    - π = TFD(Σ, $\gamma(s,a)$, $\langle t_2,\ldots,t_k\rangle$)
    - if π = failure then return failure
    - else return $a \bullet \pi$



  - else

    t_1 is nonprimitive

    - $Act = \{m \in M \mid \text{task}(m) \text{ matches } t_1 \text{ and } m \text{ is applicable in } s\}$
    - if $Act = \varnothing$ then return failure
    - nondeterministically choose any $a \in Act$
    - return TFD(Σ, M, s, sub(m) $\bullet$ $\langle t_2,\ldots,t_k\rangle$)

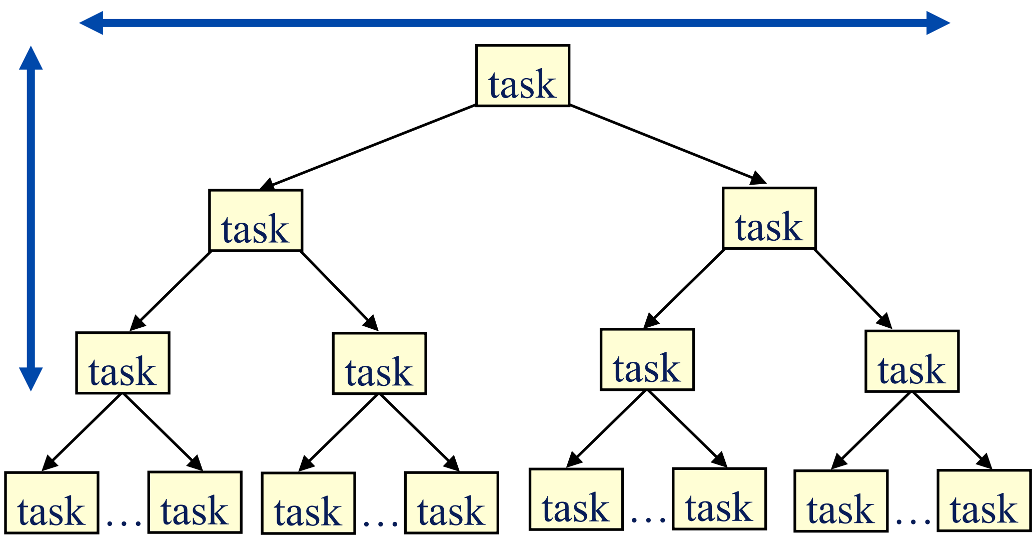


HTN Planning in General

- SHOP uses the book version of TFD
  - Pyhop uses the state-variable version
- Other formalisms and algorithms
  - Some of them use partially ordered tasks
    - Total-order forward search – PFD in the book, SHOP2
    - Plan-space planning – SIPE, O-Plan, UMCP
      - These allow more constraints than just preconditions
        - postconditions, "during" conditions, etc.
  - Some of them use goals and subgoals instead of tasks and subtasks
    - Angelic Hierarchical A*
    - GDP, GoDeL

Comparison to Forward and Backward Search

- In HTN planning, more possibilities than just forward or backward
  - A little like the choices to make in parsing algorithms
- SHOP, Pyhop, GDP, GoDeL:
  - down, then forward
  - backtracking
- SIPE, O-Plan, UMCP
  - plan-space
    (down and backward)
- Angelic Hierarchical A*
  - use abstract actions to produce abstract states
  - forward A*, at the top level
  - forward A*, one level down
  - …



HTN Planning vs. Domain-Independent Planning

- Advantage: HTN planners can encode "recipes" as collections of methods and operators
  - Express things that can't be expressed in classical planning
  - Specify standard ways of solving problems
    - Otherwise, the planning system would have to derive these again and again from "first principles," every time it solves a problem
    - Can speed up planning by many orders of magnitude (e.g., polynomial time versus exponential time)
- Disadvantage: writing and debugging an HTN domain model can be much more work than just writing actions

- In problems that a classical planner can solve, why go to the trouble?
  - If it's important to achieve high performance
  - If you need more expressive power than classical planners can provide
- Otherwise it might not be worth the effort

Example

- All of the competitions included domain-independent planners
- AIPS 2000 and *IPC* 2002 also included configurable planners
- The configurable planners
  - Solved the most problems
  - Solved them the fastest
  - Usually found better solutions
  - Worked in non-classical planning domains that were beyond the scope of the domain-independent planners
- Subsequent competitions didn't include configurable planners
- Hard to enter them in the competition
  - Must write all the domain knowledge yourself
  - Too much trouble except to make a point
  - The authors of TLPlan, TALplanner, and SHOP2 felt they had already made their point

AIPS 1998 Planning Competition

AIPS 2000 Planning Competition

IPC 2002

IPC 2004

IPC 2006

⋮