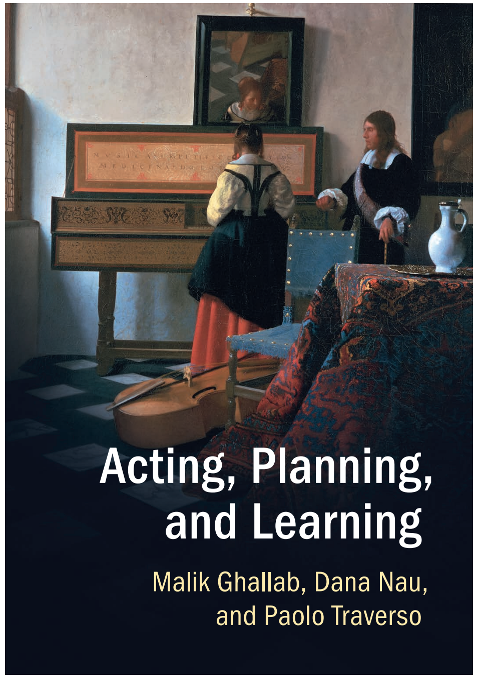
Last update: 11:09 PM, March 7, 2025



Chapter 5
HTN Representation and Planning

Dana S. Nau
University of Maryland

with contributions from
Mark “mak” Roberts

Hierarchical Task Network (HTN) Planning

- 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

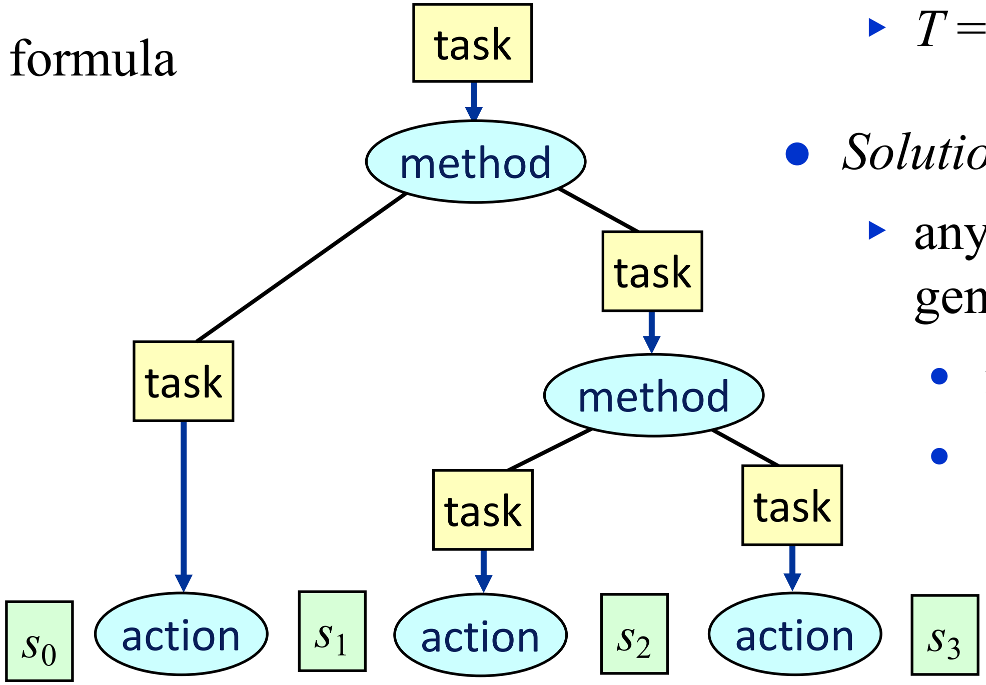
- Two ways to put such information into a planner
  - Domain-specific algorithm
  - Domain-independent planning engine + domain-specific planning information
    - HTN planning (this Part)

- Ingredients:
  - state-variable planning domain (Part I)
  - *tasks*: activities to perform
  - *HTN methods*: ways to perform tasks

Total-Order HTN Planning

- Three kinds of tasks
  - *Primitive* task: head of an action
  - *Compound* task: *name*(*args*)
    - *name* is a *compound-task name*
  - *Goal task:* goal(g)
    - g is any classical goal formula
- Method: a tuple (*head*, *nonprimitive task*, *preconditions*, *subtasks*)
- Write it as pseudocode:

  method-name(*args*)
  Task: *nonprimitive task*
  Pre: *preconditions*
  Sub: *list of subtasks*



- TOHTN planning domain: a pair $(\Sigma, \mathcal{M})$
  - Σ: state-variable planning domain
  - $\mathcal{M}$: set of methods
- TOHTN planning problem $P = (\Sigma, \mathcal{M}, s_0, T)$
  - $T = \langle t_1, t_2, \ldots, t_k \rangle$
- *Solution* for *P:*
  - any executable plan that can be generated for T by applying
    - methods to nonprimitive tasks
    - actions to primitive tasks

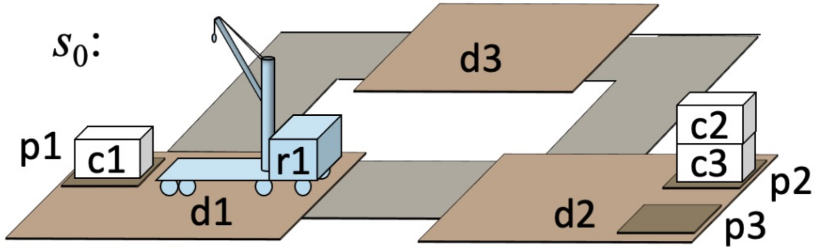
The DWR Domain from Chapter 2

- The slides for Chapter 2 used a simpler domain than the one in the book
  - Too simple to illustrate what HTNs can do
- Here's the DWR domain from the book

- Objects:
  - robots r1, r2
  - loading docks d1, d2, d3
  - containers c1, c2, c3
  - piles p1, p2, p3
- Rigid relations:
  - adjacent = {(d1,d2), (d2,d1), (d2,d3), (d3,d2), (d3,d1), (d1,d3)};
  - at = {(p1, d1), (p2, d2), (p3, d2)}.

- State variables:
  - cargo(r) ∈ *Containers* ∪ {nil}
  - loc(r) ∈ *Docks*
  - occupied(d) ∈ {T, F}
  - pile(c) ∈ *Piles* ∪ {nil}
  - pos(c) ∈ *Robots* ∪ *Containers* ∪ {nil}
  - top(p) ∈ *Containers* ∪ {nil}
  - where r ∈ *Robots*, c ∈ *Containers*, p ∈ Piles

- s_0 = {cargo(r1) = nil, cargo(r2) = nil,
  loc(r1) = d1, loc(r2) = d2,
  occupied(d1)=T, occupied(d2)=F, occupied(d3)=F,
  pile(c1) = p1, pile(c2) = p2, pile(c3) = p2,
  pos(c1) = nil, pos(c2) = c3, pos(c3) = nil,
  top(p1)=c1, top(p2) = c2, top(p3) = nil}



The DWR Domain from Chapter 2

Action schemas:

- take(r, c, c', p, d)
  - pre: at(p,d), cargo(r) = nil, loc(r) = d, pos(c)= c', top(p) = c
  - eff: cargo(r) $\leftarrow c$, pile(c) $\leftarrow$ nil, pos(c) $\leftarrow r$, top(p) $\leftarrow c'$
- put(r, c, c', p, d)
  - pre: at(p,d), pos(c) = r, loc(r) = d, top(p) = c'
  - eff: cargo(r) $\leftarrow$ nil, pile(c) $\leftarrow p$, pos(c) $\leftarrow c'$, top(p) $\leftarrow c$
- move(r, d, d')
  - pre: adjacent(d,d'), loc(r) = d, occupied(d') = F
  - eff: loc(r) $\leftarrow d'$, occupied(d) $\leftarrow$ F, occupied(d') $\leftarrow$ T

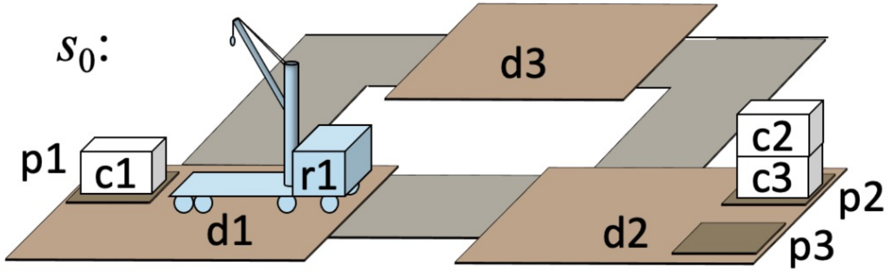
where

- $c \in$ *Containers*; $c' \in$ *Containers* ∪ *Robots* ∪ {nil};
- $d, d' \in$ *Docks*; $p \in$ *Piles*; $r \in$ *Robots*.

- State variables:
  - cargo(r) ∈ *Containers* ∪ {nil}
  - loc(r) ∈ *Docks*
  - occupied(d) ∈ {T, F}
  - pile(c) ∈ *Piles* ∪ {nil}
  - pos(c) ∈ *Robots* ∪ *Containers* ∪ {nil}
  - top(p) ∈ *Containers* ∪ {nil}
  - where $r \in$ *Robots*, $c \in$ *Containers*, $p \in$ Piles

- s_0 = {cargo(r1) = nil, cargo(r2) = nil,
  loc(r1) = d1, loc(r2) = d2,
  occupied(d1)=T, occupied(d2)=F, occupied(d3)=F,
  pile(c1) = p1, pile(c2) = p2, pile(c3) = p2,
  pos(c1) = nil, pos(c2) = c3, pos(c3) = nil,
  top(p1)=c1, top(p2) = c2, top(p3) = nil}

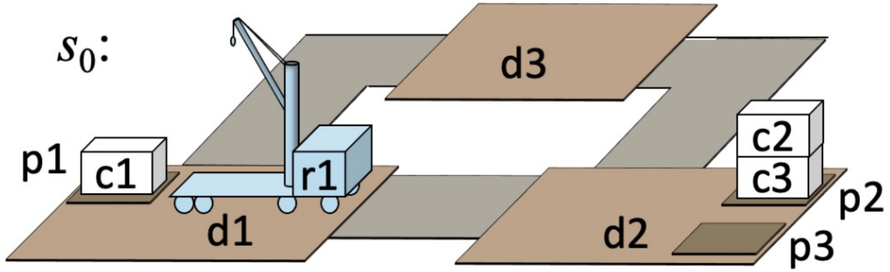
Poll: Notice that cargo(r) = c iff pos(c) = r. Can we rewrite the domain to eliminate cargo(r)?
A. yes B. no C. don't know



TOHTN Planning Domain

If I say "HTN" assume TOHTN unless stated otherwise

- TOHTN planning domain $\Sigma = (\Sigma_c, \mathcal{M})$
  - Σ_c = DWR domain on the previous pages
  - $\mathcal{M}$ = a set of eight methods:



- Compound task put-in-pile(r,c,p,d): put container c into pile p if it isn't there already

m1-put-in-pile(r, c, p, d)
  task: {pile(c) = p}
  pre: at(p, d), pile(c) $\neq p$, cargo(r) = nil
  sub: get-container(r, c), navigate(r, d), put(r, c, top(p), p, d)

  - Preconditions:
    - 1st one ensures d has the correct value
    - Others check for applicability
  - Last subtask: one of the args is a state variable, top(p)
    - Violates a restriction in Chapter 2
    - But many HTN algorithms don't need the restriction

TOHTN Planning Domain (continued)

- Goal task: goal(cargo(r)=c)
  - Get c onto r
  - Subtask of m2-put-in-pile
- We aren't doing classical planning, so we need a method:

m1-get-container(r, c)
  task: get-container(r, c)
  pre: cargo$(r) = c$
  sub: *// no subtasks*

m2-get-container(r, c, p, d)
  task: get-container(r, c)
  pre: cargo$(r) =$ nil, pile$(c) = p$, at(p, d)
  sub: navigate(r, d), uncover(c),
       take$(r, c,$ pos$(c), p, d)$

- Compound task uncover(c):
  - Subtask of m1-fetch
- Remove any containers that may be piled on top of c

m1-uncover(c)
  task: uncover(c)
  pre: top(pile$(c)) = c$
  sub: *// no subtasks*

m2-uncover(r, c, p, c', p', d)
  task: uncover(c)
  pre: pile$(c) = p$, top$(p) = c'$, $c' \neq c$,
       at(p, d), at(p', d), $p \neq p'$,
       loc$(r) = d$, cargo$(r) =$ nil
  sub: take$(r, c',$ pos$(c'), p, d)$,
       put$(r, c',$ top$(p'), p', d)$,
       uncover(c)

Poll: Can we rewrite m2-uncover to eliminate c'?
A. yes   B. no   C. don't know

TOHTN Planning Domain (continued)

- Compound task navigate(r,d):
  - Get robot r from current location to d
  - May require several move actions
- These methods are just for illustration, I don't recommend using them
  - Use a route planner instead
- Method for the case where loc(r) = d

  m1-navigate(r, d)
  task: navigate(r, d)
  pre: loc$(r) = d$
  sub: — *// no subtasks*

- Method for cases where loc(r) is adjacent to d

  m2-navigate(r, d', d)
  task: navigate(r, d)
  pre: adjacent(d', d), loc$(r) = d'$
  sub: move(r, d', d)

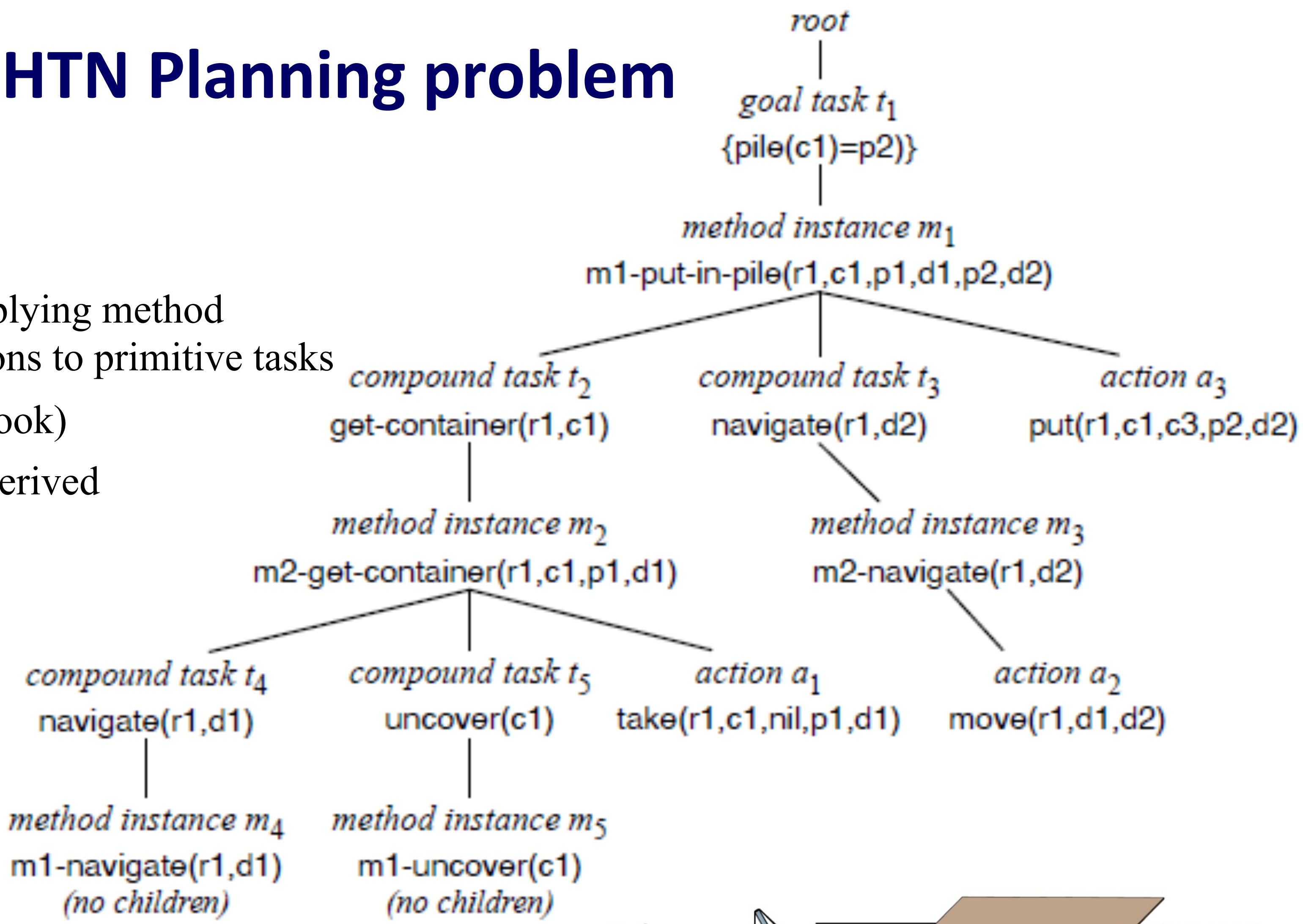
- Method for cases where loc(r) isn't adjacent to d

  m3-navigate(r, d', d)
  task: navigate(r, d)
  pre: loc$(r) \neq d$, ¬adjacent(loc$(r), d)$, adjacent(loc$(r), d')$
  sub: move$(r,$ loc$(r), d')$ *// primitive task*
  navigate(r, d) *// compound task*

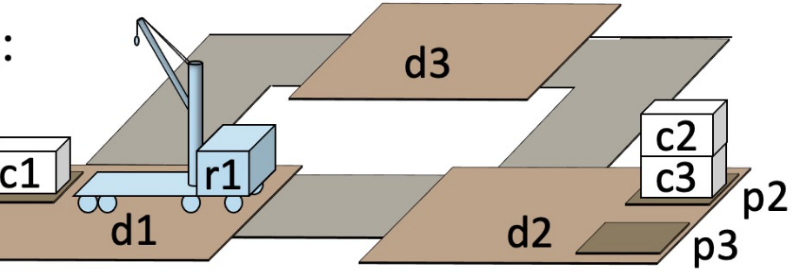
- Methods in $\mathcal{M}$:
  - m1-put-in-pile, m2-put-in-pile, m1-get-container, m2-get-container, m1-uncover, m2-uncover, m1-navigate, m2-navigate, m3-navigate
- Tasks in Σ:
  - Primitive: all instances of
    - move(r,l,m), take(r,c,l), put(r,c,l)
  - Compound: all instances of
    - put-in-pile(c,p), uncover(c), and navigate(r,d)
  - Goal tasks: all instances of goal(cargo(r) = c)

TOHTN Planning problem

- *Planning problem: $P = (\Sigma, s_0, T)$*
- *Solution* (detailed definition in book)
  - any executable plan produced by applying method instances to nonprimitive tasks, actions to primitive tasks
- *Refinement tree* (detailed definition in book)
  - tree showing how the solution was derived
  - Nodes:
    root, compound tasks, goal tasks,
    method instances, actions
  - Edges:
    root → top-level tasks,
    task → method instance,
    method instance → subtasks



s_0:



$P = (\Sigma, s_0, \langle$ {pile(c1)=p2} $\rangle)$

$\pi = \langle$take(r1,c1,c2,p1,d1), move(r1,d1,d2), put(r1,c1,c3,p2,d2)$\rangle$

Planning Algorithm

- Definitions

  $Achievers(s,t) = \{a \in Applicable(s) \mid \gamma(s,a) \models t\}$

  $Ground(\mathcal{M}) = \{$all ground instances of methods in $\mathcal{M}\}$

  $Refiners(s,t,\mathcal{M}) = \{m \in Ground(\mathcal{M}) \mid t$ is refinable by m in $s\}$

- Three cases: primitive, compound, goal task
  - Primitive task: apply action

    state s; $T = \langle t, t_2, \ldots, t_k \rangle$

    new state $\gamma(s,t)$; $T' = \langle t_2, \ldots, t_k \rangle$

  - Compound task: apply method instance

    state s; $T = \langle t, t_2, \ldots, t_k \rangle$

    method instance m

    $\langle \overbrace{u_1, \ldots, u_j}, t_2, \ldots, t_k \rangle$

  - Goal task: apply method instance or action

```
TO-HTN-Forward(Σc, M, s, T)
    if T is empty then return ⟨⟩
    t ← the first element of T;  T′ ← the rest of T
1   M ← HTN-Get-Candidates(Σc, M, s, t)
    if M = ∅ then return failure
    nondeterministically choose m ∈ M
    switch m do
2       case m is an action do
            π ← TO-HTN-Forward(Σc, M, γ(s,m), T′)
            if π ≠ failure then return m·π
            else return failure
3       case m is a ground method do
            return TO-HTN-Forward(Σc, M, s, subtasks(m)·T′)

HTN-Get-Candidates(Σc, M, s, t)
    switch t do
4       case t is an action do
            if t is applicable in s then M ← {a}
            else M ← ∅
5       case t is a compound task do M ← Methods(s, t, M)
6       case t is a goal task do
            M ← Methods(s, t, M) ∪ Actions(s, t)
7           if s ⊨ t then M ← M ∪ {null}
    return M
```

Planning Algorithm

- Most implementations do depth-first
  - Can use heuristic function, but the ones in Chapter 3 will probably need modification
  - Primitive task: apply action

$$\text{state } s; \quad T = \langle t, t_2, \ldots, t_k \rangle$$

$$\text{new state } \gamma(s,t)\ ; \quad T = \langle t_2, \ldots, t_k \rangle$$

  - Compound task: apply all method instances

$$\text{state } s; \quad T = \langle t, t_2, \ldots, t_k \rangle$$

method instance m

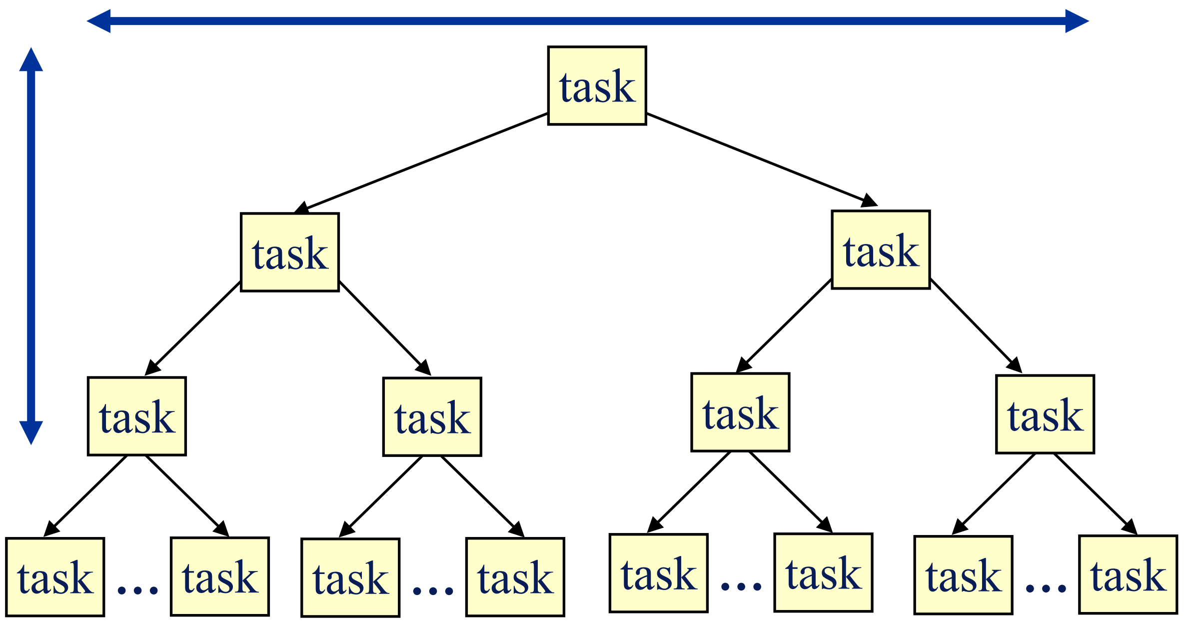
$$\langle \overbrace{u_1, \ldots, u_j}^{}, t_2, \ldots, t_k \rangle$$

  - Goal task: apply all method instances and actions

TO-HTN-Forward-Det($\Sigma_c, \mathcal{M}, s_0, T_0$)
  Frontier $\leftarrow \{(\langle\rangle, s_0, T_0)\}$ // {initial node}
  Expanded $\leftarrow \varnothing$
  while *Frontier* $\neq \varnothing$ **do**
1   select a node $v = (\pi, s, T) \in$ *Frontier*
    remove v from *Frontier* and add it to *Expanded*
    if $T = \langle\rangle$ **then** return π
    $t \leftarrow$ the first element of T; $T' \leftarrow$ the rest of T
    switch t **do**
      case t is an action **do**
        if t is applicable in s **then** *Children* $\leftarrow \{(\pi \cdot t, \gamma(s,t), T')\}$
        else *Children* $\leftarrow \varnothing$
      case t is a compound task **do**
        Children $\leftarrow \{(\pi, s, \mathrm{sub}(m) \cdot T') \mid m \in \mathit{Refiners}(s, t, \mathcal{M})\}$
      case t is a goal task **do**
        Children $\leftarrow \{(\pi \cdot a, \gamma(s,a), T') \mid a \in \mathit{Achievers}(s,t)\} \cup \{(\pi, s, \mathrm{sub}(m) \cdot T') \mid m \in \mathit{Refiners}(s, t, \mathcal{M})\}$
2   **if** $s \models t$ **then** *Children* $\leftarrow$ *Children* $\cup \{(\pi, s, T')\}$
    prune 0 or more nodes from *Children*, *Frontier* and *Expanded*
    Frontier $\leftarrow$ *Frontier* $\cup$ *Children*
  return failure

Search Direction, Search Strategies

- Down, then forward (progression)
  - totally-ordered compound tasks: SHOP, Pyhop, GTPyhop
  - partially-ordered compound tasks: SHOP2, SHOP3
  - totally-ordered goal tasks: GDP, GoDeL
  - acting, task refinement: RAE
  - Monte Carlo rollouts: UPOM
- Down and backward (regression)
  - plan-space planning: SIPE, O-Plan, UMCP
- Forward, then down (level 1, level 2, level 3, …)
  - AHA*: A* search
  - Bridge Baron 1997: game-tree generation



Complexity and Expressivity

- HTN planning is Turing-complete
  - There are HTN planning problems that are undecidable
- TOHTN planning is decidable, but is more expressive than classical planning
  - Every classical planning problem can be translated into an equivalent TOHTN planning problem
  - There are TOHTN planning problems that cannot be translated into classical planning problems
- Some subsets of TOHTN planning can be translated into classical planning problems
- Some subsets of TOHTN planning can be translated into propositional logic
- These translation techniques have been used to produce efficient TOHTN planners
- All of these are worst-case results
  - Most TOHTN planning problems are much simpler (e.g., in NP)
  - Example later

Pyhop

- A simple HTN planner written in Python
  - `import pyhop`
  - Depth-first version of TO-HTN-Forward with no goal tasks
  - Less than 150 lines of code, works in both Python 2 and 3
- Open-source software, Apache license
  - http://bitbucket.org/dananau/pyhop
- State: Python object that contains state variables
  - `s = gtpyhop.State('Current state')`
  - To say r1 is at d1 in state *s*:
    - `s.loc['r1'] = 'd1'`
- Actions and methods: ordinary Python functions
- Some limitations compared to most other HTN planners
  - I'll discuss later

Comparison

Task: transport(c,y,z) – transport c from y to z

- TOHTN method:

Method m_transport(r,x,c,y,z)
  Task: transport(c,y,z)
  Pre: loc(r) = x, cargo(r) = nil, loc(c) = y
  Sub: move(r,x,y), take(r,c,y), move(r,y,z), put(r,c,z)

Most HTN planners:

- Write in a planning language the planner can read and analyze
- Can have parameters not mentioned in the task
    - robot r, location x
  - Backtrack over multiple possibilities
- Planner knows in advance what the subtasks are
  - Helps with implementing heuristic functions

- Pyhop method: ordinary Python function
  - Args: state s and the task parameters

```
def m_transport(s,c,y,z):
  (r,x) = find_suitable_robot('transport',s,c,y,z)
  if r != 'failure':
    return [('move',r,x,y), ('take',r,c,y), \
      ('move',r,y,z), ('put',r,c,z)]
  else: return False
```

- Advantages
  - Don't need to learn a planning language: write methods and actions in Python
- Disadvantages:
  - Planner doesn't know in advance what the subtasks are
    - How to implement a heuristic function?
  - What about parameters not mentioned in the task?

GTPyhop

- GTPyhop (2021):
- Like Pyhop, but has both compound tasks and goal tasks
  - declare *task methods* for compound tasks
  - declare *goal methods* for goal tasks
- Open-source: https://github.com/dananau/GTPyhop
- Mostly backward-compatible with Pyhop

Two kinds of goals:

- *Unigoal*: a single atom
  - represented as a triple (*name, arg, value*)

    `('pos', 'a', 'b')`
  - goal: get to a state *s* in which

    `s.pos['a']=='b'`
- *Multigoal*: a conjunction of atoms
  - represented as a state-like object

    ```
    g = gtpyhop.Multigoal('Sussman goal')
    g.pos = {'a':'b', 'b':'c'}
    ```
  - goal: get to a state *s* in which

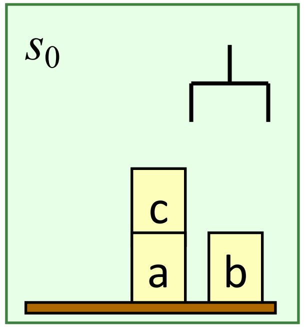
    `s.pos['a']=='b' and s.pos['b']=='c'`

Example: Blocks World

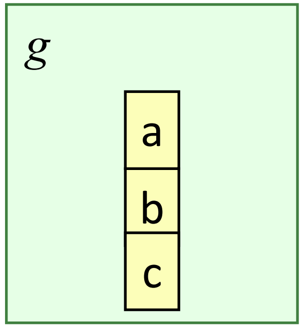
- Simple classical planning domain
  - Blocks, robot hand for stacking them, infinitely large table
- State-variable notation:
- pickup(x)
  - pre: loc(x)=table, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x
- putdown(x)
  - pre: holding=x
  - eff: holding=nil, loc(x)=table, clear(x)=T
- unstack(x,y)
  - pre: loc(x)=y, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x, clear(y)=T
- stack(x,y)
  - pre: holding=x, clear(y)=T
  - eff: holding=nil, clear(y)=F, loc(x)=y, clear(x)=T

- The "Sussman anomaly"
  - Planning problem that caused problems for early classical planners

s_0 = {clear(a)=F, clear(b)=T,
clear(c)=T,
loc(a)=table,
loc(b)=table, loc(c)=a,
holding(hand)=nil}



g = {loc(a)=b, loc(b)=c}

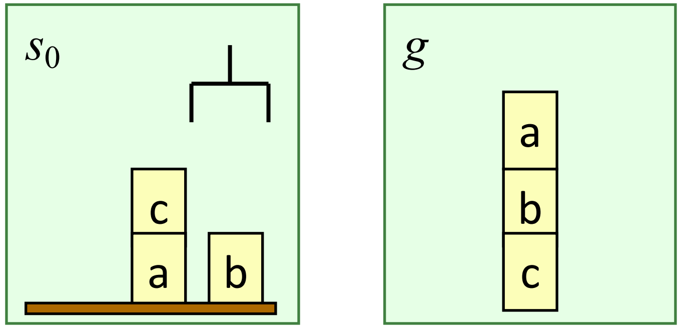


π = ⟨unstack(c,a), putdown(c),
pickup(b), stack(b,c),
pickup(a), stack(a,b)⟩

Domain-Specific Algorithm

loop

if there's clear block that needs to be moved
and it can immediately be moved to a place
where it won't need to be moved again
then move it there
else if there's a clear block that needs to be moved
then move it to the table
else if the current state satisfies the goal
then return success
else return failure



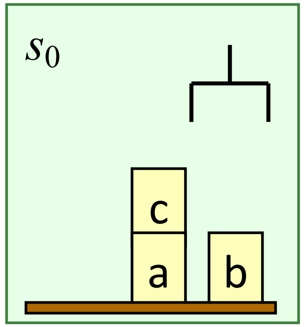
$\pi = \langle$unstack(c,b),
putdown(c),
pickup(b),
stack(b,c),
pickup(a),
stack(a,b)$\rangle$

- Situations in which c needs to be moved:
  - loc(c)=d, goal contains loc(c)=e, and $d \neq e$
  - loc(c)=d, d is a block, goal contains loc(b)=d for some $b \neq c$
  - loc(c)=d and d is a block that needs to be moved
- Can extend this to include situations involving clear and holding

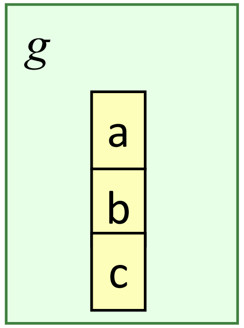
- Sound, complete, guaranteed to terminate
- Runs in time $O(n^3)$
  - Can be modified to run in time $O(n)$
- Often finds optimal (shortest) solutions, but sometimes only near-optimal
  - For block-stacking problems, PLAN-LENGTH is NP-complete
- Can implement as GTPyhop methods

States and Goals

Initial state:



Goal:



- A State object to hold all the state-variable bindings:

```
s0 = gtpyhop.State('Sussman initial state')
s0.pos = {'a':'table', 'b':'table', 'c':'a'}
s0.clear = {'a':False, 'b':True, 'c':True}
s0.holding = {'hand':False}
```

  - `s0.pos = {'a':'table’, 'b':'table', 'c':'a’}`

    is Python *dictionary* notation for

```
s0.pos['a'] = 'table'
s0.pos['b'] = 'table'
s0.pos['c'] = 'a'
```

Two ways to write goals:

- *Unigoal*: a single atom
  - represented as a triple (*name, arg, value*)

    `('pos', 'a', 'b')`
  - get to a state *s* in which

    `s.pos['a']=='b'`
- *Multigoal*: a conjunction of atoms
  - represented as a state-like object

```
g = gtpyhop.Multigoal('Sussman goal')
g.pos = {'a':'b', 'b':'c'}
```

  - get to a state *s* in which

    `s.pos['a']=='b' and s.pos['b']=='c'`

Actions

- pickup(x)
  - pre: loc(x)=table, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x
- putdown(x)
  - pre: holding=x
  - eff: holding=nil, loc(x)=table, clear(x)=T
- unstack(x,y)
  - pre: loc(x)=y, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x, clear(y)=T
- stack(x,y)
  - pre: holding=x, clear(y)=T
  - eff: holding=nil, clear(y)=F, loc(x)=y, clear(x)=T

- Args: current state s, block x

```
def pickup(s,x):
  if s.pos[x] == 'table' \
       and s.clear[x] == True \
       and s.holding['hand'] == False:
     s.pos[x] = 'hand'
     s.clear[x] = False
     s.holding['hand'] = x
     return s
```

Preconditions: *if* test

Effects: modify variable bindings in s

```
def putdown(s,x):
  if s.holding['hand'] = x:
       s.pos[x] = 'table'
     s.clear[x] = True
     s.holding['hand'] = False
     return s

gtpyhop.declare_actions(pickup,putdown)
```

- Tell GTPyhop these are actions

Poll. How many arguments does the unstack task have?
A. 1 B. 2 C. 3 D. other E. don't know

Task Methods

- `m_take`: method to pick up a clear block *x*, regardless of what it's on
  - Args: current state *s*, block *x*.
  - if *x* is clear:
    - return one task list if *x* is on the table, another task list if *x* isn't on the table
  - Else return nothing
    - means method is inapplicable
    - (also OK to return false like Pyhop does)
  - Declare m_take to be a task method
    - relevant for all tasks of the form `(take, ...)`
- `m_put`: similar

```
def m_take(s,x):
  if s.clear[x] == True:
    if s.pos[x] == 'table':
      return [('pickup', x)]
    else: return [('unstack',x,s.pos[x])]

gtpyhop.declare_task_methods('take',m_take)

def m_put(s,x,y):
  if s.holding['hand'] == x:
    if y == 'table': return [('putdown',x)]
    else: return [('stack',x,y)]
  else: return False
```
optional

```
gtpyhop.declare_task_methods('put',m_put)
```
declare relevant for task 'put'

Poll. In a TOHTN planning domain, how many methods would we need for take?

A. 1 B. 2 C. 3 D. other E. don't know

Goal Methods

loop

if there's clear block that needs to be moved
and it can immediately be moved to a place
where it won't need to be moved again

then move it there

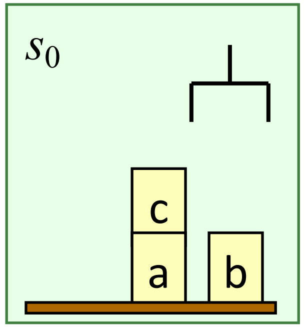
else if there's a clear block that needs to be moved

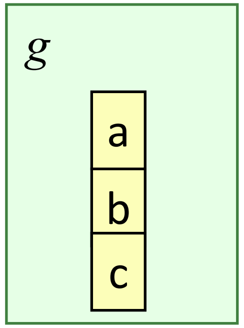
then move it to the table

else if the current state satisfies the goal

then return success

else return failure






```
def m_moveblocks(s, mgoal):
  for x in all_clear_blocks(s):
    stat = status(x, s, mgoal)
    if stat == 'move-to-block':
      where = mgoal.pos[x]
      return [('take',x), ('put',x,where), mgoal]
    elif stat == 'move-to-table':
      return [('take',x), (put,x,'table'), mgoal]
    for x in all_clear_blocks(s):
      if status(x,s,mgoal) == 'waiting' \
          and s.pos[x] != 'table':
        return [('take',x), ('put',x,'table'), mgoal]
    return [ ]
```

s = current state
mgoal = a multigoal
boldface: helper functions

```
gtpyhop.declare_multigoal_methods(m_moveblocks)
```

} declare relevant for every multigoal

```
gtpyhop.find_plan(s0,g)
```

returns

```
[('unstack','c','a'), ('putdown','c'),
('pickup','b’), ('stack','b','c'),
('pickup','a'), ('stack','a','b')]
```

Poll. Can we rewrite this as a set of TOHTN methods?

A. Yes   B. No

C. Don't know

POHTN (Partially Ordered HTN) Planning

- Sometimes we don't want to specify a total ordering on tasks
- Represent partially ordered tasks as a *task network*:
  - a pair $\mathcal{T} = (T, \prec)$
  - T is a set of task nodes
  - $\prec$ is a partial ordering of T
- *Task node:* a pair $\tau = (l, t)$
  - t is a task
  - l is a name that uniquely identifies τ
- Need labels so we can have multiple occurrences of t
- *POHTN Method*: a tuple

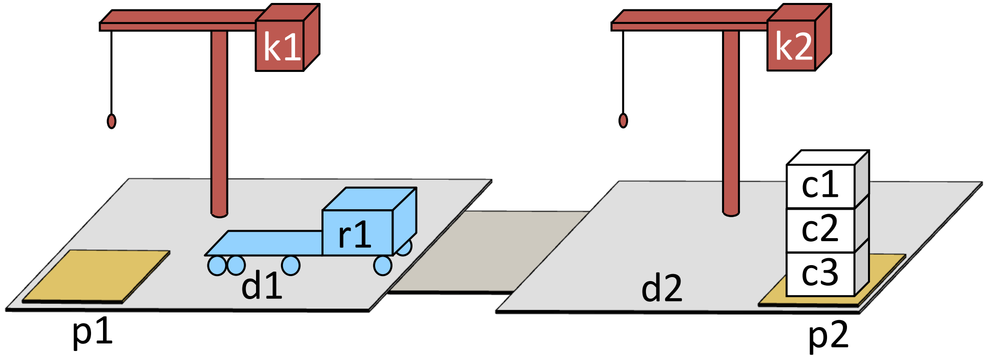
  $(head, task, pre, sub, \prec)$

- As usual, write POHTN methods as pseudocode:

  method-name(*args*)
  Task: *nonprimitive task*
  Pre: *preconditions*
  Sub: *subtask nodes*
  $\prec$: *partial ordering of the subtask nodes*

- TOHTN planning is a special case of POHTN planning
  - $\prec$ is a total ordering
- Details on the following slides
  - We'll skip them

Example POHTN Problem



- Σ_c : DWR with cranes attached to loading docks, not robots

unstack(k, c, c', p, d) *// take container c from pile p*
pre: at(k, d), at(p, d), holding(k) = nil, pos$(c) = c'$, top$(p) = c$
eff: holding$(k) \leftarrow c$, pos$(c) \leftarrow k$, pile$(c) \leftarrow$ nil, top$(p) \leftarrow c'$

stack(k, c, c', p, d) *// put container c onto pile p*
pre: at(k, d), at(p, d), holding$(k) = c$, top$(p) \leftarrow c'$
eff: holding$(k) \leftarrow$ nil, pos$(c) = c'$, pile$(c) \leftarrow p$, top$(p) = c$

unload(k, c, r, d) *// take container c from robot r*
pre: at(k, d), holding$(k) = c$, loc$(r) = d$
eff: cargo$(r) \leftarrow c$, pos$(c) \leftarrow r$, holding$(k) \leftarrow$ nil

load(k, c, r, d) *// put container c onto robot r*
pre: at(k, d), holding(k) = nil, loc$(r) = d$, cargo$(r) = c$
eff: pos$(c) \leftarrow k$, holding$(k) \leftarrow c$, cargo$(r) \leftarrow$ nil

Poll. How many solution plans?
A. 1 B. 2 C. 3 D. 4 E. other

- $P = (\Sigma, s_0, (T, \prec))$
  - T = {put-on-robot(c1,r1)}; $\prec = \emptyset$

- $\Sigma = (\Sigma_c, \mathcal{M})$
  - $\mathcal{M}$: three methods

m1-put-on-robot(k, c, c', r, d, p)
task: put-on-robot(c, r)
pre: cargo(r) = nil, top$(p) = c$, at(p,d), attached(k,d), holding(k) = nil
sub: (t1, navigate(r, d))
(t2, unstack(k,c,c',p,d))
(t3, load(k,r,c,d))
$\prec$: t1 $\prec$ t3, t2 $\prec$ t3

m1-navigate(r, d)
task: navigate(r, d)
pre: loc$(r) = d$
sub: *// none*
$\prec$: *// none*

m2-navigate(r, d', d)
task: navigate(r, d)
pre: adjacent(d', d), loc$(r) = d'$
sub: (t1, move(r, d', d))
$\prec$: *// none*

Solution Trees

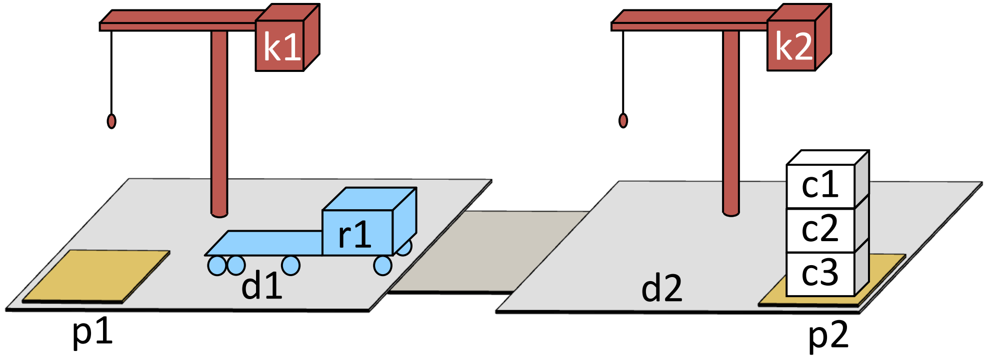
$\mathcal{R}_1$:

- *root*
  - *compound task* put-on-robot(c1,r1)
    - *method instance* m1-put-on-robot(c1,r1)
      - *compound task* navigate(r1,d2)
        - *method instance* m2-navigate(r1,d2)
          - *action* move(r1,d1,d2)
      - *action* unstack(k2,c1,c2,p2,d2)
      - *action* load(k2,c1,r1,d2)

$\mathcal{R}_2$:

- *root*
  - *compound task* put-on-robot(c1,r1)
    - *method instance* m1-put-on-robot(c1,r1)
      - *action* unstack(k2,c1,c2,p2,d2)
      - *compound task* navigate(r1,d2)
        - *method instance* m2-navigate(r1,d2)
          - *action* move(r1,d1,d2)
      - *action* load(k2,c1,r1,d2)

- π_1 = move(r1,d1,d2), unstack(k2,c1,c2,p2,d2), load(k2,c1,r1,d2)
- π_2 = unstack(k2,c1,c2,p2,d2), move(r1,d1,d2), load(k2,c1,r1,d2)



- $P = (\Sigma, s_0, (T, \prec))$
  - T = {put-on-robot(c1,r1)}; $\prec = \emptyset$

m1-put-on-robot(k, c, c', r, d, p)
task: put-on-robot(c, r)
pre: cargo(r) = nil, top(p) = c, at(p,d), attached(k,d), holding(k) = nil
sub: (t1, navigate(r, d))
(t2, unstack(k,c,c',p,d))
(t3, load(k,r,c,d))
$\prec$: t1 $\prec$ t3, t2 $\prec$ t3

m1-navigate(r, d)
task: navigate(r, d)
pre: loc(r) = d
sub: // *none*
$\prec$: // *none*

m2-navigate(r, d', d)
task: navigate(r, d)
pre: adjacent(d', d), loc(r) = d'
sub: (t1, move(r, d', d))
$\prec$: // *none*

Planning Algorithm

Three cases: primitive, compound, goal task

- Primitive task node: apply action

$\mathcal{T} = (T, \prec)$, $T = \{\tau, \tau_2, \ldots, \tau_k\}$, nothing precedes τ

state s_0 i.e., $\nexists \tau' \in T$ s.t. $\tau' \prec \tau$

new state $\gamma(s_0, \tau)$; $\{\tau_2, \ldots, \tau_k\}$

- Compound task node: apply method instance

$\mathcal{T} = (T, \prec)$, $T = \{\tau, \tau_2, \ldots, \tau_k\}$, nothing precedes τ

method instance m i.e., $\nexists \tau' \in T$ s.t. $\tau' \prec \tau$

state s_0 $\{\overbrace{\upsilon_1, \ldots, \upsilon_j}, \tau_2, \ldots, \tau_k\}$

make $\upsilon_1, \ldots, \upsilon_j$ precede everything τ preceded

PO-HTN-Forward$(\Sigma_c, \mathcal{M}, s, \mathcal{T})$
  if $\mathcal{T}$ is empty **then return** $\langle\rangle$
1 **nondeterministically choose** a node τ in $\mathcal{T}$ that has no predecessors in $\mathcal{T}$
  foreach τ' in $\mathcal{T}$ that has no predecessors in $\mathcal{T}$ **do**
2   **if** $\tau' \neq \tau$ **then** add ordering constraints to $\mathcal{T}$ to make $\tau \prec \tau'$
  $t \leftarrow$ task(τ)
  $M \leftarrow$ HTN-Get-Candidates$(\Sigma_c, \mathcal{M}, s, t)$
  if $M \neq \varnothing$ **then**
    nondeterministically choose $m \in M$
    if m is an action **then**
      $\pi \leftarrow$ PO-HTN-Forward$(\Sigma_c, \mathcal{M}, \gamma(s, a), \mathcal{T} \setminus \{\tau\})$
      if $\pi \neq$ failure **then return** $a \cdot \pi$
    else if m is a ground method **then**
      return PO-HTN-Forward$(\Sigma_c, \mathcal{M}, s, \mathit{refine}(\mathcal{T}, \tau, m))$
  return failure

Summary

- 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
  - TOHTN: tasks are totally ordered
    - Planning algorithm: TO-HTN-Forward
  - POHTN: tasks are partially ordered
    - Planning algorithm: PO-HTN-Forward
- Pyhop: Python implementation of total-order HTN planning
  - Open source: `http://bitbucket.org/dananau/pyhop`
- GTPyhop: Python implementation of HTN + HGN planning
  - Open source: `https://github.com/dananau/GTPyhop`
- Examples: DWR, blocks world, cranes