Lecture 15

CS 6260 Automated Planning and Learning

Planners for TOHTN Planning

Department of Computer Science and Engineering

Indian Institute of Technology Madras

Some material based on lecture slides for Automated Planning and Acting by Nau/Roberts.

Logistics and Announcement

Logistics

- 3 Exams:
  1. [Quiz 1] Feb 19: 08:00 AM – 08:50 AM
  2. [Quiz 2] Apr 09: 08:00 AM – 08:50 AM
  3. [End Sem] May 12: 08:00 AM – 08:50 AM

- Office Hours: 1 hour after class every week Tu, W, Th

- Tomorrow's Class: 12:00 PM – 12:50 PM
  - Slot interchange with slot D

Talk Announcement

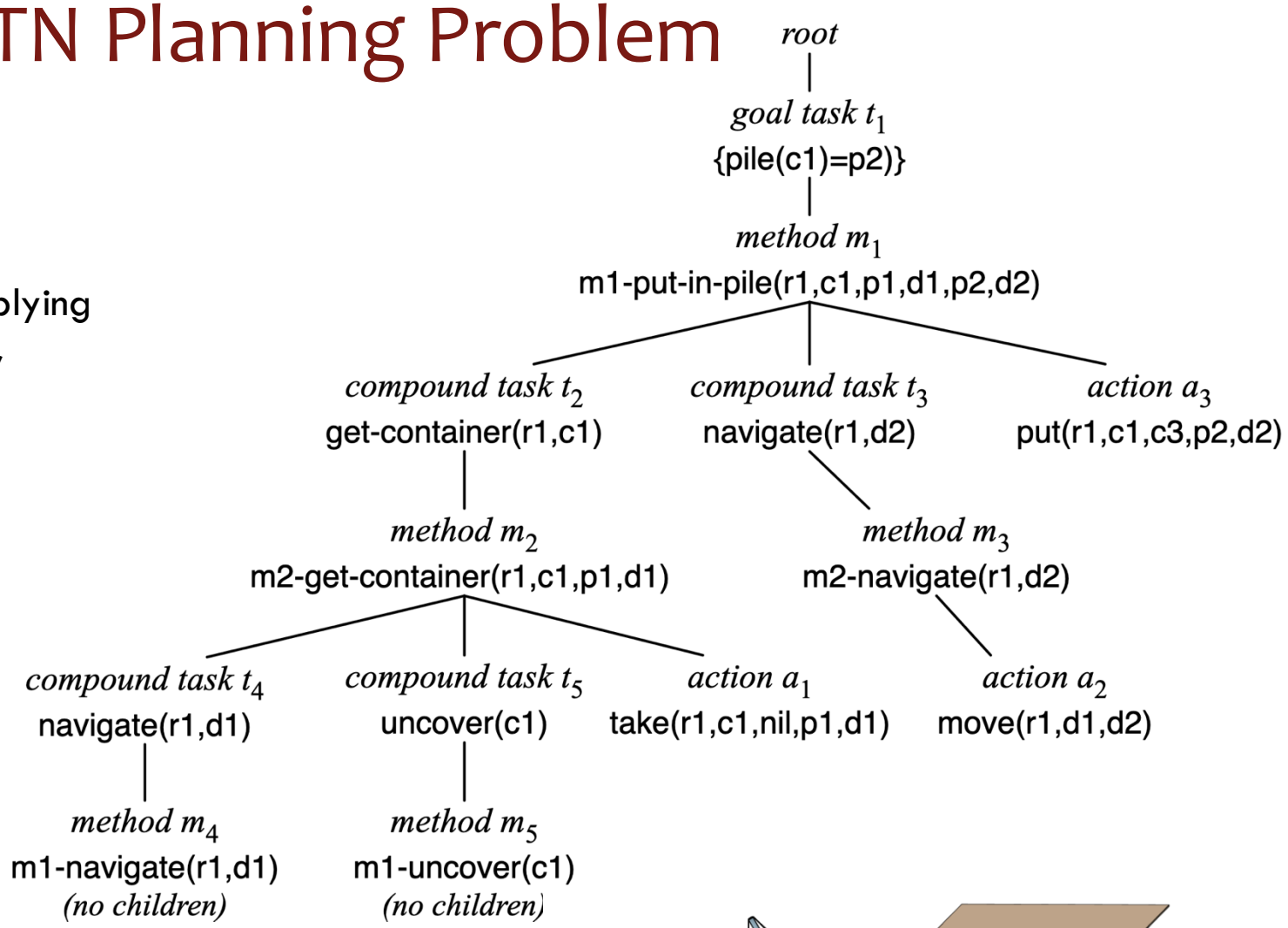
- **Title:** Integration of Structured Reasoning and Data-driven Learning for Acting and Planning
- **Date:** March 24, 2026
- **Time:** 09:30 AM – 10:30 AM
- Online link will be posted on Moodle

Dr. Sunandita Patra
J. P. Morgan AI Research, USA
https://sunanditapatra.wixsite.com/camp

Recap: TO HTN Planning Solution

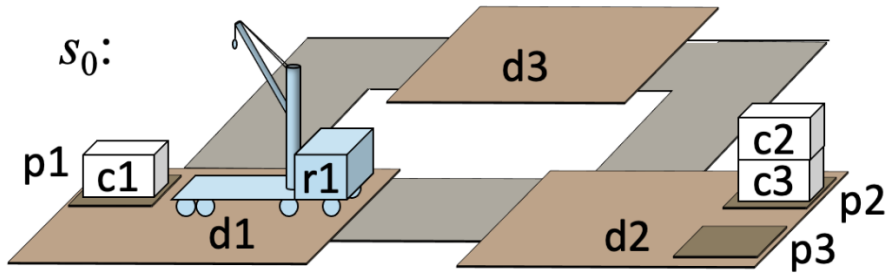
TOHTN Planning Problem

- *Planning problem:* $P = (\Sigma, s_0, T)$
- *Solution*
  - any executable plan produced by applying method instances to nonprimitive tasks, actions to primitive tasks



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

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

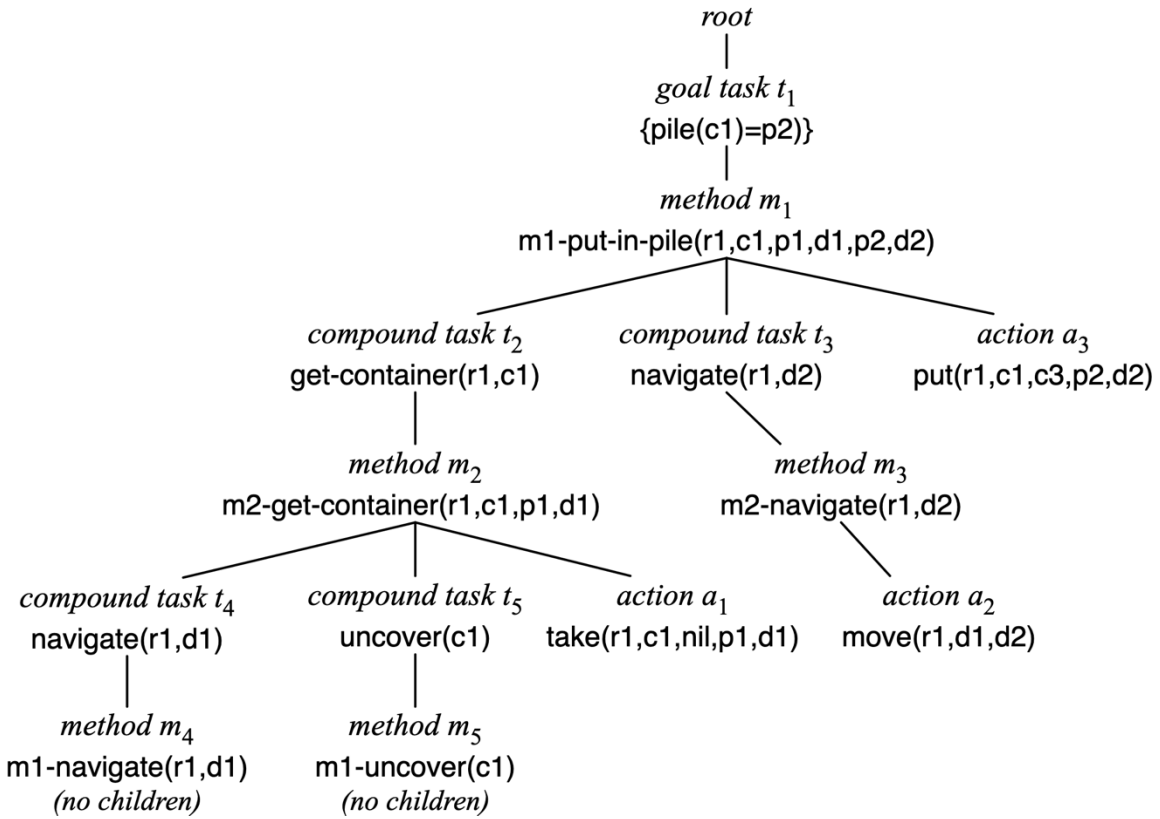


Planning Algorithm

- Definitions

t= get-container(r1,c1)
T'= ⟨navigate(r1,d2), put(r1,c1,c3,p2,d2)⟩

M = HTN-Get-Candidates(Σ_c,$\mathcal{M}$, s_0, get-container(r1,c1))



TO-HTN-Forward($\Sigma_c, \mathcal{M}, s, T$)
  if T is empty **then return** $\langle\rangle$
  $t \leftarrow$ the first element of T; $T' \leftarrow$ the rest of T
1 $M \leftarrow$ HTN-Get-Candidates($\Sigma_c, \mathcal{M}, s, t$)
  if $M = \varnothing$ **then return** failure
  nondeterministically choose $m \in M$
  switch m **do**
2 **case** m is an action **do**
    $\pi \leftarrow$ TO-HTN-Forward($\Sigma_c, \mathcal{M}, \gamma(s,m), T'$)
    if $\pi \neq$ failure **then return** $m \cdot \pi$
    else return failure
3 **case** m is a method instance **do**
    return TO-HTN-Forward($\Sigma_c, \mathcal{M}, s$, sub$(m) \cdot T'$)

HTN-Get-Candidates($\Sigma_c, \mathcal{M}, s, t$)
  switch t **do**
4 **case** t is an action **do**
    if t is applicable in s **then** $M \leftarrow \{a\}$
    else $M \leftarrow \varnothing$
5 **case** t is a compound task **do** $M \leftarrow$ *Refiners*$(s,t,\mathcal{M})$
6 **case** t is a goal task **do**
    $M \leftarrow$ *Refiners*$(s,t,\mathcal{M}) \cup$ *Achievers*(s,t)
7   **if** $s \models t$ **then** $M \leftarrow M \cup \{$null$\}$
  return $\mathcal{M}$

Planners for HTN Planning

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
- 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
- Open-source software, Apache license
  - http://bitbucket.org/dananau/pyhop

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'
    ```

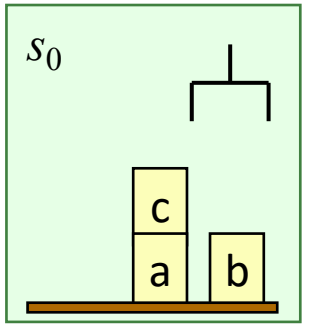
Example: Blocks World

- Simple classical planning domain
  - Blocks, robot hand for stacking them, infinitely large table
- State-variable notation:

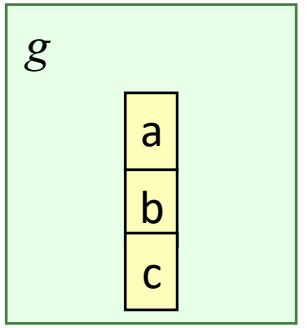
- pickup(x)
  - pre: loc(x)=table, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x
- putdown(x)
  - pre: holding=x
  - eff: holding=nil, loc(x)=table, clear(x)=T
- unstack(x,y)
  - pre: loc(x)=y, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x, clear(y)=T
- stack(x,y)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



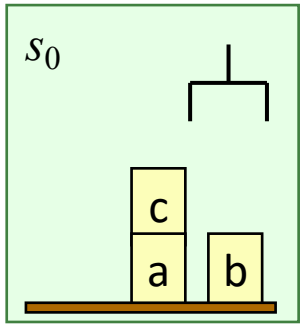
g = {loc(a)=b, loc(b)=c}

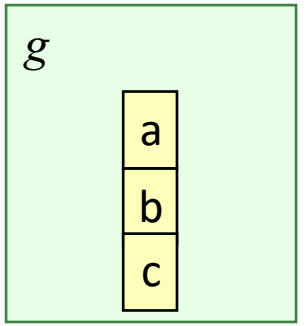


π = ⟨unstack(c,a), putdown(c),
pickup(b), stack(b,c),
pickup(a), stack(a,b)⟩

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





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

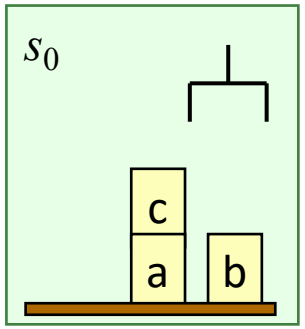
- Situations in which *c* needs to be moved:
  - loc(*c*)=*d*, goal contains loc(*c*)=*e*, and *d* ≠ *e*
  - loc(*c*)=*d*, *d* is a block, goal contains loc(*b*)=*d* for some *b* ≠ *c*
  - loc(*c*)=*d* and *d* is a block that needs to be moved

  Can extend this to include situations involving clear and holding

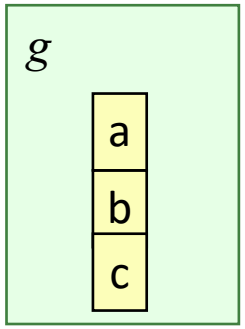
- Sound, complete, guaranteed to terminate
- Runs in time $O(n^3)$
  - Can be modified to run in time $O(n)$
- Often finds optimal (shortest) solutions, but sometimes only near-optimal
  - For block-stacking problems, PLAN-LENGTH is NP-complete
- Can implement as GTPyhop methods

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

Ques. How many arguments does the unstack task have for a GTPyhop implementation?

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

- 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

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)
```

Preconditions: *if* test

Effects: modify variable bindings in s

- Tell GTPyhop these are actions

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'

Goal Methods

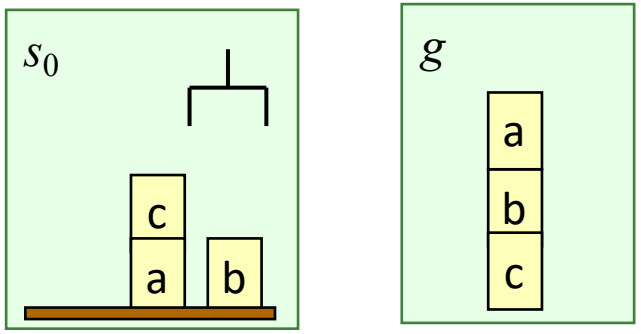
loop

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



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

```
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 [ ]

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')]
```

Socks and Shoes

- Suppose you have actions to put on socks or shoes.
  - A sock must be placed on the foot before a shoe, where $\prec$ indicates an ordering constraint
  - There are two feet
- Now suppose the following Partial Plan:
  - Left sock $\prec$ left shoe
  - Right sock $\prec$ right shoe

- If it helps, use the following naming:
  - Left Sock: LK
  - Left Shoe: LO
  - Right Sock: RK
  - Right Shoe: RO