CS 4649/7649
Robot Intelligence: Planning

Hierarchical Network Planning

Sungmoon Joo

School of Interactive Computing
College of Computing
Georgia Institute of Technology

*Slides based in part on Dr. Dana S. Nau and Jose Luis Ambite's lecture slides

Course Info.

- **Course Website: joosm.github.io/RIP2014**
- **Course Wiki: github.com/RIP2014/RIP2014Wiki/wiki**
  - add your contact info, start grouping/filling in project ideas, etc.
  - pending invitations
- **Next Tuesday (Sept. 16)**
  - IROS @ Chicago
  - Substitute video 'lecture' on final project

Domain Independent Heuristics

Concept

- Solve a relaxed form of the problem
- Use to evaluate states for solving original problem

Approaches

- Assume complete subgoal independence
- Assume no negative interactions

FF vs. HSP

- FF's heuristic is more informed
- Takes into account positive interactions by propagating constraints back through the layers
- FF uses "Enforced Hill Climbing" which appears to work well
  EHC = Hill Climbing + BFS when stuck

Really though – which one is faster?

International Planning Competitions at ICAPS

PDDL

How to use PDDL for planning problems?

- Two files

- A domain file: predicates and actions
- A problem file: objects, initial state and goal specification

```
(define (domain hanoi-domain)
  (:requirements :equality)
  (:predicates (disk ?x) (smaller ?x ?y) (on ?x ?y) (clear ?x))
  (:action move-disk
       :parameters (?disk ?below-disk ?new-below-disk)
       :precondition (and (disk ?disk)
                  (smaller ?disk ?new-below-disk)
                  (not (= ?new-below-disk ?below-disk))
                  (not (= ?new-below-disk ?disk))
                  (not (= ?below-disk ?disk))
                  (on ?disk ?below-disk)
                  (clear ?disk)
                  (clear ?new-below-disk))
       :effect (and (clear ?below-disk)
               (on ?disk ?new-below-disk)
               (not (on ?disk ?below-disk))
               (not (clear ?new-below-disk)))))
```


PDDL

How to use PDDL for planning problems?

- Two files

- A domain file: predicates and actions
- A problem file: objects, initial state and goal specification

```
(define (problem hanoi-problem)
  (:domain hanoi-domain)
  (:objects p1 p2 p3 d1 d2 d3)
  (:init (smaller d1 p1) (smaller d2 p1) (smaller d3 p1)
(smaller d1 p2) (smaller d2 p2) (smaller d3 p2) (smaller
d1 p3) (smaller d2 p3) (smaller d3 p3) (smaller d1 d2)
(smaller d1 d3) (smaller d2 d3) (clear p1) (clear p2)
(clear d1) (disk d1) (disk d2) (disk d3) (on d1 d2) (on d2
d3) (on d3 p3))
  (:goal (and (on d1 d2) (on d2 d3) (on d3 p1))))
```

PDDL

- Domain description

Domain name

Requirements

Actions

Predicates

```
(define (domain hanoi-domain)
 (:requirements :equality)
 (:predicates (disk ?x) (smaller ?x ?y) (on ?x ?y) (clear ?x))
 (:action move-disk
      :parameters (?disk ?below-disk ?new-below-disk)
      :precondition (and (disk ?disk)
                    (smaller ?disk ?new-below-disk)
                    (not (= ?new-below-disk ?below-disk))
                    (not (= ?new-below-disk ?disk))
                    (not (= ?below-disk ?disk))
                    (on ?disk ?below-disk)
                    (clear ?disk)
                    (clear ?new-below-disk))
      :effect (and (clear ?below-disk)
                (on ?disk ?new-below-disk)
                (not (on ?disk ?below-disk))
                (not (clear ?new-below-disk)))))
```

*Requirement flags allow a planner to quickly tell if it is likely to be able to hangle the domain
*Action effects can include universal quantifiers(i.e. forall), conditionals (e.g. when)

PDDL

- Problem description

Problem name

Objects

Domain

Initial state

Goal specification

```
(define (problem hanoi-problem)
 (:domain hanoi-domain)
 (:objects p1 p2 p3 d1 d2 d3)
 (:init (smaller d1 p1) (smaller d2 p1) (smaller
d3 p1) (smaller d1 p2) (smaller d2 p2) (smaller
d3 p2) (smaller d1 p3) (smaller d2 p3) (smaller
d3 p3) (smaller d1 d2) (smaller d1 d3) (smaller
d2 d3) (clear p1) (clear p2) (clear d1) (disk d1)
(disk d2) (disk d3) (on d1 d2) (on d2 d3) (on d3
p3))
 (:goal (and (on d1 d2) (on d2 d3) (on d3 p1))))
```

*domain must match the corresponding domain name

Summary

Search
- Important Element of Planning
- Heuristics: Admissible / Informed
- A* is desirable but often too expensive
- Best-First Search & Hill Climbing tend to work well

Heuristic Planning
- Domain Dependent Heuristics
- Domain Independent Heuristics
- Solving Relaxed Problems to Guide Actual Solution
- GraphPlan as a Method for Relaxing Problems!

Satisfiability
- Our Final Approach to Planning
- Most Efficient when combined with GraphPlan!

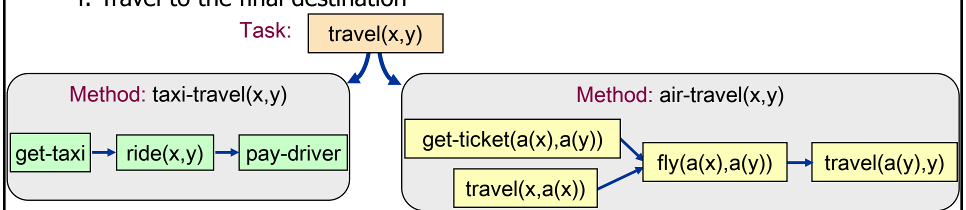
PDDL
- Domain
- Problem

Domain Knowledge for Planning

- For some planning domains you are familiar with, you may already know preferred ways of solving the planning problems

- Brute-force search over the entire search space vs. Search over a limited number of 'recipe's (i.e. preferred ways of doing something)

e.g. Travel to a destination that is far away
1. Buy a flight ticket from a local airport to a remote airport close to the destination
2. Travel to the local airport
3. Fly from the local airport to the remote airport
4. Travel to the final destination



Domain Knowledge Transfer

Two Approaches

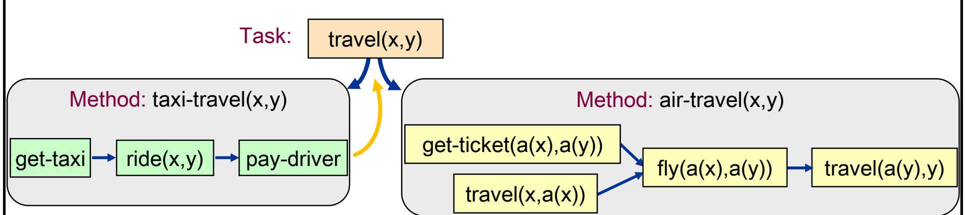
- Control rules:
  - Classical planning efficiency often suffers from combinatorial complexity
  - Write rules to prune every action that does not fit the recipe (i.e. cut the unpromising nodes)
  - Focus on identifying actions not to consider (i.e. actions that need to be pruned)
- Hierarchical Task Network(HTN):
  - Describe the actions and subtasks that do fit the recipe
  - Focus on identifying actions and tasks to consider
  - HTN methods are applied only when the preconditions are met

Abstract-search(u)
  if Terminal(u) then return(u)
  $u \leftarrow$ Refine(u) ;; *refinement step*
  $B \leftarrow$ Branch(u) ;; *branching step*
  [$B' \leftarrow$ Prune(B) ;; *pruning step* **]**
  if $B' = \emptyset$ then return(failure)
  nondeterministically choose $v \in B'$
  return(Abstract-search(v))
end

Abstract-search(u)
  if Terminal(u) then return(u)
  $u \leftarrow$ Refine(u) ;; *refinement step*
  [$B \leftarrow$ Branch(u) ;; *branching step* **]**
  $B' \leftarrow$ Prune(B) ;; *pruning step*
  if $B' = \emptyset$ then return(failure)
  nondeterministically choose $v \in B'$
  return(Abstract-search(v))
end

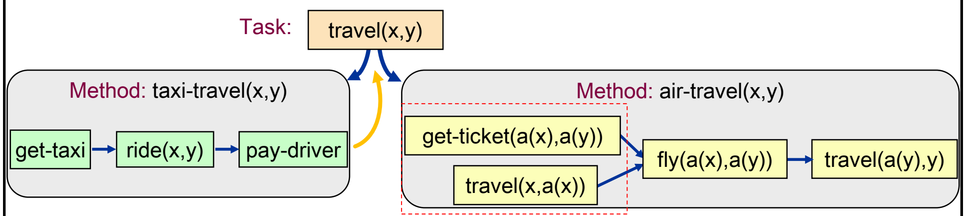
Problem Reduction

- Problem reduction
  - Capture hierarchical structure of the domain
  - Planning domain contains non-primitive actions and schemas for reduction
    : Methods to decompose tasks into subtasks (given by the design)
  - Tasks (activities) rather than goals
  - Enforce constraints: preconditions, task orders
    - E.g., taxi not good for long distances travel
  - Backtrack, if necessary



Problem Reduction

- Problem reduction
  - Capture hierarchical structure of the domain
  - Planning domain contains non-primitive actions and schemas for reduction
    : Methods to decompose tasks into subtasks (given by the design)
  - Tasks (activities) rather than goals
  - Enforce constraints: preconditions, task orders
    - E.g., taxi not good for long distances travel
  - Backtrack, if necessary



Hierarchical Task Network (HTN) Planning

- HTN planning domain
  - States(description of the current situation) and operators
  - Tasks: Activities to perform [primitive tasks & non-primitive(compound) tasks]
  - Methods: Ways to perform the activities, How to decompose compound tasks
    May be more than one method for the same task (e.g. taxi & flight)
- HTN planning problem
  - Domain
  - Initial state
  - Initial task network (tasks to accomplish, with some ordering of the tasks)
- HTN planners may be domain-specific or domain-configurable
- Domain-configurable HTN planner
  - Domain – independent planning algorithm
  - Domain – states, operators, tasks, and methods
  - Planning problem – domain, initial state, initial task network

Simple Task Network (STN) Planning

- A special case (simplified version) of HTN planning: Totally/Partially ordered
- State(list of ground atoms) and operator
  - The same as in classical planning
- Task
  - Two kinds of task symbols (and tasks):
    - primitive: tasks that we know how to execute directly
      - task symbol(name) is an operator name
    - non-primitive(compound): tasks that must be decomposed into subtasks
      - use methods
- Plan: Sequence of ground primitive tasks (operators)

*term: variable, constant, function expression

Simple Task Network (STN) Planning

- A special case (simplified version) of HTN planning: Totally/Partially ordered
- State(list of ground atoms) and operator
  - The same as in classical planning
- Task

  Definition (Task)

  A task is an expression of the form $t(r_1,r_2,\ldots,r_k)$

  where t is a task symbol (operator symbol for primitive task or a method symbol for compound task), and $r_1,r_2,\ldots,r_k$ are terms.
- Plan: Sequence of ground primitive tasks (operators)

*term: variable, constant, function expression

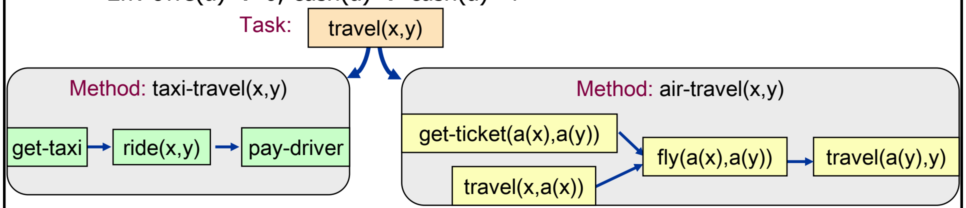
Primitive Task (Operator)

- Directly executable task. Primitive task is achieved by applying an operator

  get-taxi (a:Agents, x:Locations)
  - Pre:
  - Eff: loc(taxi) ← x

  ride(x:Locations, y:Locations)
  - Pre: loc(a) = x, loc(taxi) = x
  - Eff: loc(a) ← y, loc(taxi) ← y, owe(a) ← rate(x,y)

  pay-driver(a:Agents)
  - Pre: owe(a)=r, cash(a)>= r
  - Eff: owe(a) ← 0, cash(a) ← cash(a) - r

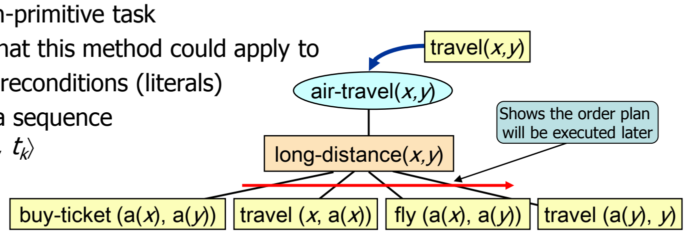


Method(totally ordered method)

- Totally ordered method: a 4-tuple

  m = (name(m), task(m), precond(m), subtasks(m))
  - name(m): an expression of the form $n(x_1,\ldots,x_n)$
    - $x_1,\ldots,x_n$ are parameters (variable symbols)
    - n is a name of the method (method symbol)
  - task(m): a non-primitive task

    task that this method could apply to
  - precond(m): preconditions (literals)
  - subtasks(m): a sequence of tasks $\langle t_1, \ldots, t_k \rangle$



t_1 t_2 t_3 t_4

air-travel(x,y)

task: travel(x,y)

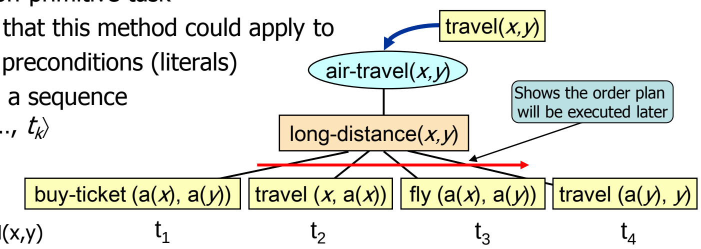
precond: long-distance(x,y)

subtasks (TO network): ⟨buy-ticket(a(x), a(y)), travel(x,a(x)), fly(a(x), a(y)), travel(a(y),y)⟩

- Task Network: TN = $(\{u_1,u_2,u_3,u_4\},\{(u_1,u_2),(u_2,u_3),(u_3,u_4)\})$ where $\forall i$, $u_i=t_i$
- For totally ordered TN, we usually write TN = $< t_1,t_2,t_3,t_4 >$

Method(totally ordered method)

- Totally ordered method: a 4-tuple
  m = (name(m), task(m), precond(m), subtasks(m))
  - name(m): an expression of the form $n(x_1,\ldots,x_n)$
    - $x_1,\ldots,x_n$ are parameters (variable symbols)
    - n is a name of the method (method symbol)
  - task(m): a non-primitive task
    task that this method could apply to
  - precond(m): preconditions (literals)
  - subtasks(m): a sequence of tasks $\langle t_1, \ldots, t_k \rangle$



air-travel(x,y)
task: travel(x,y)
precond: long-distance(x,y)

Definition (Simple Task Network)
A simple task network is an acyclic digraph W = (U,E), where U is the node set u ∈ U contains a task t_u. and E is the edge set that defines a partial ordering of U, e.g. u< v iff there is a path from u to v.

Methods

Definition (Applicable Method)
A method instance m is applicable in a state s if $precond^+(m) \subseteq s$ and $precond^-(m) \cap s = \emptyset$.

Definition (Relevant Method)
Let t be a task and m a method instance, if there is a substitution(of terms) σ such that $\sigma(t)$ = task(m), then m is relevant for t, and the decomposition of t by m under σ is $\delta(t,m,\sigma)$ = network(m). If m is totally ordered, we may write $\delta(t,m,\sigma)$ =subtasks(m).

(Example)
Let *t* be the non-primitive task move-stack(p1a,q), *s* the state of the world, and *m* be the method instance recursive-move(p1a,p1b,c11,c12). *m* is applicable to *s*, relevant for *t* under substitution σ = {q ← p1b}, and decomposes *t* into:

$\delta(t,m,\sigma)$ = <move-topmost-container(p1a,p1b),move-stack(p1a,p1b)>

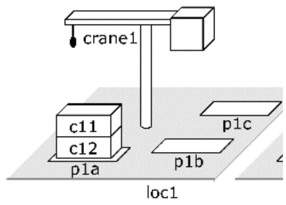
Example: Total-order Formulation

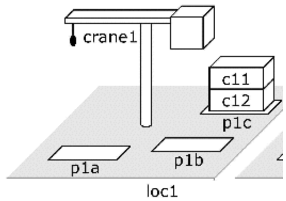
take-and-put$(c, k, l_1, l_2, p_1, p_2, x_1, x_2)$:
  task: move-topmost-container(p_1, p_2)
  precond: top(c, p_1), on(c, x_1), ; *true if p_1 is not empty*
  attached(p_1, l_1), belong(k, l_1), ; *bind l_1 and k*
  attached(p_2, l_2), top(x_2, p_2) ; *bind l_2 and x_2*
  subtasks: ⟨take(k, l_1, c, x_1, p_1), put(k, l_2, c, x_2, p_2)⟩

recursive-move(p, q, c, x):
  task: move-stack(p, q)
  precond: top(c, p), on(c, x) ; *true if p is not empty*
  subtasks: ⟨move-topmost-container(p, q), move-stack(p, q)⟩
  ;; the second subtask recursively moves the rest of the stack

do-nothing(p, q)
  task: move-stack(p, q)
  precond: top$(pallet, p)$; *true if p is empty*
  subtasks: ⟨⟩ ; *no subtasks, because we are done*

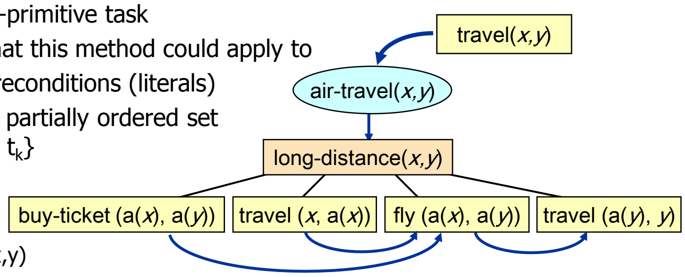
move-each-twice()
  task: move-all-stacks()
  precond: ; *no preconditions*
  subtasks: ; *move each stack twice:*
  ⟨move-stack(p1a,p1b), move-stack(p1b,p1c),
  move-stack(p2a,p2b), move-stack(p2b,p2c),
  move-stack(p3a,p3b), move-stack(p3b,p3c)⟩





Method(partially ordered method)

- Partially ordered method: a 4-tuple
  m = (name(m), task(m), precond(m), subtasks(m))
  - name(*m*): an expression of the form $n(x_1,\ldots,x_n)$
    - $x_1,\ldots,x_n$ are parameters (variable symbols)
    - n is a name of the method (method symbol)
  - task(*m*): a non-primitive task
    task that this method could apply to
  - precond(*m*): preconditions (literals)
  - subtasks(m): a partially ordered set of tasks $\{t_1, \ldots, t_k\}$

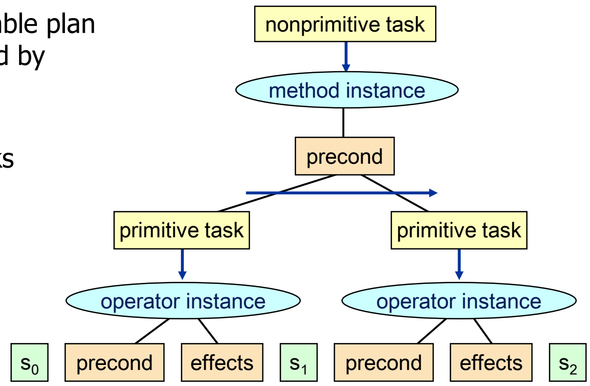


air-travel(x,y)
  task: travel(x,y)
  precond: long-distance(x,y)
  network: $(\{t_1$=buy-ticket(a(x),a(y)), t_2= travel(x,a(x)), t_3= fly(a(x), a(y))
  t_4= travel(a(y),y)}, $\{(t_1,t_3), (t_2,t_3), (t_3, t_4)\})$

STN Planning: Domain, Problem

- STN planning domain: operators, methods
- STN planning problem: domain, initial state, initial task network

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



STN Planning: Solution (Plan)

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

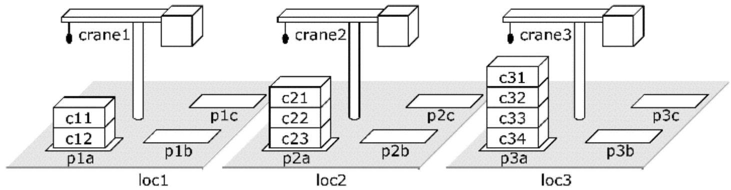
Definition (Solution Plan)
Let P = (s_0,w,O,M) be a STN planning problem. Then a plan π = <a_1, …, a_n> is a solution for P for the following cases:
Case 1 : w is empty. Then the empty plan π = <> is the solution.

Case 2: There is a primitive task node u $\in$ w that has no predecessor in w. Then π is a solution for P if a_1 is applicable to t_u in s_0 and the plan π = <a_2, …, a_n> is a solution of the planning problem P' = (γ(s_0,a_1),w-{u}, O, M)

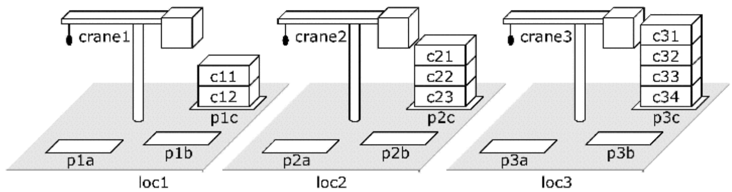
Case 3: There is a non-primitive task node u $\in$ w that no predecessor in w. Suppose there is an instance m of some method in M such that m is relevant for t_u and applicable in s_0 . Then plan π is a solution for P if there is a task network w' $\in$ δ(w,u,m,σ) such that π is a solution for (s_0,w',O,M).

STN Planning: Example

Suppose we want to move three stacks of containers in a way that preserves the order of the containers



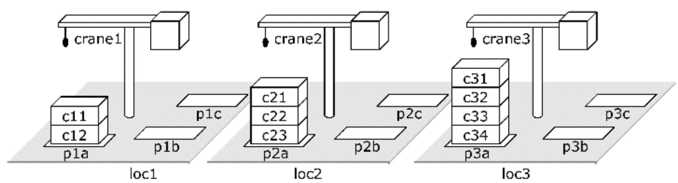
(a) initial state



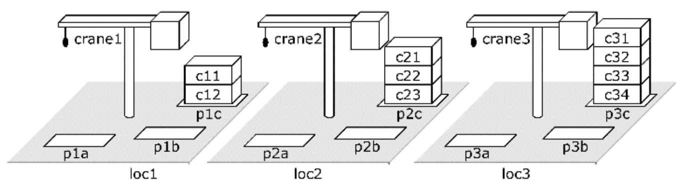
(b) goal

STN Planning: Example

- A way to move each stack:
  - first move the containers from *p* to an intermediate pile *r*
  - then move them from *r* to *q*



(a) initial state



(b) goal

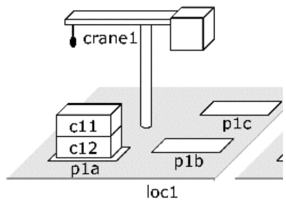
Example: Total-order Formulation

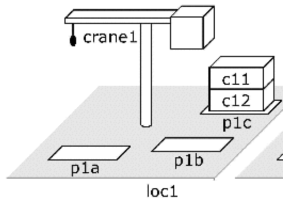
take-and-put($c, k, l_1, l_2, p_1, p_2, x_1, x_2$):
  task: move-topmost-container(p_1, p_2)
  precond: top(c, p_1), on(c, x_1), *; true if p_1 is not empty*
    attached(p_1, l_1), belong(k, l_1), *; bind l_1 and k*
    attached(p_2, l_2), top(x_2, p_2) *; bind l_2 and x_2*
  subtasks: ⟨take(k, l_1, c, x_1, p_1), put(k, l_2, c, x_2, p_2)⟩

recursive-move(p, q, c, x):
  task: move-stack(p, q)
  precond: top(c, p), on(c, x) *; true if p is not empty*
  subtasks: ⟨move-topmost-container(p, q), move-stack(p, q)⟩
    ;; the second subtask recursively moves the rest of the stack

do-nothing(p, q)
  task: move-stack(p, q)
  precond: top($pallet, p$) *; true if p is empty*
  subtasks: ⟨⟩ *; no subtasks, because we are done*

move-each-twice()
  task: move-all-stacks()
  precond: *; no preconditions*
  subtasks: *; move each stack twice:*
    ⟨move-stack(p1a,p1b), move-stack(p1b,p1c),
    move-stack(p2a,p2b), move-stack(p2b,p2c),
    move-stack(p3a,p3b), move-stack(p3b,p3c)⟩





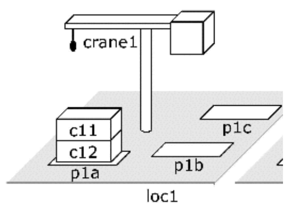
Example: Total-order Formulation

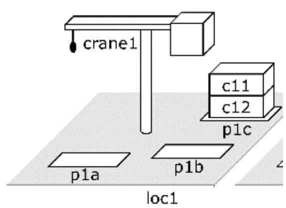
take-and-put($c, k, l_1, l_2, p_1, p_2, x_1, x_2$):
  task: move-topmost-container(p_1, p_2)
  precond: top(c, p_1), on(c, x_1), *; true if p_1 is not empty*
    attached(p_1, l_1), belong(k, l_1), *; bind l_1 and k*
    attached(p_2, l_2), top(x_2, p_2) *; bind l_2 and x_2*
  subtasks: ⟨take(k, l_1, c, x_1, p_1), put(k, l_2, c, x_2, p_2)⟩

recursive-move(p, q, c, x):
  task: move-stack(p, q)
  precond: top(c, p), on(c, x) *; true if p is not empty*
  subtasks: ⟨move-topmost-container(p, q), move-stack(p, q)⟩
    ;; the second subtask recursively moves the rest of the stack

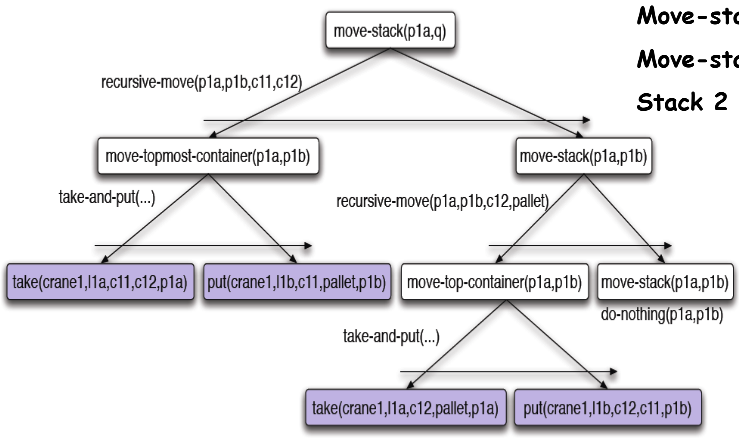
do-nothing(p, q)
  task: move-stack(p, q)
  precond: top($pallet, p$) *; true if p is empty*
  subtasks: ⟨⟩ *; no subtasks, because we are done*

move-each-twice()
  task: move-all-stacks()
  precond: *; no preconditions*
  subtasks: *; move each stack twice:*
    ⟨move-stack(p1a,p1b), move-stack(p1b,p1c),
    move-stack(p2a,p2b), move-stack(p2b,p2c),
    move-stack(p3a,p3b), move-stack(p3b,p3c)⟩





Example: Total-order Formulation



Goal: move-each-twice

Move-stack(p1a, p1b)

Move-stack(p1b, p1c)

Stack 2 and 3 empty.

Solution to TOSTN planning problems

Total-order Forward Decomposition

TFD$(s, \langle t_1, \ldots, t_k \rangle, O, M)$
  if $k = 0$ then return $\langle\rangle$ (i.e., the empty plan)
  if t_1 is primitive then
    active $\leftarrow \{(a, \sigma) \mid a$ is a ground instance of an operator in O,
      σ is a substitution such that a is relevant for $\sigma(t_1)$,
      and a is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(a, \sigma) \in$ *active*
    $\pi \leftarrow$ TFD$(\gamma(s, a), \sigma(\langle t_2, \ldots, t_k \rangle), O, M)$
    if π = failure then return failure
    else return $a.\pi$
  else if t_1 is nonprimitive then
    active $\leftarrow \{m \mid m$ is a ground instance of a method in M,
      σ is a substitution such that m is relevant for $\sigma(t_1)$,
      and m is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(m, \sigma) \in$ *active*
    $w \leftarrow$ subtasks$(m).\sigma(\langle t_2, \ldots, t_k \rangle)$
    return TFD(s, w, O, M)

task list T=(t_1 ,t_2,…)
action a

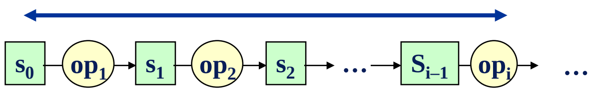
π={a_1 …, a_k, a }; task list T=(t_2, …)

task list T=(t_1 ,t_2,…)
method instance m

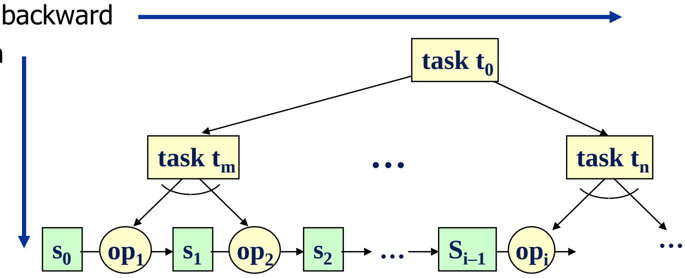
task list T=(u_1,…,u_k ,t_2,…)

Comparison to Forward and Backward Search

- In state-space planning, must choose whether to search forward or backward

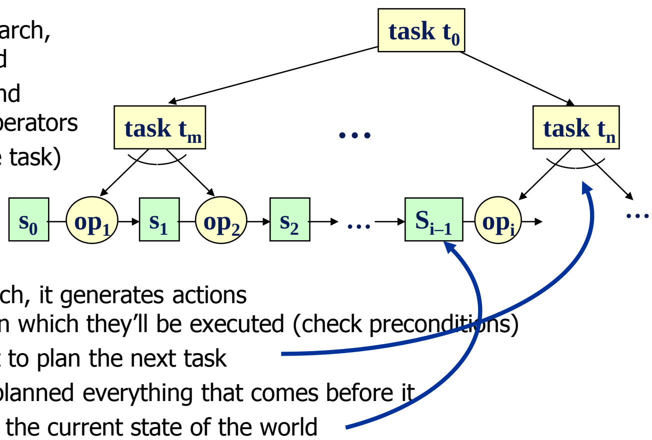


- In HTN planning, there are two choices to make about direction:
  - forward or backward
  - up or down
- TFD goes down and forward



Comparison to Forward and Backward Search

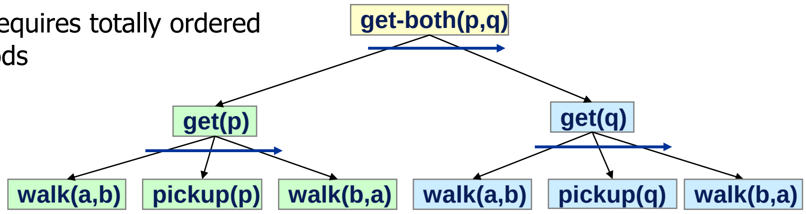
- Like a backward search, TFD is goal-directed
  - Goals correspond to tasks (i.e. operators relevant for the task)



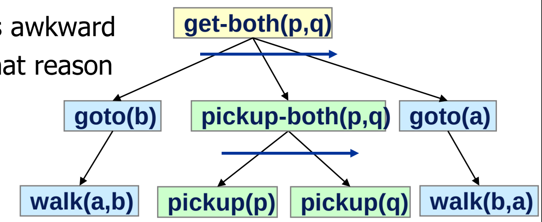
- Like a forward search, it generates actions in the same order in which they'll be executed (check preconditions)
- Whenever we want to plan the next task
  - we've already planned everything that comes before it
  - Thus, we know the current state of the world

Limitation of Ordered-Task Planning

- TFD requires totally ordered methods

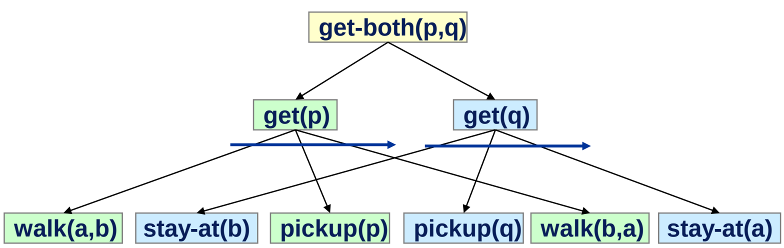


- Can't interleave subtasks of different tasks
- Sometimes this makes things awkward
  - Need to write methods that reason globally instead of locally



Partially Ordered Methods

- With partially ordered methods, the subtasks can be interleaved



- Fits many planning domains better
- Requires a more complicated planning algorithm

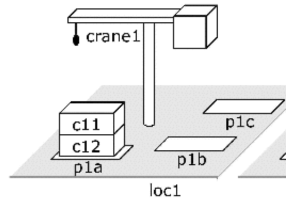
Example: Partial-order Formulation

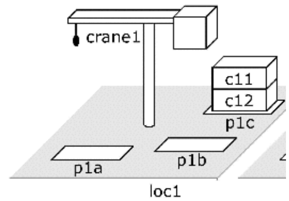
take-and-put$(c, k, l_1, l_2, p_1, p_2, x_1, x_2)$:
  task: move-topmost-container(p_1, p_2)
  precond: top(c, p_1), on(c, x_1), ; *true if p_1 is not empty*
  attached(p_1, l_1), belong(k, l_1), ; *bind l_1 and k*
  attached(p_2, l_2), top(x_2, p_2) ; *bind l_2 and x_2*
  subtasks: ⟨take(k, l_1, c, x_1, p_1), put(k, l_2, c, x_2, p_2)⟩

recursive-move(p, q, c, x):
  task: move-stack(p, q)
  precond: top(c, p), on(c, x) ; *true if p is not empty*
  subtasks: ⟨move-topmost-container(p, q), move-stack(p, q)⟩
  ;; *the second subtask recursively moves the rest of the stack*

do-nothing(p, q)
  task: move-stack(p, q)
  precond: top$(pallet, p)$; *true if p is empty*
  subtasks: ⟨⟩ ; *no subtasks, because we are done*

move-each-twice()
  task: move-all-stacks()
  precond: ; *no preconditions*
  network: ; *move each stack twice:*
  u_1 =move-stack(p1a,p1b), u_2 =move-stack(p1b,p1c),
  u_3 =move-stack(p2a,p2b), u_4 =move-stack(p2b,p2c),
  u_5 =move-stack(p3a,p3b), u_6 =move-stack(p3b,p3c),
  $\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$

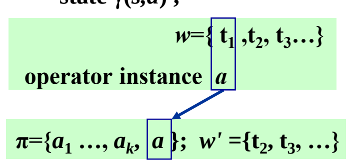


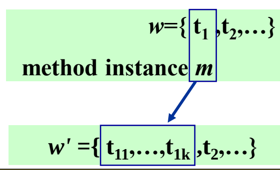


Solution to POSTN planning problems

PFD(s, w, O, M)
  if $w = \emptyset$ then return the empty plan
  nondeterministically choose any $u \in w$ that has no predecessors in w
  if t_u is a primitive task then
    active $\leftarrow \{(a, \sigma) \mid a$ is a ground instance of an operator in O,
      σ is a substitution such that name$(a) = \sigma(t_u)$,
      and a is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(a, \sigma) \in$ *active*
    $\pi \leftarrow$ PFD$(\gamma(s, a), \sigma(w - \{u\}), O, M)$
    if π = failure then return failure
    else return $a.\pi$
  else
    active $\leftarrow \{(m, \sigma) \mid m$ is a ground instance of a method in M,
      σ is a substitution such that task$(m) = \sigma(t_u)$,
      and m is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(m, \sigma) \in$ *active*
    nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
    return(PFD(s, w', O, M))

state $\gamma(s,a)$;





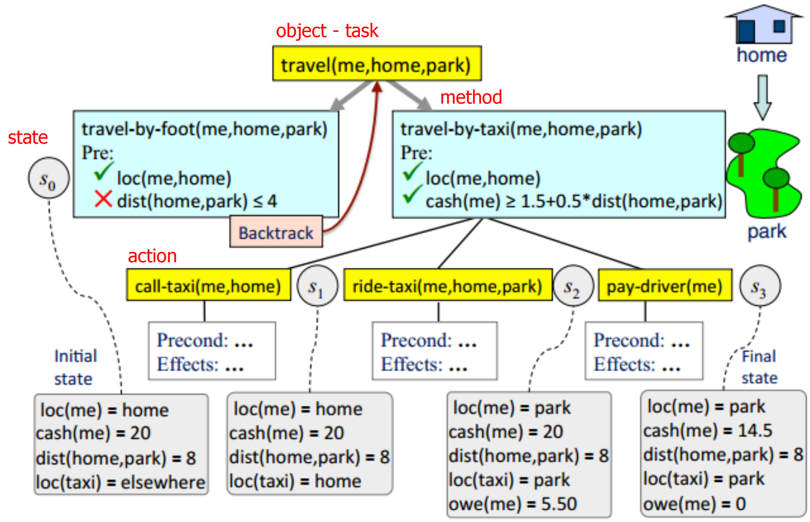
STN planning: TFD & PFD

- STN does not allow parallel execution
- But can interleave steps (PFD)
- The resulting plan is totally ordered (both TFD & PFD)

Comparison to Classical Planning

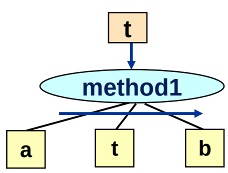
- Like :
  - Each state of the world is represented by a set of atoms.
  - Each action corresponds to a deterministic state transition.
  - Terms, literals, operators, actions, plans have same meaning as classical planning.
- Different:
  - Objective is to perform a set of tasks, not to achieve a set of goals
  - Added tasks, methods, task networks
  - Tasks decompose into subtasks
    - Constraints
    - Task orders
  - Backtrack if necessary

Comparison to Classical Planning



Comparison to Classical Planning

- Some STN planning problems are not expressible in classical planning (STN planning is strictly more expressive than classical planning)
- Example:
  - STN method:
    - ◇ No arguments
    - ◇ No preconditions



  - Two operators, a and b
    - ◇ Again, no arguments and no preconditions
  - Initial state is empty, initial task is t
  - Set of solutions is $\{a^n b^n \mid n > 0\}$
  - No classical planning problem has this set of solutions
    - ◇ The state-transition system is a finite-state automaton
    - ◇ No finite-state automaton can recognize $\{a^n b^n \mid n > 0\}$

Comparison to Classical Planning

- Advantages
  - Express things that can't be expressed in classical planning
  - Specify(encode) standard ways of solving problems (recipte)

  → Otherwise, the planner have to derive recipes repeatedly from 'first principle' every time it solves a problem

  → Can speed up by orders of magnitude (exponential → polynomial)

- Disadvantages
  - Writing/Debugging an HTN domain model can be cumbersome/complicated

    → try HTN if

    (i) it is important to achieve high performance

    (ii) you need more expressive power than classical planners can provide

(General)HTN Planning

- In STN planning, two kinds of constraints are associated with a method
  - Preconditions
  - Ordering constraints(i.e. task network)
- HTN planning can be even more general (generalization of STN)
  - More freedom about how to construct the task networks
  - Can use other decomposition procedures not just forward-decomposition
  - Can have constraints associated with tasks and methods
    - Things that must be true before, during, or afterwards
  - Like POP+STN: input - partial-order tasks, output-partially ordered plan
  - Some algorithms use causal links and threats like those in POP
  - Plan = partially ordered collection of primitive tasks
- Task Network

STN

w = (U, E) - an acyclic graph
U – set of task nodes
E – set of edges

HTN

w = (U, C)
U – set of task nodes
C – set of constraints (allow for generic task networks).

Domain Dependency

- HTN planners may be domain-specific or domain-configurable
- Domain-configurable HTN planner
  - Domain – independent planning algorithm
  - Domain – states, operators, tasks, and methods
  - Planning problem – domain, initial state, initial task network
- Domain dependent vs. Domain independent