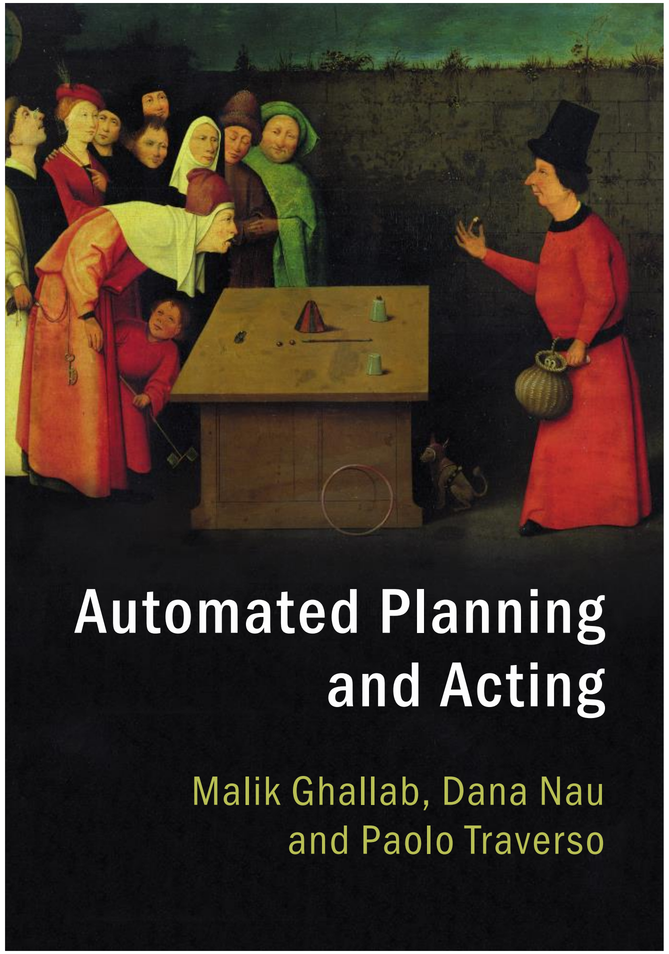
Last update: April 16, 2024

tema 7

Topic 4 Laboratory

Planning with Hierarchical Task Network (HTN)



http://www.laas.fr/planning

Automated Planning
Adrián Domínguez Díaz

Slides adapted from:

Dana S. Nau
University de Maryland

Refinement based planning

- The idea of refining a task into smaller subtasks as in the RAE actuation system (see topic 7 of theory) is also used for planning, not just acting.

- Hierarchical task network (HTN) planning is the process of providing "recipes" for breaking down or refining complex tasks into smaller tasks.
  - Complex tasks are refined into subtasks using methods.
  - Methods refine tasks in sequence of subtasks.
    - Sorting constraints can be included to generate parallelizable plans.
  - Smaller tasks are modeled as actions from a classic planning domain.

- With HTN, planning domains are created with expert knowledge to achieve a solution
  - Whether a solution is reached depends on that knowledge, not just the planner.
  - Planning here is limited to finding the best way to break down the problem.

Hierarchical Task Network (HTN)-based planning

- For many planning problems, we can have ideas of how to find solutions
- Example: Traveling to a faraway destination
  - Brute Force Search
    - Many combinations of vehicles and routes
  - Human experience: a small number of “recipes"

    Ej.: volar:

    1. Buy a ticket from the local to the destination airport
    2. Travel to the local airport
    3. Fly to the destination airport
    4. Travel to the final destination

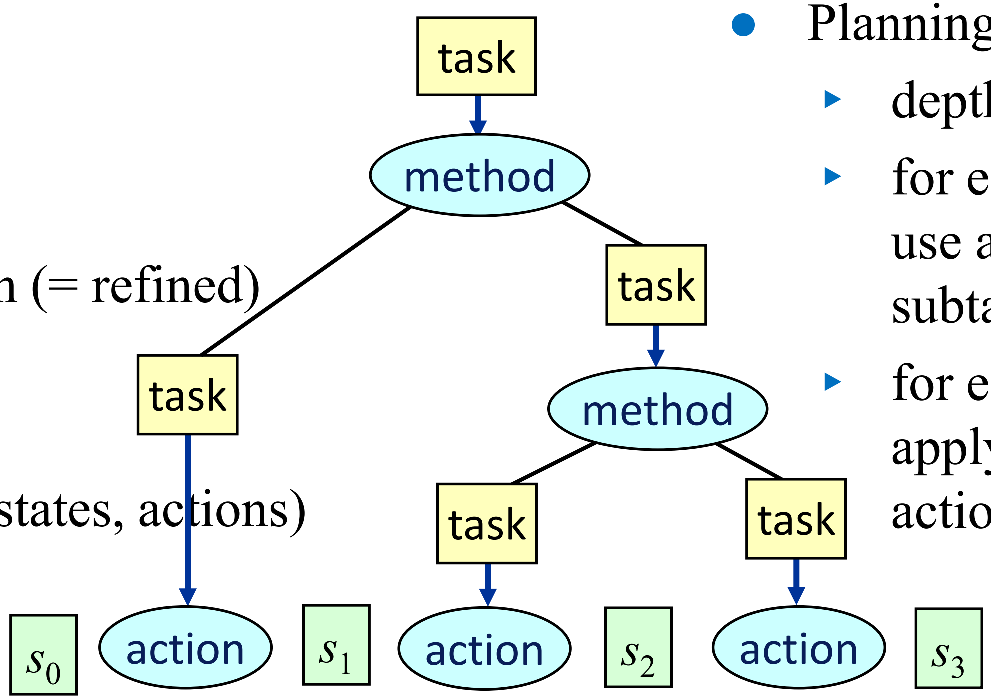
- Through HTN we can incorporate this knowledge into a planner.
  - We will focus on full-order HTN, no order restrictions.
  - Generates fully ordered, non-parallelizable plans.

Total Order HTN Planning

- Ingredients:
  - *states* and *actions*
  - *tasks*: activities to be carried out
  - *HTN methods*: ways to refine the tasks
- Format of a method:

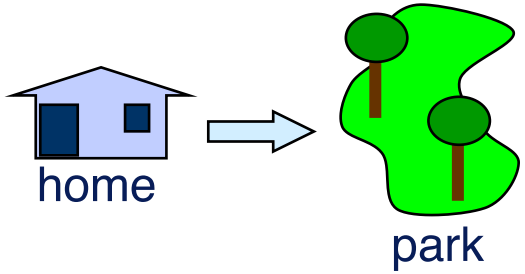
  method-name (*args*)
  Tarea: *task-name*(*args*)
  Prec: *preconditions*
  Sub: *list of subtasks*
- Two types of subtasks
  - *Primitives*: name of an action
  - *Composed*: needs to be broken down (= refined) using mehtods
- HTN planning domain: a pair (Σ, M)
  - Σ: state-variable planning domain (states, actions)
  - M: methods

- Planning problem: $P = (\Sigma, M, s_0, T)$
  - T: a list of tasks $\langle t_1, t_2, \ldots, t_k \rangle$
- Solution: any executable plan that can be generated by applying
  - methods for non-primitive tasks
  - actions for primitive tasks



- Planning algorithm
  - depth-first, from left to right
  - for each composed tasks, use a method to refine it in subtasks
  - for each primitive tasks, apply the corresponding action

Simple travel planning problem



- Action templates:

walk (a,x,y)
  Prec: loc$(a) = x$
  Effect: loc$(a) \leftarrow y$

call-taxi (a,x)
  Prec: —
  Effect: loc(taxi) $\leftarrow x$,
    loc$(a) \leftarrow$ taxi

drive-taxi (a,x,y)
  Prec: loc(a) = taxi,
    loc(taxi) $= x$
  Effect: loc(taxi) $\leftarrow y$,
    debt$(a) \leftarrow 1.50 + ½$ dist(x,y)

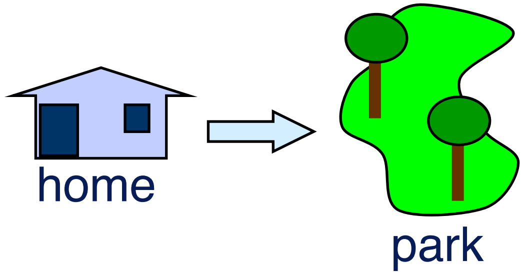
pay-taxi(a,y)
  Prec: debt$(a) \leq$ money(a)
  Effect: money$(a) \leftarrow$ money(a) – debt(a),
    debt$(a) \leftarrow 0$,
    loc$(a) = y$

- Parameters
  - $a \in Agents$
  - $x,y \in Localizations$

Simple travel planning problem

method name
Tarea: generico task

Prec:
Sub: prim / comp.



- *Initial state*:
  - I'm at home, 1
  - I have 20€ 2
  - There's a park at 8 km 3
- s_0 = {loc(me)=home, 1
  money(me)=20, 2
  dist(home,park)=8, 3
  loc(taxi)=anywhere}
- *Task*: travel to the park
  - travel(me,home,park)

- *Methods*:

travel-walking(a,x,y)

Tarea: travel(a,x,y)

Prec: loc(a,x), dist(x, y) ≤ 4

Sub: walk(a,x,y)

travel-by-taxi(a,x,y)

Tarea: travel(a,x,y)

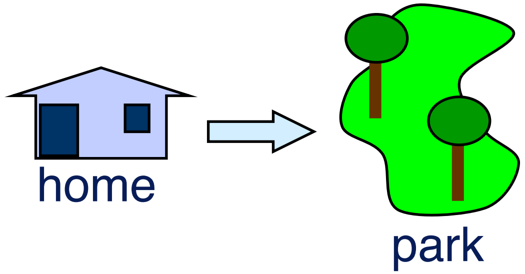
Prec: loc(a,x),
money(a) $\geq 1.50 + ½$ dist(x,y)

Sub: call-taxi (a,x),
travel-taxi (a,x,y),
pay-taxi(a,y)

- Parameters
  - $a \in Agents$
  - $x,y \in Localizations$

Simple travel planning problem

- Backtracking search from left to right:



travel(me,home,park)

travel-walking(me,home,park)
Prec:
✓ loc(me,home)
✗ dist(home,park) ≤ 4

bc it is 8km>4

Backtrack

travel-by-taxi(me,home,park)
Prec:
✓ loc(me,home)
✓ money(me) ≥ 1.5+0.5*dist(home,park)

call-taxi(me,home)
Prec: …
Efec: …

drive-taxi(me,home,park)
Prec: …
Efec: …

pay-taxi(me,park)
Prec: …
Efec: …

s_0
Initial state
loc(me) = home
money(me) = 20
dist(home,park) = 8
loc(taxi) = station

s_1
*loc(me) = taxi**
money(me) = 20
dist(home,park) = 8
***loc(taxi) = home**

(*)effects tou call-taxi action

s_2
loc(me) = taxi
money(me) = 20
dist(home,park) = 8
loc(taxi) = park *
debt(me) = 5.50 *

(*)effects tou action drive-taxi

s_3
Final state
loc(me) = park
money(me) = 14.5
dist(home,park) = 8
loc(taxi) = park
debt(me) = 0

Solution: ⟨call-taxi(me,home), drive-taxi(me,home,park), pay-taxi(me)⟩

Non-determinist planning algorithm

- find-plan(s_0, T)
  - return search-plan(s_0, T, $\langle\rangle$)
- search-plan(s, T, π)
  - if $T = \langle\rangle$ then return π
  - with $t_1, t_2, \ldots, t_k$ tasks of T
  - 1 if t_1 is primitive then
    - if there is an action a where header(a) matches t_1 and a is applicable in s:
      - return search-plan($\gamma(s,a)$, $\langle t_2,\ldots,t_k\rangle$, $\pi.a$)
    - if not: return no-solution
  - 2 if not// t_1 is composed
    - *Candidates* $\leftarrow \{m \in M \mid$ task(m) matches t_1 and m is applicable in $s\}$
    - if *Candidates* $= \emptyset$ then return no-solution
    - no-determinist election of $m \in$ *Candidates*
    - return search-plan(s, subtasks(m)$.\langle t_2,\ldots,t_k\rangle$, π)

E.g. $T = \langle t_1, t_2, \ldots, t_k\rangle$

state s, tasks T=$\langle$ $\boxed{t_1}$, t_2, …, $t_k\rangle$

action $\boxed{a}$

state $\gamma(\boxed{s,a})$, tasks T=$\langle t_2$, …, $t_k\rangle$

state s, tasks T=$\langle$ $\boxed{t_1}$,t_2,…, $t_k\rangle$

method $\boxed{m}$

state s, tasks T=$\langle$ $\boxed{u_1, \ldots, u_j}$, t_2, …, $t_k\rangle$

Integration of hierarchical planning and acting

- run_lazy_lookahead(*state, task_list*)
  - loop:
    - *plan* = find_plan(*state, task_list*)
    - if *plan* = [ ]:
      - return *state* // the new current state
    - for each *action* in the *plan*:
      - run the corresponding command
      - if the command fails:
        - break inner loop

- Travel problem:
  - run_lazy_lookahead calls:
    - find-plan(s_0, [(viajar,me,home,park)])
  - find-plan returns
    - [(call_taxi,me,home),
      (drive_taxi,me,home,park),
      (pay_taxi,me)]
  - run_lazy_lookahead runs
    - c_call_taxi(me,home)
    - c_drive_taxi(me,home,park)
    - c_pay_taxi(me)
- If everything Works ok, I'll get to the park
  - But if the taxi breaks down…

Integration of hierarchical planning and acting

- For planning and acting, it is necessary that HTN methods can recover from unexpected problems
- Example:
  - run_lazy_lookahead runs
    - c_call_taxi(me,home)
    - c_drive_taxi(me,home,park)
      - We assume that the 2nd command fails
  - run_lazy_lookahead calls
    - find_plan(s_1, [(travel,me,home,park)])
    - Error: it tries to use an undefined value
- It's necessary to consider all the possible situations and create methods to manage each of them properly.

s_0

loc(me) = home
money(me) = 20
debt(me) = 0
dist(home,park) = 8
loc(taxi) = station

c_call_taxi(me,home)

s_1

loc(me) = taxi
money(me) = 20
debt(me) = 0
dist(home,park) = 8
loc(taxi) = home

c_drive_taxi(me,home,park)

The command fails, the current state is still s_1

(travel,me,home,park)

(travel_by_foot, me, home, park)
Prec:
✓ loc(me,taxi)
✗ dist(taxi,park) undefined

Error

Example: KILLZONE 2

- "First-person shooter" game, ≈ 2009
- HTN planner designed to plan squad level combat tactics
  - Actions and methods sintax very similar to SHOP and SHOP2
  - It quickly generates linear plans that work if nothing interferes
- **How it works:**
  - Various methods were programmed to decompose behavior in combat
  - For each soldier, a workable tactical plan is generated based on the current state of the game
  - Replans multiple times per second to adapt to dynamic environment
  - You just need to come up with a plan that seems reasonable in the eyes of the player

SHOP Planners

- SHOP Planner Family (Simple Hierarchical Ordered Planner)
  - SHOP → 1999, written in LISP, total order HTN
  - SHOP2 → 2002, written in LISP, partial order HTN
  - JSHOP2 → 2003, Java version of SHOP2 with multiple limitations
  - SHOP3 → 2019, written in LISP, improves SHOP2
  - Pyhop → 2013, simplified Python version of SHOP
  - GTPyhop → 2021, improved version of Pyhop that adds goal-based planning

- SHOP planners are some of the most successful at the industry-level
  - Allow you to create highly efficient domain-specific planners using HTN
  - The SHOP3 version has been developed by SIFT research laboratories

- The Pyhop version was developed to facilitate the use of planning in traditional languages
  - It has important limitations compared to SHOP, but it can be used from Python

Actions (SHOP Operators)

walk(a, x, y)
  Prec: loc(a) = x
  Efec: loc(a) = y

call-taxi(a, x)
  Prec: —
  Efec: loc(taxi) = x

drive-taxi(a, x, y)
  Prec: loc(a) = x, loc(taxi) = x
  Efec: loc(taxi) = y,
    loc(a) = y,
    debt(a) = 1.50 + ½
   dist(x,and)

pay-taxi(a)
  Prec: debt(a) = d, money(a) ≥ d
  Efec: debt(a) = 0,
    money(a) = money(a) – d

$a \in Agents; x, and \in Localizations$

```
(defdomain simple-travel (

(:operator (!walk ?a ?x ?y)
    ((AGENT ?a) (LOCATION ?x) (LOCATION ?y) (loc ?a ?x)) ;prec
    ((loc ?a ?x))  ;delete
    ((loc ?a ?y))) ;add

(:operator (!call-taxi ?a ?x)
    ((AGENT ?a)(LOCATION ?x)(loc taxi ?y)) ;prec
    ((loc taxi ?y))  ;delete
    ((loc taxi ?x))) ;add

(:operator (!drive-taxi ?a ?x ?y)
    (;prec
        (AGENT ?a) (LOCATION ?x) (LOCATION ?y)
        (loc taxi ?x) (loc ?a ?x)
        (debt ?a ?o) (dist ?x ?y ?d)
    )(;delete
        (loc taxi ?x) (loc ?a ?x)
        (debt ?a ?o)
    )(;add
        (loc taxi ?y) (loc ?a ?y)
        (debt ?a (call + 1.5 (call * 0.5 ?d))
    )

(:operator (!pay-taxi ?a)
    ((AGENT ?a) (money ?a ?t) (debt ?a ?d) (call > ?t ?d))
    ((money ?a ?t) (debt ?a ?d)) ;delete
    ((money ?a (call - ?t ?d)) (debt ?a 0)) ;add
;...
```

Methods

travel-walking(a, x, and)
    Task: travel(a, x, and)
    Prec: loc(a) $= x$, dist(x, and) ≤ 4
    Sub: andar(a, x, and)

travel-by-taxi(a, x, and)
    Task: travel(a, x, and)
    Prec: money(a) $\geq 1.5 + 0.5*$dist(x, and)
    Sub: call-taxi (a, x),
        drive-taxi (a, x, and),
        pay-taxi(a)

```
(method (travel ?a ?x ?and)

    (;prec travel-walking
        (dist ?x ?and ?d) (call <= ?d 4)
    )
    (;subtasks travel-walking
        (!andar ?a ?x ?and)
    )
    (;prec travel-by-taxi
        (dist ?x ?and ?d) (money ?a ?t)
        (call > ?t (call + 1.5 (call * 0.5 ?d)))
    )
    (;subtasks travel-by-taxi
        (!call-taxi ?a ?x)
        (!drive-taxi ?a ?x ?and)
        (!pay-taxi ?a ?x ?and)
    )
)

);final defdomain
```

Simple travel planning problem

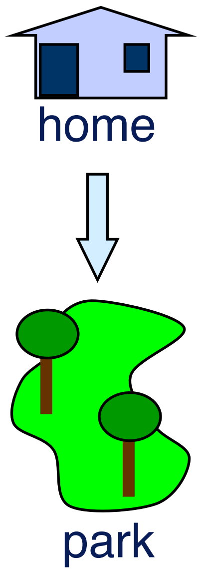
Initial state:

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

Task:

travel(me,home,park)

```
(defproblem problem simple-travel
    (;Initial state
        (AGENT me) (LOCATION home) (LOCATION park)
        (loc me home) (money me 20) (dist home park 8)
    )
    (;Task to solve
        (travel me home park)
    )
)
```



Solution plan:

call-taxi(me,home), drive-taxi(me,park), pay-taxi(me)

```
[(call-taxi,me,home), (drive-taxi me home park), (pay-taxi me)]
```

SHOP advances aspects

- Parameters that are not in the operator or method header can be used
  - Those parameters should appear in prconditions so SHOP (its underlying LISP interpreter) can unify (assign) them with appropiate values.
    - If the unification fails, SHOP backtracks and tries with different values for the parameters.
    - It is possible to give hints about the order in which possible values for parameter unification should be tried.

- A task can be refined recursively when it must be executed as a loop until a stop condition is met
  - The method includes the task being refined as one of the subtasks.
  - A stop condition must be included, which can return an emtpy list of subtasks.

- SHOP2 supports partially ordered subtasks refinements, which include order constraints between tasks to be able to generate parallelizable solutions.