chap 2a

Domain Model

- classical Planning Domain

$\Sigma = (S, A, \gamma, cost)$

S → set of states
A → set of actions

- $\gamma: S \times A \rightarrow S$ (Prediction/State transition function)
- plan → sequense of actions $\pi = \langle a_1 \dots a_n \rangle$

Classical Planning Problem

$P = (\Sigma, s_0, S_g)$

Σ → planning domain
s_0 → initial state
S_g → set of goals

→ solution of P:

a plan π such that $\gamma(s_0, \pi) \in S_g$

1. State-Variable Representation

- Action: (head, preconditions, effects, cost)
  - *head*: name and parameter list
    - Get actions by instantiating the parameters
  - *preconditions*:
    - Computational tests to predict whether an action can be performed
    - Should be necessary/sufficient for the action to run without error
  - *effects*:
    - Procedures that modify the current state
  - *cost*: procedure that returns a number
    - Can be omitted, default is cost ≡ 1

Objective Properties

rigid

- stays the same in every state

varying

(fluent property)

- property that may differ in different state

↳ represented by state variables

Domain Model

State-transition system or *classical planning domain*:

- $\Sigma = (S,A,\gamma,\text{cost})$ or (S,A,γ)
  - S - finite set of *states*
  - A - finite set of *actions*
  - $\gamma: S\times A \to S$

    prediction (or *state-transition*) function
    - *partial* function: $\gamma(s,a)$ is not necessarily defined for every (s,a)
      - a is *applicable* in s iff $\gamma(s,a)$ is defined
      - $\text{Domain}(a) = \{s \in S \mid a \text{ is applicable in } s\}$
      - $\text{Range}(a) = \{\gamma(s,a) \mid s \in \text{Domain}(a)\}$
  - cost: $S\times A \to \mathbb{R}^+$ or cost: $A \to \mathbb{R}^+$
    - optional; default is $\text{cost}(a) \equiv 1$
    - money, time, something else

- *plan*:
  - a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$
- π is *applicable* in s_0 if the actions are applicable in the order given

  $\gamma(s_0, a_1) = s_1$

  $\gamma(s_1, a_2) = s_2$

  $\ldots$

  $\gamma(s_{n-1}, a_n) = s_n$
  - In this case define $\gamma(s_0, \pi) = s_n$

- *Classical planning problem*:
  - $P = (\Sigma, s_0, S_g)$
  - planning domain, initial state, set of goal states
- *Solution* for P:
  - a plan π such that that $\gamma(s_0,\pi) \in S_g$

$\Sigma = (S, A, \gamma, \text{cost})$

$\downarrow$

$\gamma : S \times A \to S$

prediction (state-transition) function

partial function : $\gamma(S,a)$ -> not necessarily defined for every (S,a)

- a applicable in S iff $\gamma(S,a)$ defined

$\text{Domain}(a) = \{s \in S \mid a \text{ is applicable in } s\}$

$\text{Range}(a) = \{\gamma(s,a) \mid s \in \text{Domain}(a)\}$

$\Sigma = (S, A, \gamma)$

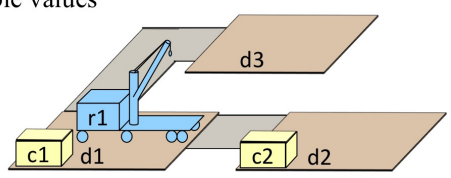
$P = (\Sigma, s_0, S_g)$

States as Functions

- represent states ως μια function που δινει assign values σε state variables

- For each state variable x, $s(x)$ is one x's possible values

s_1(loc(r1)) = d1, s_1(cargo(r1)) = nil,
s_1(loc(c1)) = d1, s_1(loc(c2)) = d2



- Mathematically, a function is a set of ordered pairs

  s_1 = {(loc(r1), d1), (cargo(r1), nil), (loc(c1), d1) , (loc(c2), d2)}

- Equivalently, write it as a set of *ground positive literals* (or *ground atoms*):

  s_1 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1, loc(c2)=d2}

  - Here, we're using '=' as a predicate symbol

Action Templates

a = (head, pre, eff, cost)
- head: name/parameters
- pre: preconditions
- eff: effect literals
- cost: a number

=> example:

- move(r,l,m) → head (name/parameters)
  - pre: loc(r) = l , adjacent(l,m)
    - loc(r) = l : robot r is at location l
    - adjacent(l,m) : location l is adjacent to location m
  - eff: loc(r) ← m
    - loc(r) ← m : robot r is now at location m

=> move function says: if 2 locations m,l are adjacent => move robot from location l to location m

- take(r,l,c)
  - pre: cargo(r) = nil, loc(r) = l, loc(c) = l
    - cargo is empty, location of cargo & robot is the same
  - eff: cargo(r) ← c , loc(c) ← r

So: A: set of action templates

move(r,l,m)
pre: loc(r)=l, adjacent(l, m)
eff: loc(r) ← m

take(r,l,c)
pre: cargo(r)=nil, loc(r)=l, loc(c)=l
eff: cargo(r) ← c, loc(c) ← r

put(r,l,c)
pre: loc(r)=l, loc(c)=r
eff: cargo(r) ← nil, loc(c) ← l

$r \in Robots = \{r1\}$
$l,m \in Locs = \{d1,d2,d3\}$
$c \in Containers = \{c1,c2\}$

Actions

→ replace each parameter with sth in its range

A= {all actions we can get from A}

= {all ground instances of members of A}

Applicability

a is applicable in s if:

- for every positive literal $l \in$ pre(a), $l \in s$ or l is in one of the rigid relations
- for every negative literal $\neg l \in$ pre(a), $l \notin s$ and l isn't in any of the rigid relations

Summary: δίνονται κάποια rigid relations → σημαίνουν adjacent = ...,
κάποια states πχ s_1 = {loc ..., cargo, ...}
[Γενικά σε γίνεται ποιος είναι που ...]
είναι applicable κάποιο function or τα predicates
βγαίνουν με τα states και τα rigid relations

State-transition Function

αν το action είναι apply ⊛ στο S state

=> γ(a,s) δίνεται ως εξης:

Προσθέτει το effect
(απλαδη τα αποτελεσματα
του action πχ. move)

note (γ)

In automated planning, the function γ (gamma) typically refers to a transition function. It maps a state and an action to a new state, indicating the outcome of applying that action in the given state. This function is crucial for understanding how actions affect the environment, enabling the planning system to predict the results of actions and make informed decisions to achieve its goals.

slide 16 eg

State-Variable Planning Domain

- Let
  - B = finite set of objects
  - R = finite set of rigid relations over B
  - X = finite set of state variables
    - for every state variable x, Range(x) ⊆ B
  - S = state space over X
    - = {all value-assignment functions that have sensible interpretations}
  - $\mathcal{A}$ = finite set of action templates
    - for every parameter y, Range(y) ⊆ B
  - A = {all ground instances of action templates in $\mathcal{A}$}
  - $\gamma(s,a)$ = {(x,w) | eff(a) contains the effect $x \leftarrow w$}
    ∪{$(x,w)\in s$ | x isn't the target of any effect in eff(a)}
- Then $\Sigma = (S,A,\gamma)$ is a *state-variable planning domain*

?

.mean that:

στο οικο μας intended interpretation (ερμηνεια)
π.χ ενα robot και βαλα 1 η 2 boxes?

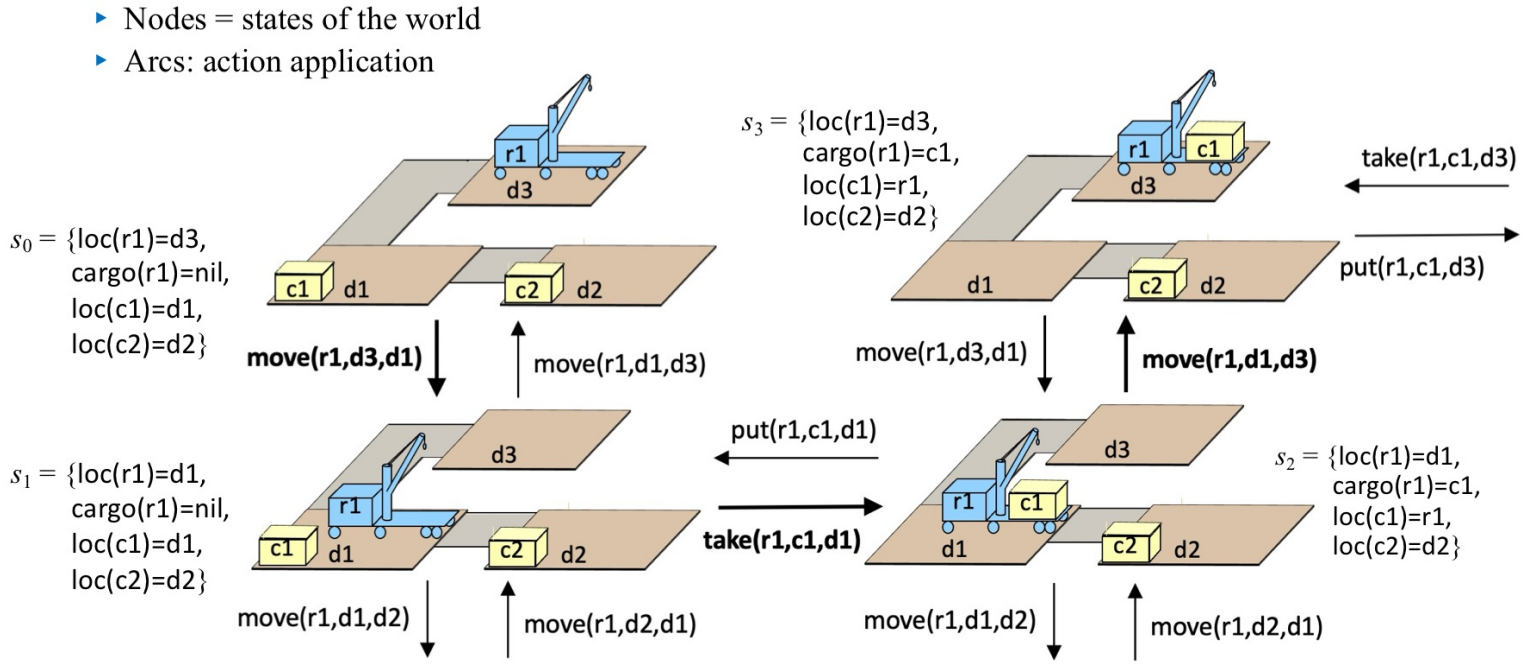
πως κανω enforce την ερμηνεια μου?
δηλαδη πως κανω make sure οτι ολα δουλευουν
οπως εγω θελω?

- explicitly: mathematical axioms / integrity constraints

- implicitly:
  - βραφω initial state So that satisfies env εμπνεια
  - βραφω τα functions με εφοπο οπου οταν το S τους satisfy την εμπνεια => τοτε και το γ(S,a) θα την τους satisfy (το transition function) => δηλαδη ολα ...

State Space

- *State Space*: a directed graph
  - Nodes = states of the world
  - Arcs: action application



Applying a plan

→ Plan π είναι applicable in a state S if we can apply the actions in the order that they appear in π

=> produces a path στο state space (*)

Planning Problems

State-Variable Planning Prob.

P = (Σ, So, g)
- Σ → (S, A, γ) — St-var planning domain
- So → initial state
- g → set of ground literals (goals) → S_g = {states in S that satisfy g}

if γ(So, π) satisfies g => π is a solution for P

*remember π is a sequence of actions $\pi = \langle a_1 \dots a_n \rangle$

2. Classical Representation

- Classical representation is equivalent to state-variable representation
  - No distinction between rigid and varying properties
  - Both represented as logical predicates
  - Both are in the current state

adjacent(l,m) - location l is adjacent to m

loc(r) = l ⟶ loc(r,l) - robot r is at location l

loc(c) = r ⟶ loc(c,r) - container c is on robot r

cargo(r) = c ⟶ loaded(r) - there's a container on r

why not loaded(r,c)?

- State s = a set of ground atoms
  - Atom a is true in s iff $a \in s$

s_0 = {adjacent(d1,d2), adjacent(d2,d1)
adjacent(d1,d3), adjacent(d3,d1)
loc(c1,d1), loc(r1,d2)}

Poll: Should s_0 also contain ¬ loaded(r1)?

A: yes B: no

C: unsure

Classical Planning Operators

Difference with previous

- Action templates

move(r,l,m)
pre: loc(r)=l, adjacent(l, m)
eff: loc(r) ← m *

take(r,l,c)
pre: cargo(r)=nil, loc(r)=l, loc(c)=l
eff: cargo(r) ← c, loc(c) ← r

put(r,l,c)
pre: loc(r)=l, loc(c)=r
eff: cargo(r) ← nil, loc(c) ← l

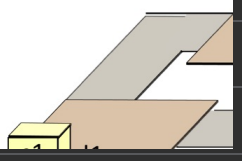
Range(r) = *Robots* = {r1}
Range(l) = Range(m) = *Locs* = {d1,d2,d3}
Range(c) = *Containers* = {c1,c2}

- Classical planning operators

move(r,l,m)
pre: loc(r,l), adjacent(l, m)
eff: ¬loc(r,l), loc(r,m)

take(r,l,c)
pre: ¬loaded(r), loc(r,l), loc(c,l)
eff: loaded(r), ¬loc(c,l), loc(c,r)

put(r,l,c)
pre: loc(r,l), loc(c,r)
eff: ¬loaded(r), loc(c,l), ¬loc(c,r)



* you make the previous predicate false with '¬' and make true the effect!

Actions

Actions

explained previously

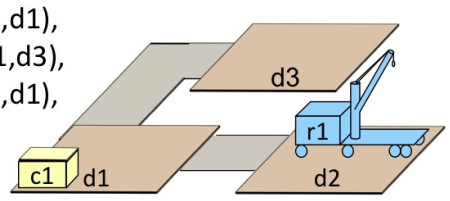
- Planning operator:

o: move(r,l,m)
pre: loc(r,l), adjacent(l,m)
eff: ¬loc(r,l), loc(r,m)

- Action:

a_1: move(r1,d2,d1)
pre: loc(r1,d2), adjacent(d2,d1)
eff: ¬loc(r1,d2), loc(r1,d1)

s_0 = {adjacent(d1,d2),
adjacent(d2,d1),
adjacent(d1,d3),
adjacent(d3,d1),
loc(c1,d1),
loc(r1,d2)}



- Let
  - pre⁻(a) = {a's negated preconditions}
  - pre⁺(a) = {a's non-negated preconditions}
- a is applicable in state s iff

$s \cap \text{pre}^-(a) = \emptyset$ and $\text{pre}^+(a) \subseteq s$

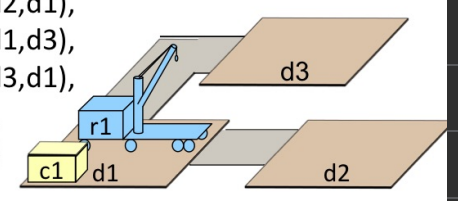
if state does not include the negated preconditions of action a, but should include the non-negated ones

- If a is applicable in s then
  - $\gamma(s,a) = (s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)$

means remove from State transition the negated effects & add the non-negated ones

meaning?

$\gamma(s_0, a_1)$ = {adjacent(d1,d2),
adjacent(d2,d1),
adjacent(d1,d3),
adjacent(d3,d1),
loc(c1,d1),
loc(r1,d1)}



PDDL

```
(:action   move
 :parameters   (?r ?l ?m)
 :precondition (and (loc ?r ?l)
                    (adjacent ?l ?m)
 :effect  (and (not (loc ?r ?l))
               (loc ?r ?m)))
```

classical representation

move(r,l,m)
Precond: loc(r,l), adjacent(l,m)
Effects: ¬loc(r,l), loc(r,m)

```
(define (problem example-problem-1)
    (:domain example-domain-1))

    (:init
      (adjacent d1 d2)
      (adjacent d2 d1)
      (adjacent d1 d3)
      (adjacent d3 d1)
      (loc c1 d1)
      (loc r1 d2)

    (:goal (loc c1 r1)))
```

→ So is the initial State (init in PDDL)

+sets of objects

```
(define (problem example-problem-2)
    (:domain example-domain-2))

  (:objects
      r1 - robot
      c1 - container
      d1 d2 d3 - location)

    (:init
      (adjacent d1 d2)
      (adjacent d2 d1)
      (adjacent d1 d3)
      (adjacent d3 d1)
      (loc c1 d1)
      (loc r1 d2)

    (:goal (loc c1 r1)))
```

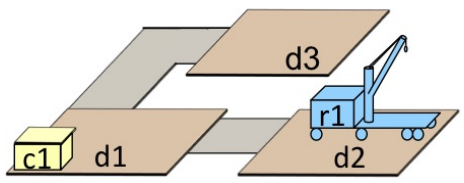
→ goal

here is the domain: προσδιορίζει τύπους & predicates καθε function

Typed domain

State-variable planning:

- Sets of objects
  - *B* = *Movable_objects* ∪ *Locs*
  - *Movable_objects* = *Robots* ∪ *Containers*
  - *Robots* = {r1}
  - *Containers* = {c1}
  - *Locs* = {d1, d2, d3}



- Parameter ranges
  - $r \in Robots$
  - $l,m \in Locs$
  - $c \in Containers$

```
define (domain example-domain-2)
   (:requirements
       :negative-preconditions
       :typing)
   (:types
       location movable-obj - object
       robot container - movable-obj)

   (:predicates
       (loc ?r - movable-obj
            ?l - location)
       (loaded ?r - robot)
       (adjacent ?l ?m - location))
```

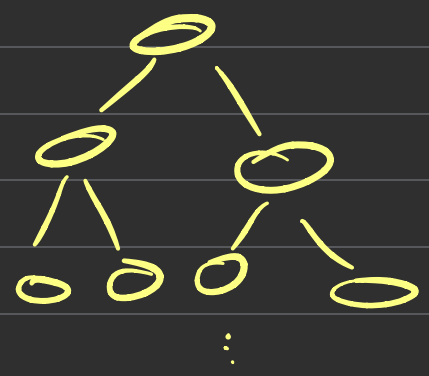
like saying *Locations, Movable_objects* ⊆ *B*
Robots, Containers ⊆ *Movable_objects*

```
(:action move
 :parameters (?r - robot
              ?l ?m - location)
 :precondition (and (loc ?r ?l)
                    (adjacent ?l ?m))
 :effect (and (not (loc ?r ?l))
              (loc ?r ?m)))

(:action take
 :parameters (?r - robot
              ?l - location
              ?c - container)
 :precondition (and (loc ?r ?l)
                    (loc ?c ?l)
                    (not (loaded ?r)))
 :effect (and (not (loc ?r ?l))
              (loc ?r ?m)))

(:action put
 :parameters (?r - robot
              ?l - location
              ?c - container)
 :precondition (and (loc ?r ?l)
                    (loc ?c ?r))
 :effect (and (loc ?c ?l)
              (not (loc ?c ?r))
              (not (loaded ?r)))))
```

like saying $r \in Robots$, $l \in Locs$, $c \in Containers$

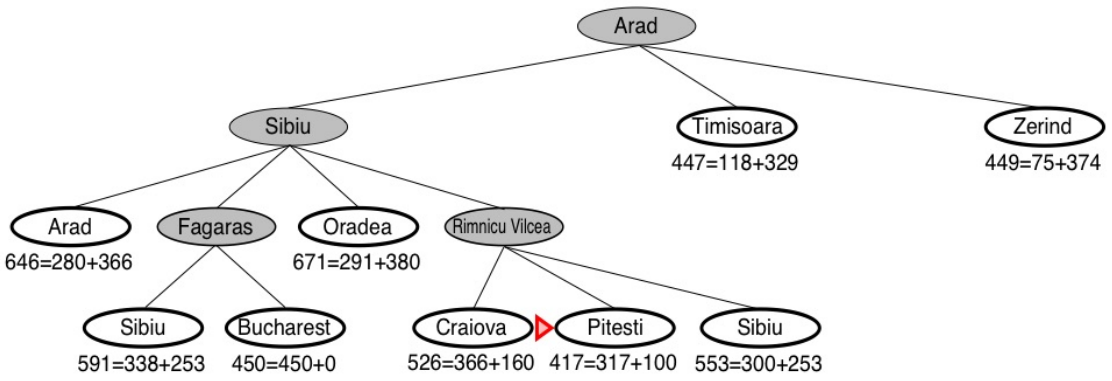
Planning as Search

- Most AI planning procedures are search procedures
  - *Search tree*: the data structure the procedure uses to keep track of which paths it has explored



Search Tree Terminology

Search-Tree Terminology



- *Node* ≈ a pair $\nu = (\pi, s)$, where $s = \gamma(s_0, \pi)$
  - In practice, ν will contain other things too
    - depth(ν), cost(π), pointers to parent and children, …
  - π isn't always stored explicitly, can be computed from the parent pointers
- *children* of $\nu = \{(\pi.a, \gamma(s,a)) \mid a \text{ is applicable in } s\}$
- *successors* or *descendants* of ν:
  children, children of children, etc.
- *ancestors* of ν
  $= \{\text{nodes that have } \nu \text{ as a successor}\}$
- *initial* or *starting* or *root* node $\nu_0 = (\langle\rangle, s_0)$
  - root of the search tree
- *path* in the search space: sequence of nodes $\langle \nu_0, \nu_1, \ldots, \nu_n \rangle$ such that each ν_i is a child of ν_{i-1}
- *height* of search space
  = length of longest acyclic path from ν_0
- *depth* of ν
  = length(π) = length of path from ν_0 to ν
- *branching factor* of ν
  = number of children of ν
- *branching factor* of a search tree
  = max branching factor of the nodes
- *expand* ν: generate all children

Forward Search

Non-deterministic algorithm:
· βρίσκω ένα π και το επιστρέφω, this is the solution! (SOUND)
· αν το πρόβλημα είναι solvable → κουμ. I από τα πιθανα execution traces θα επιστρέψουν αυτόν (COMPLETE)

→ Forward Search: determinist algorithm
→ all sound!
→ αν είναι complete depends στο πως θα κανεις implement το non-deterministic choice!
όπ.: ποιο leaf node to expand?
⋮

DFS / BFS (36,37 page)

Uniform Cost Search 39/65

· depth (του node v): αριθμος edge (οπαδοη path) απο την ριζα (initial node) εως το συγκεκριμενο node v.

· path: ακολουθια κομβων V1...Vn οπου καθε Vi ειναι father του Vi+1

· height (κομβου v / node v): μεγαλυτερο μονοπατι απο node v προς τα παιδια του

· expand a node = generate all of its children

· max branch factor → max num of children απο ολα τα nodes

```
Deterministic-Search(Σ, s0, g)
    Frontier ← {(⟨⟩, s0)}
    Expanded ← ∅
    while Frontier ≠ ∅ do
        select a node v = (π, s) ∈ Frontier      (i)
        remove v from Frontier
        add v to Expanded
        if s satisfies g then return π
        Children ←
            {(π.a, γ(s,a)) | s satisfies pre(a)}
        prune 0 or more nodes from
            Children, Frontier, Expanded         (ii)
        Frontier ← Frontier ∪ Children
    return failure
```

special cases: DFS, BFS, A*

πως τα ξεχωριζω

1. how they select nodes (i)

2. how they prune nodes (ii)

BFS: (i) select node v, (π,s) ∈ Frontier οπου exei smallest length(π) → smallest number of edges!

tie → select oldest

(ii) remove every (π,s) ∈ Children ∪ Frontier εαν ηδη s ∈ Expanded!

→ δια ταυτην την αρνω expand 2 φορες το ιδιο state

- Terminates
- Returns solution if one exists
  - shortest, but not least-cost
- Worst-case complexity:
  - memory $O(|S|)$
  - running time $O(b|S|)$
- where
  - b = max branching factor
  - $|S|$ = number of states in S

$O(b^d)$

$O(b^d)$

DFS:

(i) Select node v (π,s) ∈ Frontier
often (ex) largest length(π)
→ largest number of edges
*tie? smallest h(s) heuristic * later

(ii)

- *Pruning.* First do cycle-checking. Then, to eliminate nodes that the algorithm is done with, remove ν from *Expanded* if it has no children in *Frontier* ∪ *Expanded*, and do the same with each of ν's ancestors until no more nodes are removed. This *garbage-collection* step corresponds to what happens when a recursive version of depth-first search returns from a recursive call.

- Terminates
- Returns solution if there is one
  - No guarantees on quality
- Worst-case running time $O(b^l)$
- Worst-case memory $O(bl)$
  - b = max branching factor
  - l = max depth of any node

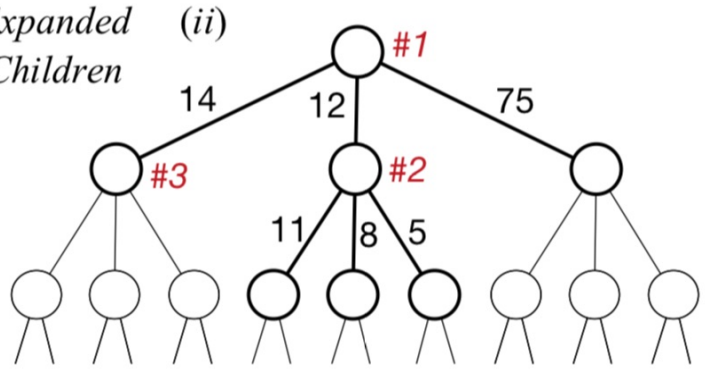
Uniform-Cost Search

Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if s satisfies g then return π
    Children ←
      {($\pi.a$, $\gamma(s,a)$) | s satisfies pre(a)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

(*i*): Select (π,s) ∈ *Frontier* that has smallest cost(π)

(*ii*): Prune every (π,s) ∈ *Children* ∪ *Frontier*
such that *Expanded* already contains a node (π',s)

- Properties
  - Terminates
  - Finds optimal (i.e., least-cost) solution if one exists
  - Worst-case time $O(b|S|)$
  - Worst-case memory $O(|S|)$



Poll: If node ν is expanded before node ν′, then how are cost(ν) and cost(ν′) related?

A. cost(ν) < cost(ν′)
B. cost(ν) ≤ cost(ν′)
C. cost(ν) > cost(ν′)
D. cost(ν) ≥ cost(ν′)
E. none of the above

Heuristic functions

idea: estimate the cost of getting from a state s to a goal

- Let $h^*(s) = \min\{\text{cost}(\pi) \mid \gamma(s,\pi) \in S_g\}$
  - Note that $h^*(s) \geq 0$ for all s

- *heuristic function* $h(s)$:
  - Returns estimate of $h^*(s)$
  - Require $h(s) \geq 0$ for all s

Greedy Best-First Search

idea: choose a node that's likely to be close to a goal
- select node $v = (\pi, s) \in$ Frontier with $h(s)$ is smallest
- pruning: keep path with lowest cost

Properties: quickly, near optimal

A*

idea: try to choose a node on an optimal path from s_0 to goal
- select node $v = (\pi, s)$ that has smallest value of $f(v) = \text{cost}(\pi) + h(s)$

Admissibility

- Notation:
  - $v = (\pi, s)$, where π is the plan for going from s_0 to s
  - $h^*(s) = \min\{\text{cost}(\pi') \mid \gamma(s,\pi') \text{ satisfies } g\}$
  - $f^*(v) = \text{cost}(\pi) + h^*(s)$
  - $f(v) = \text{cost}(\pi) + h(s)$

"admissibility" in automated planning refers to a characteristic of algorithms that help them find the best or "optimal" solution to a problem. Specifically, an algorithm is considered admissible if it is guaranteed to find the most efficient path or sequence of actions to achieve a goal, provided such a path exists. This means that the algorithm won't overlook a better solution in favor of a worse one. It's like being sure that the route your GPS suggests is the shortest or quickest way to get to your destination, assuming it has all the necessary information to make that decision

Definition: h is admissible if for every s, $h(s) \leq h^*(s)$

- *Optimality:*
  - in classical planning problems, if h is admissible then any solution returned by A* will be optimal (least cost)

Dominance

Definition: . let h_1, h_2 be admissible heuristic functions
. h_2 dominates h_1 if $\forall s$, $h_1(s) \leq h_2(s) \leq h^*(s)$

- Suppose h_2 dominates h_1, and A* always resolves ties in favor of the same node. Then
  - A* with h_2 will never expand more nodes than A* with h_1
  - In most cases, A* with h_2 will expand fewer nodes than A* with h_1

Digression

- Straight-line distance to GOAL is a DOMAIN-SPECIFIC heuristic function
  - OK for planning a path to Bucharest
  - Not for other planning problems

  → eg.

- DOMAIN-INDEPENDENT heuristic function:
  → can be used in ANY classical planning domain

Properties of A*

In classical planning problems:

- *Termination:* A* will always terminate
- *Completeness:* if the problem is solvable, A* will return a solution
- *Optimality:* if h is admissible then the solution will be optimal (least cost)
- *Dominance:* If h_2 dominates h_1 then (assuming A* always resolves ties in favor of the same node)
  - A* with h_2 will never expand more nodes than A* with h_1
  - In most cases, A* with h_2 will expand fewer nodes than A* with h_1
- A* needs to store every node it visits
  - Running time $O(b|S|)$ and memory $O(|S|)$ in worst case
  - With good heuristic function, usually much smaller
- The book discusses additional properties

Comparison

- If h is admissible, A* will return optimal solutions
  - But running time and memory requirement grow exponentially in b and d ☹

- GBFS returns the first solution it finds
  - There are cases where GBFS takes more time and memory than A*
    - But with a good heuristic function, such cases are rare
  - On classical planning problems with a good heuristic function
    - GBFS usually near-optimal solutions
    - GBFS does very little backtracking
    - Running time and memory requirement usually much less than A*

GBFS is mostly used!

Iterative Deepening (IDS)

Comparisons

- If h is admissible, both A* and DFBB will return optimal solutions
  - Usually DFBB generates more nodes, but A* takes more memory
  - DFBB does badly in highly connected graphs (many paths to each state)
    - Can have exponentially worse running time than A* (generates nodes exponentially many times)
  - DFBB best in problems where S is a tree of uniform height, all solutions at the bottom (e.g., constraint satisfaction)
    - DFBB and A* have similar running time
    - A* can take exponentially more memory than DFBB
- DFS returns the first solution it finds
  - can take much less time than DFBB
  - but solution can be very far from optimal

Properties:

- Termination, completeness, optimality
  - same as BFS
- Memory (worst case): $O(bd)$
  - vs. $O(b^d)$ for BFS
- If the number of nodes grows exponentially with d:
  - worst-case running time $O(b^d)$, vs. $O(b^l)$ for DFS
  - b = max branching factor
  - l = max depth of any node
  - d = min solution depth if there is one, otherwise l

Incorporating Planning into an Actor

In classical planning, we assumed...

- Finite, static world, just one actor
- No concurrent actions, no explicit time
- Determinism, no uncertainty

▸ Sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$

Most real-world environments don't satisfy the assumptions
=> Errors in prediction

- OK if
  - errors occur infrequently, and
  - they don't have severe consequences
- What to do if an error *does* occur?

Using Planning in Acting pages 59-65 Chap 2a

What can go wrong?

- Execution failures
- unexpected events
- incorrect information
- missing information

How to detect & recover from errors?

Sajes: call Lookahead → av π = failure ...
replans immediately!
→looks a limited distance ahead

Problem? · needs to return quickly
· otherwise may pause repeatedly while waiting for Lookahead to return
· if state changes during the wait?

call Lazy-Lookahead 60/65

call Concurrent-Lookahead 61/65

how to do a Lookahead? Full planning, Receding Horizon, Sampling, Subgoaling
↓
Monte-Carlo rollout

chap 2c

Backward Search

forward search vs backward search

- Forward search: forward from initial state
  - In state s, choose applicable action a
  - Compute state transition $s' = \gamma(s,a)$
- Backward search: backward from the goal
  - For goal g, choose *relevant* action a
    - A possible "last action" before the goal
    - Sometimes this has a lower branching factor

- Compute *inverse* state transition $g' = \gamma^{-1}(g,a)$
  - g' = properties a state s' should satisfy in order for $\gamma(s',a)$ to satisfy g
- Equivalently, if S_g = {all states that satisfy g} then
  - $S_{g'}$ = {all states s such that $\gamma(s,a) \in S_g$}

Relevance

idea: when can a be useful as the last action of a plan to achieve g?

✓ a κανει true ενα atom (π.χ. loc(c1)=d1) το οποιο υπαρχει στο g και προηγουμενως ηταν false

✓ a δεν κανει false τοποτα part του g

- a is *relevant* for $g = \{x_1=c_1, x_2=c_2, \ldots, x_k=c_k\}$ if
  - at least one atom in g is also in eff(a)
    - e.g., if eff(a) contains $x_1 \leftarrow c_1$
  - eff(a) doesn't make any atom in g false
    - e.g., eff(a) must not contain $x_2 \leftarrow c_2'$ (where $c_2' \neq c_2$)
  - whenever pre(a) requires an atom of g to be false, eff(a) makes the atom true
    - e.g., if pre(a) contains $x_3 = c_3'$ (where $c_3' \neq c_3$), then eff(a) must contain $x_3 \leftarrow c_3$

6/36 chap2c:

$g = \{loc(c1)=r1\}$ $\gamma^{-1}(g, load(r1,c1,d3)) =$

$\{cargo(r1)=nil, loc(r1)=d3, loc(c1)=r1\}$

το κανω remove (δεν ειναι με το goal) $loc(c1)=d3$

$\gamma^{-1}(g, load(r2,c1,d1))$ = undefined (not relevant to the goal state)

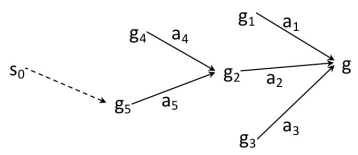
Inverse State Transitions

if a is relevant for g, then $\gamma^{-1}(g,a) = pre(a) \cup (g - eff(a))$

if a is not relevant for g, then $\gamma^{-1}(g,a)$ is undefined

Backward Search

Backward-search(Σ, s_0, g_0)
  $\pi \leftarrow \langle\rangle$; $g \leftarrow g_0$ (*i*)
  loop
    if s_0 satisfies g then return π
    $A' \leftarrow \{a \in A \mid a \text{ is relevant for } g\}$
    if $A' = \varnothing$ then return failure
    nondeterministically choose $a \in A'$
    $g \leftarrow \gamma^{-1}(g, a)$ (*ii*)
    $\pi \leftarrow a.\pi$ (*iii*)



Cycle checking:

- After line (*i*), put *Visited* $\leftarrow \{g_0\}$
- After line (*iii*), put this:
    if $g \in$ *Visited* then
      return failure
    Visited $\leftarrow$ *Visited* $\cup \{g\}$
  or this:
    if $\exists g' \in$ *Visited* s.t. $g \Rightarrow g'$ then
      return failure
    Visited $\leftarrow$ *Visited* $\cup \{g\}$

- With cycle checking, sound and complete
  - If (Σ, s_0, g_0) is solvable, then at least one execution trace will find a solution

motivation for Backward search?
to reduce the branching factor

(but it doesn't accomplish that)

↳ lifting solves this!

- Solve this by *lifting*:
  - When possible, leave variables uninstantiated
  - Most implementations of Backward-search do this

8/36 chap 2c

Lifted Backward Search

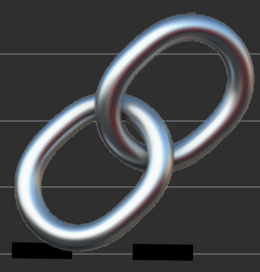
- Like Backward-search but much smaller branching factor
- Must keep track of what values were substituted for which parameters
  - I won't discuss the details
  - PSP (later) does something similar

9/36 code

Deliberation with deterministic models

Create a heuristic function:

- Weaken some of the constraints, get additional solutions
- *Relaxed* planning domain Σ' and relaxed problem $P' = (\Sigma', s_0, g')$ such that
  - every solution for P is also a solution for P'
  - additional solutions with lower cost
- Suppose we have an algorithm A for solving planning problems in Σ'
  - Heuristic function $h_A(s)$ for P:
    - Find a solution π' for (Σ', s, g'); return cost(π')
    - Useful if A runs quickly
  - If A always finds optimal solutions, then h_A is admissible

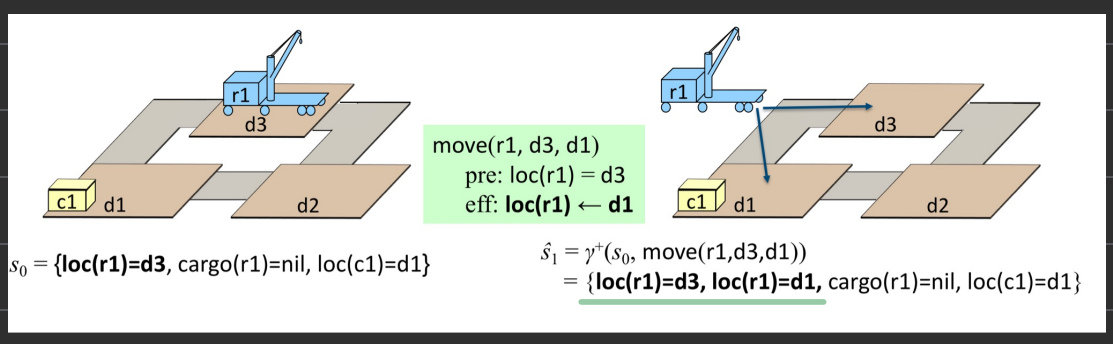


Domain-independent Heuristics

1. Delete-Relaxation Heuristics

allow a state variable to have more than 1 value at the same time

↳ σταν κάνω assign ένα value (after effect) κάνω keep και το old one too



Relaxed Applicability

Action a is r-applicable in a relaxed state $\hat{s}$ if an r-subset of $\hat{s}$ satisfies a's preconditions

=> if a is r-applicable then: $\gamma^+(\hat{s}, a) = \hat{s} \cup \gamma(s, a)$

Suppose:

έχω δύνοδο με actions $a_1, a_2 \ldots$ Consplan $\pi = \langle a_1, a_2 \ldots a_n \rangle$ και κάνω r-apply αυτά τα actions σε states $\hat{s_0}, \hat{s_1} \ldots$ με την σειρά . r-apply a_1 in $\hat{s_0}$ => έχω $\hat{s_1} = \gamma^+(\hat{s_0}, a_1)$

a_2 in $\hat{s_1}$ => έχω $\hat{s_2} = \gamma^+(\hat{s_1}, a_2)$

$\vdots$

a_n in $\hat{s_{n-1}}$ => $\hat{s_n} = \gamma^+(\hat{s_{n-1}}, a_n)$

APA π is r-applicable in $\hat{s_0}$ => $\gamma^+(\hat{s_0}, \pi) = \hat{s_n}$

example: Εχω $\hat{s_0}$ αρχικο state και $\hat{s_2}$ τελικό
αρα $\gamma(s_0, \langle move(\dots), load(\dots) \rangle) = \hat{s_2}$
↳ a plan π

more definitions:
r-state $\hat{s}$ r-satisfies a formula g if an r-subset of $\hat{s}$ satisfies g

Relaxed Solution → a plan π such that not $\gamma^+(s_0, \pi)$ r-satisfies g.

Planning problem $P = (\Sigma, s_0, g)$

Optimal Relaxed Solution heuristic:

$h^+(s)$ = minimum cost of ALL relaxed solutions for (Σ, s, g)

Example: Greedy Best Forward Search

- GBFS with initial state s_0, goal g, heuristic h^+
- Applicable actions a_1, a_2 produce states s_1, s_2
- GBFS computes $h^+(s_1)$ and $h^+(s_2)$, chooses the state that has the lower h^+ value

Fast-Forward Heuristic

- Every state is also a relaxed state
- Every solution is also a relaxed solution

$h^+(s)$ → minimum cost of all relaxed solutions

→ thus h^+ admissible

PROBLEM: it's NP hard

✓Solution:

- Fast-Forward Heuristic, h^{FF}
  - An approximation of h^+ that's easier to compute
    - Upper bound on h^+
  - Name comes from a planner called *Fast Forward*

Preliminaries

- Let A_1 be a set of actions that all are r-applicable in s_0
  - Can r-apply them in any order and get same result
  - $\hat{s}_1 = \gamma^+(\hat{s}_0, A_1) = \hat{s}_0 \cup \text{eff}(A_1)$
    - where $\text{eff}(A) = \cup\{\text{eff}(a) \mid a \in A\}$

- Suppose A_2 is a set of actions that are r-applicable in $\hat{s}_1$
  - $\hat{s}_2 = \gamma^+(\hat{s}_0, \langle A_1, A_2\rangle) = \hat{s}_0 \cup \text{eff}(A_1) \cup \text{eff}(A_2)$

  …
- Define $\gamma^+(\hat{s}_0, \langle A_1, A_2, \ldots, A_n\rangle)$ in the obvious way

Fast-Forward Heuristic

i.e., no proper subset is a relaxed solution

HFF(Σ, s, g): // *find a minimal relaxed solution, return its cost*

1. At each iteration, include all r-applicable actions

1*

// construct a relaxed solution $\langle A_1, A_2, \ldots, A_k\rangle$:
$\hat{s}_0 \leftarrow s$
for $k = 1$ by 1 until $\hat{s}_k$ r-satisfies g
  $A_k \leftarrow$ {all actions r-applicable in $\hat{s}_{k-1}$}; $\hat{s}_k \leftarrow \gamma^+(s_{k-1}, A_k)$
  if $k > 1$ and $\hat{s}_k = \hat{s}_{k-1}$ then return ∞ // *there's no solution*

2. At each iteration, choose a minimal set of actions that r-achieve $\hat{g}_i$

2*

// extract minimal relaxed solution $\langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k\rangle$:
$\hat{g}_k \leftarrow g$
for $i = k, k{-}1, \ldots, 1$:
  $\hat{a}_i \leftarrow$ any minimal subset of A_i such that $\gamma^+(\hat{s}_{i-1}, \hat{a}_i)$ r-satisfies $\hat{g}_i$
  $\hat{g}_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i)$
return $\sum$ costs of the actions in $\hat{a}_1, \ldots, \hat{a}_k$ // *upper bound on* h^+

$\text{pre}(\hat{a}_i) = \cup\{\text{pre}(a) \mid a \in \hat{a}_i\}$
$\text{eff}(\hat{a}_i) = \cup\{\text{eff}(a) \mid a \in \hat{a}_i\}$

ambiguous → • Define $h^{\text{FF}}(s)$ = the value returned by HFF(Σ, s, g)

Example: Greedy Best Forward Search (GBFS)

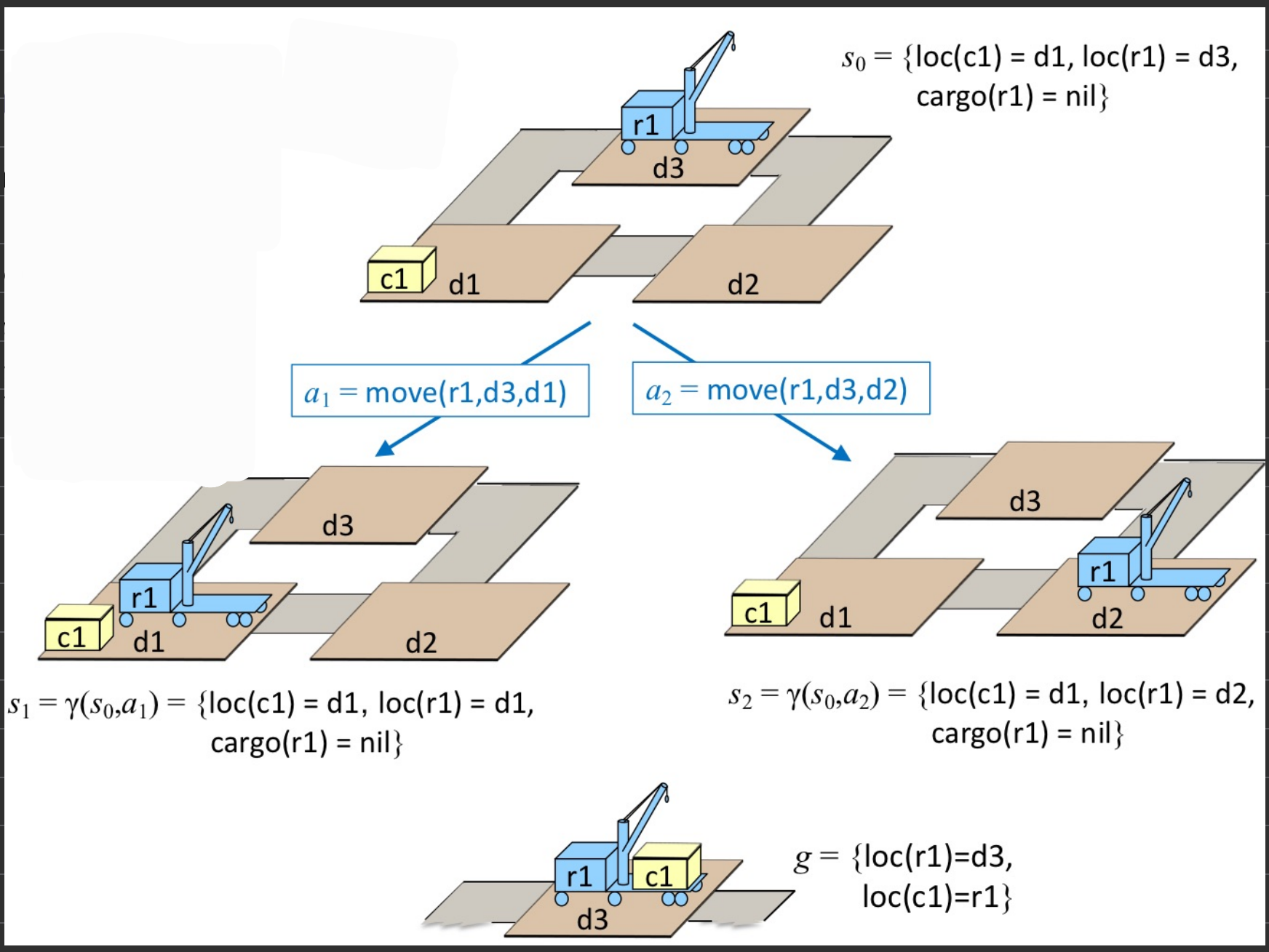
- GBFS με initial state: s_0, goal: g, heuristic h^{FF}
- 2 applicable actions: a_1, a_2

  ⇒ resulting states: s_1, s_2

  GBFS computes $h^{FF}(s_1)$, $h^{FF}(s_2)$

  → chooses the state that has the lower h^{FF} value

EXAMPLE



- computing $h^{FF}(s_1)$
  1. construct a relaxed solution
     at each step, include all r-applicable actions

1*

Below is a relaxed solution $\langle A_1, A_2 \ldots A_n \rangle$

$\hat{s}_0 \leftarrow s$

For $k=1$ do until $\hat{s}_k$ r-satisfies g

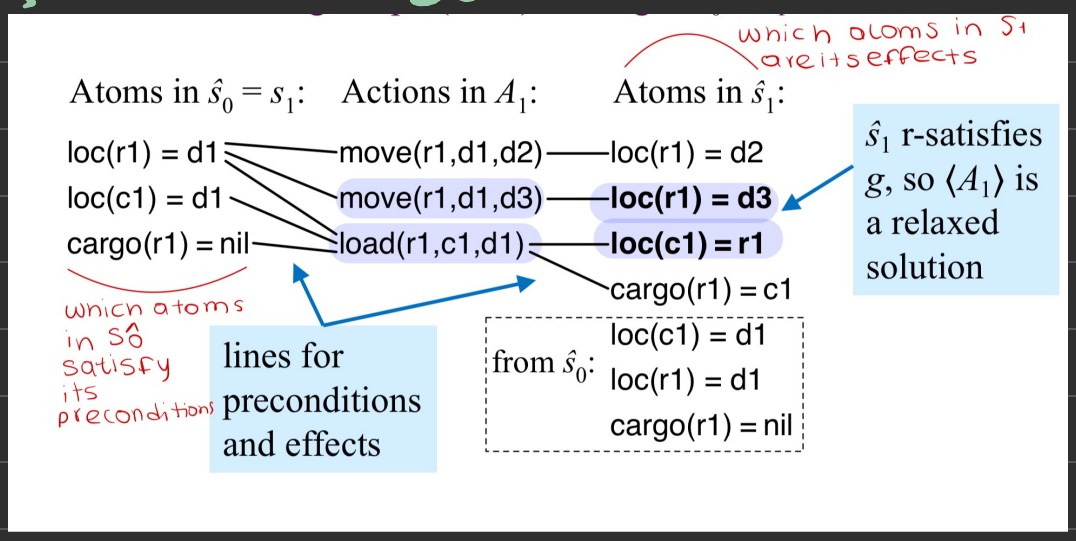
$A_k \leftarrow$ {all actions r-applicable to $\hat{s}_{k-1}$}

$\hat{s}_k \leftarrow \gamma^+(s_{k-1}, A_k)$

if $k>1$ and $\hat{s}_k = \hat{s}_{k-1}$ then return ∞

Relaxed Planning Graph (RPG)

first one $\hat{s}_0 = s_1$



2. extract a minimal relaxed solution
   · if u remove any actions from it
   => it is no longer a relaxed solution

2* bplbkw minimal relaxed solution $\langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle$

$\hat{g}_k \leftarrow g$

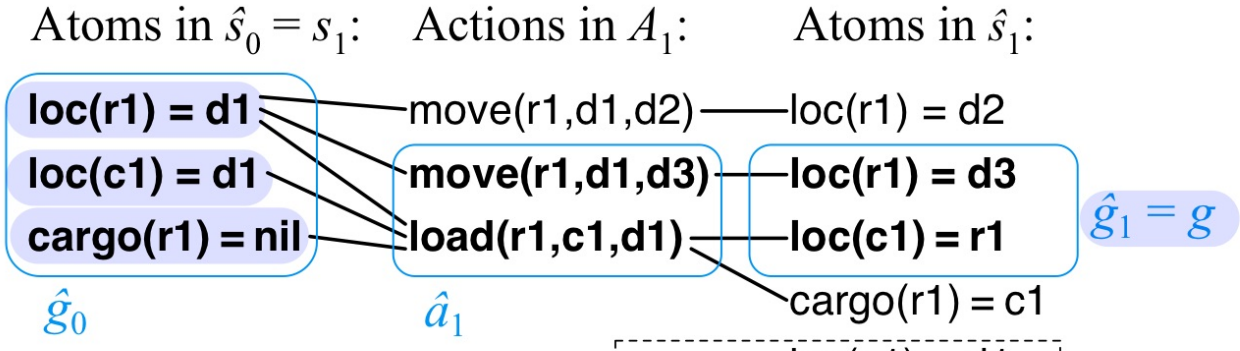
For $i = k, k-1, \ldots, 1$

$\hat{a}_i \leftarrow$ min subset of A_i such that $\gamma^+(\hat{s}_{i-1}, \hat{a}_i)$ r-satisfies $\hat{g}_i$

$g_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i)$

ion

Solution extraction starting at $\hat{g}_1 = g$



- $\hat{a}_1$ is a minimal set of actions such that $\gamma^+(\hat{s}_0, \hat{a}_1)$ r-satisfies $\hat{g}_1$
  - $\langle \hat{a}_1 \rangle$ is a minimal relaxed solution
- Two actions, each with cost 1, so $h^{\text{FF}}(s_1) = 2$

compute $h^{FF}(s_2)$ and compare them!

1st step: relaxed solution construction

Start $\hat{s}_0 = s_2$

s_0 = {loc(c1) = d1, loc(r1) = d3, cargo(r1) = nil}

s_2 = {loc(r1)=d2, cargo(r1)=nil, loc(c1)=d2}

atoms in $\hat{s}_0 = s_2$

loc(r1) = d2
cargo(r1) = nil
loc(c1) = d2

Actions in A1

..?

Properties

Running time is polynomial in $|A| + \sum_{x \in X} |Range(x)|$

$h^{FF}(s)$ = value returned by HFF(Σ, s, g)
$= \sum_i cost(\hat{a}_i)$
$= \sum_i \sum \{cost(a) \mid a \in \hat{a}_i\}$

- each $\hat{a}_i$ is a minimal set of actions such that $\gamma^+(s_{i-1}, \hat{a}_i)$ r-satisfies $\hat{g}_i$

  * minimal does not mean smallest

- $h^{FF}(s)$ is <u>ambiguous</u>

  ↓

  depends on which minimal sets we choose

- $h^{FF}(s)$ not admissible

- $h^{FF}(s) \geq h^+(s)$ = Smallest cost of any relaxed plan from s to goal

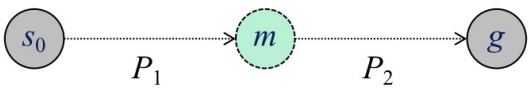
Landmark heuristics

*simple words:
notable feature that stands out because it is different from its surroundings

↳ why use them?

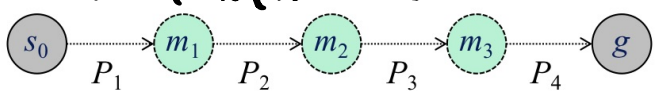
Why are Landmarks Useful?

- Can break a problem down into smaller subproblems



- Suppose m is a landmark
  - Every solution to P must achieve m
- Possible strategy:
  - find a plan to go from s_0 to any state s_1 that satisfies m
  - find a plan to go from s_1 to any state s_2 that satisfies g

"βαζω το goal σε πολλα m (m1,2,..) και προσπαθω ενα ενα σιγουρα"



- Suppose m_1, m_2, m_3 are landmarks
  - Every solution to P must achieve m_1, then m_2, then m_3
- Possible strategy:
  - find a plan to go from s_0 to any state s_1 that satisfies m_1
  - find a plan to go from s_1 to any state s_2 that satisfies m_2
  - …

Computing Landmarks

- Some landmarks are easier to find – polynomial time
  - Several procedures for finding them
  - I'll show you one based on relaxed planning graphs

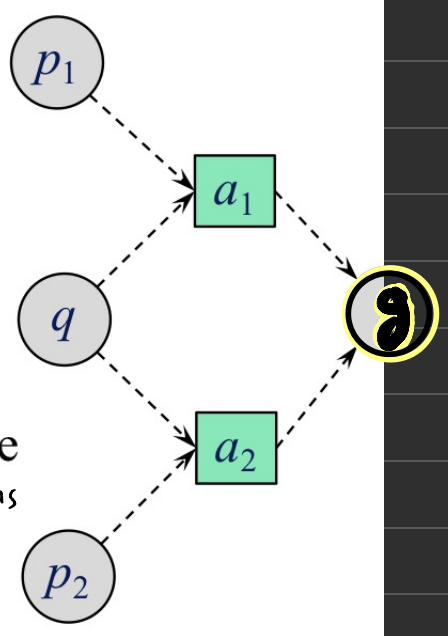
RPG
relaxed planning graphs

why use them?

- Easier to solve relaxed planning problems
- Easier to find landmarks for them
- A landmark for a relaxed planning problem is also a landmark for the original planning problem

IDEA:

- Key idea: if φ is a landmark, get new landmarks from the preconditions of the actions that achieve φ
  - goal g
  - {actions that achieve g} = $\{a_1, a_2\}$
    - pre(a_1) = $\{p_1, q\}$
    - pre(a_2) = $\{p_2, q\}$
  - To achieve g, must achieve $(p_1 \wedge q) \vee (p_2 \wedge q)$ — reasons
    - same as $q \wedge (p_1 \vee p_2)$
  - Landmarks:
    - q — g is a landmark
    - $p_1 \vee p_2$ — these are the preconditions of the actions that achieved g! so they are also landmarks



ένα φ landmark
⇒ new landmark is in preconditions των actions που επιλέγω achieve το φ

RPG-based Landmark Computation

- Suppose goal is $g = \{g_1, g_2, \ldots, g_k\}$
  - Trivially, every g_i is a landmark
- Suppose g_1 = loc(r1)=d1
  - Two actions can achieve g_1:
    move(r1,d3,d1) and move(r1,d2,d1)
- Preconditions loc(r1)=d3 and loc(r1)=d2
- New landmark:
  - φ' = loc(r1)=d3 ∨ loc(r1)=d2

goal/landmark

act.

pres.

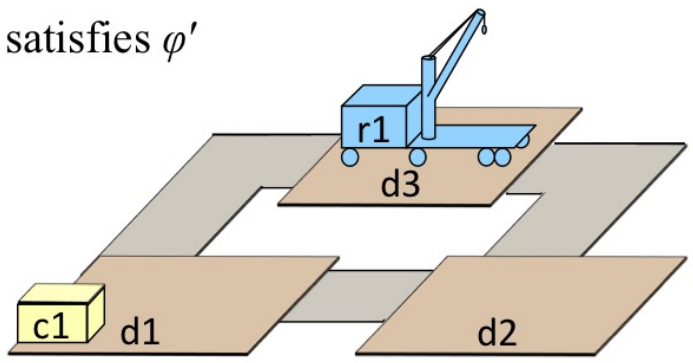
landmarks

move(r, d, e)
pre: loc(r)=d
eff: loc(r) ← e

load(r, c, l)
pre: cargo(r)=nil, loc(c)=l, loc(r)=l
eff: cargo(r) ← c, loc(c) ← r

unload(r, c, l)
pre: loc(c)=r, loc(r)=l
eff: cargo(r) ← nil, loc(c) ← l

- In this example, s_0 satisfies φ'



s_0 = {loc(r1)=d3, cargo(r1)=nil, loc(c1)=d1}

slides 26-29 full step by step computation!
chap 2b

Example

1. βαλε στο queue οσιδηποτε g_i δεν καλει satisfy το goal g

   queue = $\langle loc(c_1) = r_1 \rangle$

   Landmark = $\emptyset$

2. · βγαλε το g_i απο το queue (για να το κανουμε test, στο case μας ειναι προφανως εκει το queue) και βαλε το g_i στα Landmarks

   · βαλε στο R τα actions των οποιων τα effects καλουν include το g_i (σημαινει το g_i ειναι ενα απο τα effects αυτων των actions)

   queue = $\langle\rangle$

   $g_i = loc(c_1) = r_1$

   Landmarks = $loc(c_1) = r_1$

   $R = \{ load(r_1, c_1, d_1),\ load(r_1, c_1, d_2),\ load(r_1, c_1, d_3) \}$

   * why?
   Διοτι εαυτο που μας δινεται ΜΟΝΟ το load εχει σαν effect το $loc(c) \leftarrow r$! στα βαλω ολα τα cases των loads (το d_1, d_2, d_3)

3. generate RPG (relaxed planning graph) πως ? καλω A/R , σταματω οταν $\hat{S}_k = \hat{S}_{k-1}$

A/R = { move & unload actions }

why?

G ← load, move, unload − load

A − R

RPG using A/R

$\hat{S}_0$:
- $loc(c_1) = d_1$
- $loc(r_1) = d_3$
- $cargo(r_1) = nil$

A_1:
- $move(r_1, d_3, d_1)$
- $move(r_1, d_3, d_2)$

both $\hat{S}_1, \hat{S}_2$:
- $loc(r_1) = d_1$
- $loc(r_1) = d_2$
- $loc(c_1) = d_1$
- $loc(r_1) = d_3$
- $cargo(r_1) = nil$

(from $\hat{S}_0$)

4. βαζω στο N τα actions του R που είναι r-applicable στο $\hat{S_k}$

αν $N = \emptyset$ → return false

queue = <>

gi = loc(c1) = r1

Landmarks = loc(c1) = r1

R = { load(r1, c1, d1)
load(r1, c1, d2)
load(r1, c1, d3)

N = { load(r1, c1, d1) ←

↳ why? because these are the preconditions of this action

RPG using A/R

$\hat{S_0}$:
loc(c1) = d1
loc(r1) = d3
cargo(r1) = nil

A1:
move(r1, d3, d1)
move(r1, d3, d2)

both $\hat{S_1}$, $\hat{S_2}$:
loc(r1) = d1
loc(r1) = d2 *
loc(c1) = d1
loc(r1) = d3
cargo(r1) = nil

from $\hat{S_0}$

5. Βαζε στα preconds → $U\{pre(a) \mid a \in N\} \setminus S_0$

[όλα τα preconditions των actions που έχει το σύνολο N μείον τα preconditions του S_0]

βαλε στο $\Phi \rightarrow \{p_1 \vee p_2 \vee \dots \vee p_m \mid m \leq u$, every action in N has at least one p_i as a precond, & every $p_i \in$ preconds$\}$

[όλα όσα actions έχουν τουλ 1 p_i ως precondition και κάθε precondition τους ανήκει στο σύνολο που φτιάχνει πριν (preconds)]

queue = <>

gi = loc(c1) = r1

Landmarks = loc(c1) = r1

R = { load(r1, c1, d1)
load(r1, c1, d2)
load(r1, c1, d3)

N = { load(r1, c1, d1)

load(r1, c1, d1)

pre: 1 cargo(r1) = nil
2 loc(c1) = d1
3 loc(r1) = d1

in so

(1+2+3) - (1+2)
≡
3
≡
loc(r1) = d1

preconds = { loc(r1) = d1 }

Φ = { loc(r1) = d1 }

queue = < loc(r1) = d1 >

вдет по queue то φ ∈ Φ

queue = < loc(r1) = d1 >
Landmarks = { loc(c1) = r1 }

6. Bjade gi ano queue -> put it in landmarks

R ← ra actions whose effects include gi

if So satisfies pre(a) for some a ∈ R
then return Landmarks

queue = <>

gi = loc(c1) = r1

Landmarks = {loc(c1) = r1,
loc(r1) = d1}

R = {move(r1, d2, d1)
move(r1, d3, d1)}

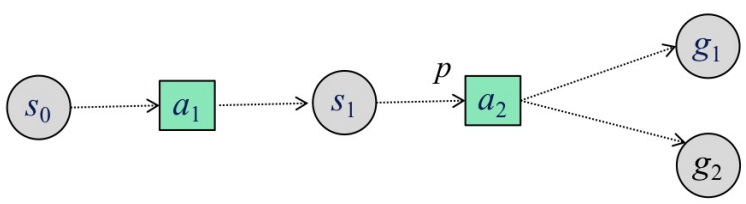
So satisfies pre(move(r1, d3, d1))

why? action move(r1, d3, d1) can be done
with preconditions loc(r1) = d3

this precond is in So

Landmark Heuristic

- Every solution to the problem needs to achieve all the computed landmarks
- One possible heuristic:
  - $h^{sl}(s)$ = number of landmarks returned by RPG-Landmarks
- Not admissible



$g = \{g_1, g_2\}$
Three landmarks: g_1, g_2, p
Optimal plan: $\langle a_1, a_2 \rangle$, length = 2

- There are other more-advanced landmark heuristics
  - Some of them are admissible
  - Check textbook for references

Max-Cost & Additive Cost Heuristics

visualized as an AND/OR search
going backward from g

h^{max} is → the cost of an optimal solution to a relaxed problem in which a goal (set of literals g, or precond of an action) can be reached by achieving just ONE of the goal's literals
=> that is THE MOST Expensive one to achieve

$$\Delta^{\max}(s, g) = \max_{g_i \in g} \Delta^{\max}(s, g_i);$$

$$\Delta^{\max}(s, g_i) = \begin{cases} 0, & \text{if } g_i \in s, \\ \min\{\Delta^{\max}(s, a) \mid a \in A \text{ and } g_i \in \text{eff}(a)\}, & \text{otherwise;} \end{cases}$$

$$\Delta^{\max}(s, a) = \text{cost}(a) + \Delta^{\max}(s, \text{pre}(a)).$$

In a planning problem $P = (\Sigma, s_0, g)$, the *max-cost heuristic* is

$$h^{\max}(s) = \Delta^{\max}(s, g).$$

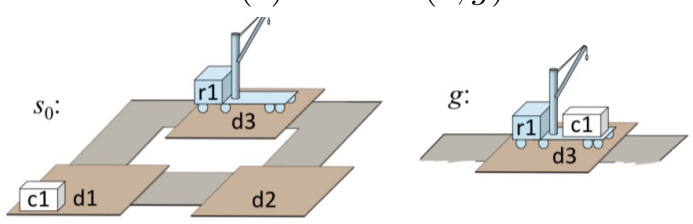


Figure 2.4: Initial state and goal for Example 2.21.

example

Example 2.21. Figure 2.4 shows a planning problem $P = (\Sigma, s_0, g)$ in a planning domain $\Sigma = (B, R, X, \mathcal{A})$ that is a simplified version of the one in Figure 2.3. B includes one robot, one container, three docks, no piles, and the constant nil:

$$B = \mathit{Robots} \cup \mathit{Docks} \cup \mathit{Containers} \cup \{\mathsf{nil}\};$$
$$\mathit{Robots} = \{\mathsf{r}_1\};$$
$$\mathit{Docks} = \{\mathsf{d}_1, \mathsf{d}_2, \mathsf{d}_3\};$$
$$\mathit{Containers} = \{\mathsf{c}_1\}.$$

There are no rigid relations, that is, $R = \varnothing$. There are two state variables, $X = \{\mathsf{cargo(r1)}, \mathsf{loc(c1)}\}$, with

$$\text{Range}(\mathsf{cargo(r1)}) = \{\mathsf{c1}, \mathsf{nil}\};$$
$$\text{Range}(\mathsf{loc(c1)}) = \{\mathsf{d1}, \mathsf{d2}, \mathsf{d3}, \mathsf{r1}\}.$$

$\mathcal{A}$ contains three action templates:

load(r, c, l)
pre: cargo$(r) = nil, loc(c) = l$, loc$(r) = l$
eff: cargo$(r) \leftarrow c$, loc$(c) \leftarrow r$
cost: 1

So = {loc(r1) = d3, cargo(r1) = nil, loc(c1) = d1}

unload(r, c, l)
pre: cargo$(r) = c$, loc$(r) = l$
eff: cargo$(r) \leftarrow$ nil, loc$(c) \leftarrow l$
cost: 1

move(r, d, e)
pre: loc$(r) = d$
eff: loc$(r) \leftarrow e$
cost: 1

r1, d3
nil, d1, d2

The action templates' parameters have the following ranges:

Range$(c) = \mathit{Containers}$; Range$(d) = $ Range$(e) = \mathit{Docks}$;
Range$(l) = \mathit{Locations}$; Range$(r) = \mathit{Robots}$.

P's initial state and goal are

$$s_0 = \{\mathsf{loc(r1)} = \mathsf{d3}, \mathsf{cargo(r1)} = \mathsf{nil}, \mathsf{loc(c1)} = \mathsf{d1}\};$$
$$g = \{\mathsf{loc(r1)} = \mathsf{d3}, \mathsf{loc(c1)} = \mathsf{r1}\}.$$

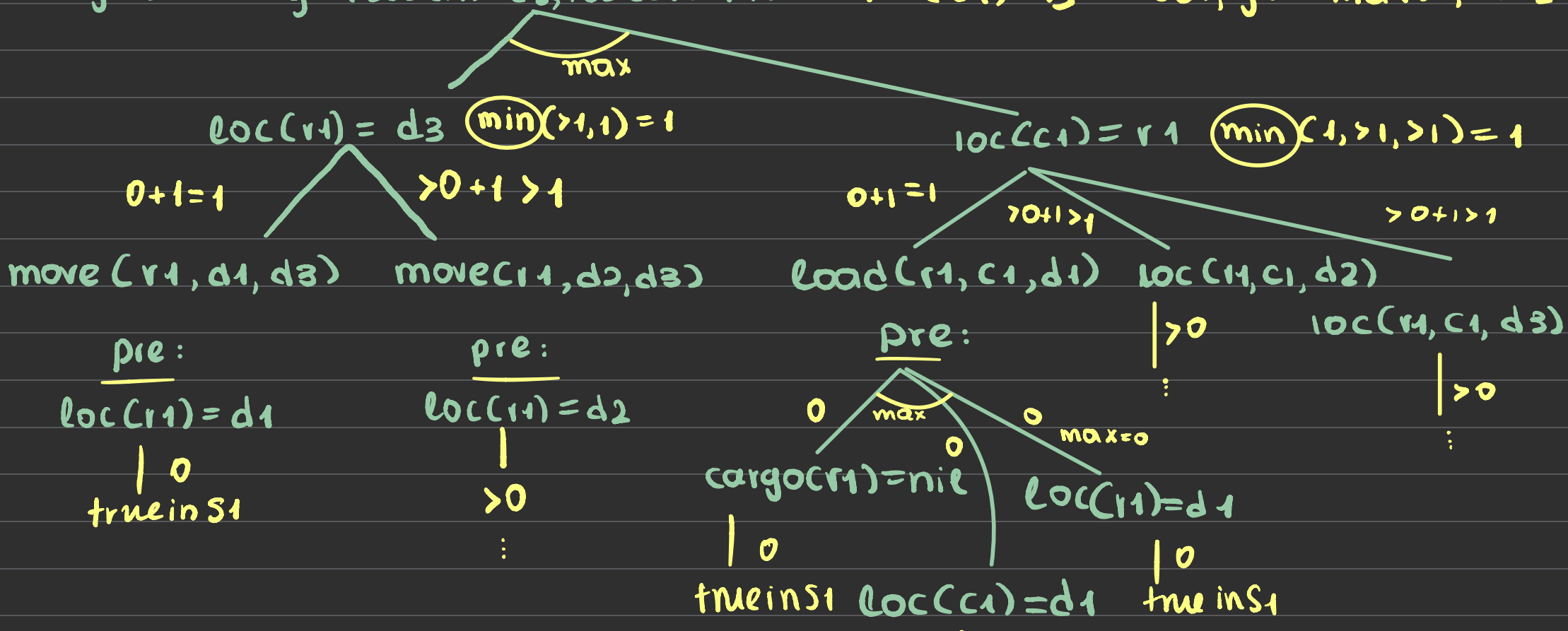
Suppose we are running GBFS (see Section 2.2.7) on P. In s_0, there are two applicable actions: $a_1 = \mathsf{move(r1, d3, d1)}$ and $a_2 = \mathsf{move(r1, d3, d2)}$. Let

$$s_1 = \gamma(s_0, a_1) = \{\mathsf{loc(r1)} = \mathsf{d1}, \mathsf{cargo(r1)} = \mathsf{nil}, \mathsf{loc(c1)} = \mathsf{d1}\}; \quad (2.10)$$
$$s_2 = \gamma(s_0, a_2) = \{\mathsf{loc(r1)} = \mathsf{d2}, \mathsf{cargo(r1)} = \mathsf{nil}, \mathsf{loc(c1)} = \mathsf{d1}\}. \quad (2.11)$$

solution ↓

Για S1:

goal: $g = \{loc(r1) = d3, loc(c1) = r1\}$ $h^{max}(s_1) = \Delta^{max}(s_1, g) = max(1,1) = 1$



$$\Delta^{\max}(s, g) = \max_{g_i \in g} \Delta^{\max}(s, g_i);$$

$$\Delta^{\max}(s, g_i) = \begin{cases} 0, & \text{if } g_i \in s, \\ \min\{\Delta^{\max}(s, a) \mid a \in A \text{ and } g_i \in \text{eff}(a)\}, & \text{otherwise;} \end{cases}$$

$$\Delta^{\max}(s, a) = \text{cost}(a) + \Delta^{\max}(s, \text{pre}(a)).$$

In a planning problem $P = (\Sigma, s_0, g)$, the *max-cost heuristic* is

$$h^{\max}(s) = \Delta^{\max}(s, g).$$

κανω max για preconditions

↓

min κανω μονο στο gi, οπιαδηποτε απο τα 2 literals (υπο goals) που διαχωριζαμε ξ'α υπο το example

h^{add} is the same as h^{max} but instead it adds up the costs of each set of literals rather than taking their max

σπ3 οταν κανω break a goal or a state variable σε πολλα parts → εκει κανω το add or max

οταν διαλεγω ενα state variable σε possible actions κανω min

=> h^{max} & h^{add} polynomnial time complexity.