chap 2c

Plan-Space Search

- Constraint satisfaction problem

→ using constraint-satisfaction techniques → to get flexible solutions (e.g. plans with partially ordered actions)

BASIC IDEA

- Backward search from goal
- Each node of search space
  - ↓
  - partial plan that contains flaws (*)
    - ↓
    - remove the flaws by making refinements
      - ↓
      - if successful → we get a PARTIALLY ORDERED SOLUTION

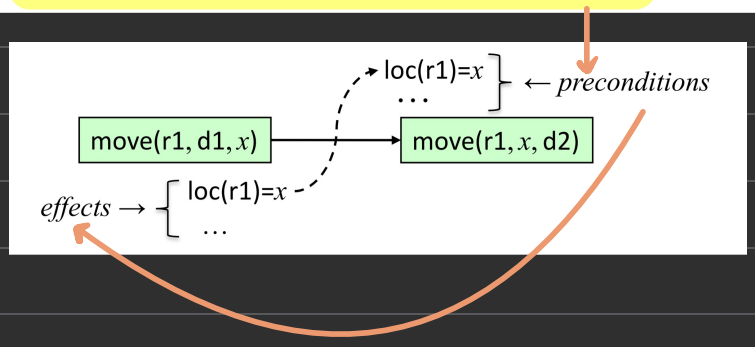
Definitions

✓ Partially ordered plan
- partially ordered set of nodes
- each node has an action

✓ Partially ordered solution for a planning problem P
- partially ordered plan π:
  such that every total ordering of π is a solution for P

✓ partial plan
- partially ordered set of nodes
  - ↳ that contain partially instantiated actions
    - Griglioni
- inequality constraints
- casual links (↗)

- constraint: action *a must* be the action that establishes action *b*'s precondition *p*



✱ Flaws: 1. Open Goals

Action (b), precondition (p)

- p is an *open goal* if there is no causal link for p

↓

Resolve the flaw by creating a casual link

- Find an action a (either already in π, or add it to π) that can *establish* p
  - can precede b
  - can have p as an effect
- Do substitutions on variables to make a assert p (replacements → result)
- Add an ordering constraint $a \prec b$
- Create a causal link from a to p (a makes p true)

(a)
αυτο το action μπορει να
1. βαλθει πριν το b.
2. να εχει το p ως effect

eg. 15/36

✱ Flaws: 2. Threads

το effect κάποιου action κάνει threat όταν μπορεί να μπει ανάμεσα στο casual support κάποιου action & να το κάνει modify το effect εκείνο (με replacements μεταβλητών)

(l) casual link from an (effect) of action (a) to a precondition (p) of action (b)

⇒ Action (c) threatens (l) if (c) may come between (a) & (b) & (c) and may affect (p)

- "c may come between a and b" means the plan's current ordering constraints don't prevent it
  - plan doesn't already have $c \prec a$ or $b \prec c$

allows c to do that :)

↳ we have to do this!

- "c may affect p" means
  - can substitute values for variables such that c's effects either make p true or make p false ☹

eg.16/36

↓

Resolving Threads

- Suppose action c threatens a causal link l from an effect of action a to a precondition p of action b

3 possible resolves:

1. add a precedence constraint $c \prec a$
2. add a precedence constraint $b \prec c$
3. add inequality constraints that prevent c from affecting p

* make sure that c happens before a (so it can't interfere with a's effects)

* same with above

* πχ. r2 ≠ r1
robot r2 can't be r1

- Each of these is applicable iff it doesn't make the plan inconsistent
  - e.g., 2 isn't applicable if the plan already has $c \prec b$

PSP Algorithm

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

PSP Algorithm:

loop:

αν δεν υπαρχουν Flaws return the plan

else

select f from Flaws

R ← όλα τα resolvers για το f που είναι possible να γίνουν

αν δεν υπαρχουν resolvers → Failure

chose ρ ∈ R

→ return π ← ρ(π)

Full example explanation

- 2 open goals
- no threats

select loc(r1) = d2

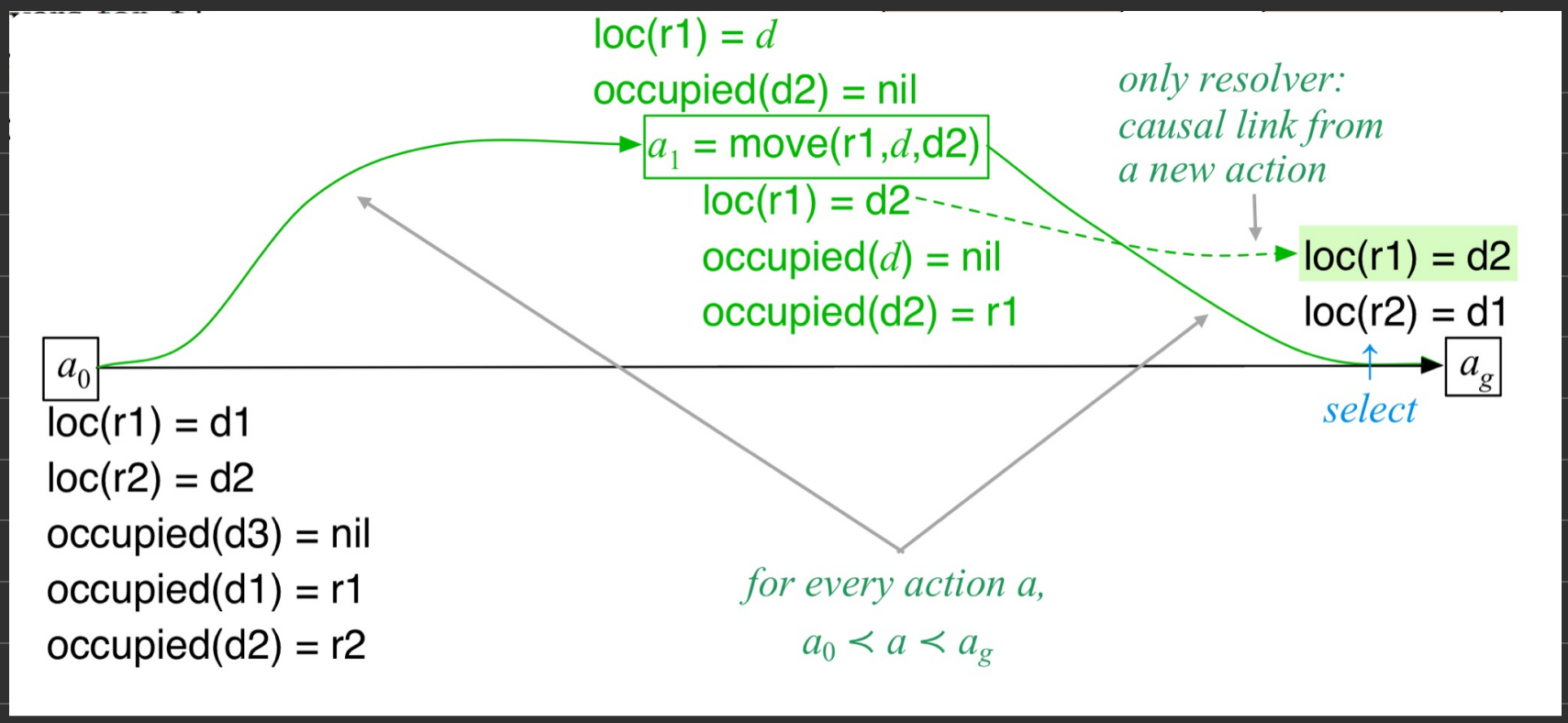
open goal

- Find an action a (either already in π, or add it to π) that can *establish* p
  - can precede b
  - can have p as an effect
- Do substitutions on variables to make a assert p
- Add an ordering constraint $a \prec b$
- Create a causal link from a to p

action a : πρεπει να εφαρμοζει το p
δηλαδη τα effects της να ειναι το p

a_1 = move(r1, d, d2) $\xrightarrow{\text{effect}}$ loc(r1) = d2

p => loc(r1) = d2
a => move(r1, d, d2)

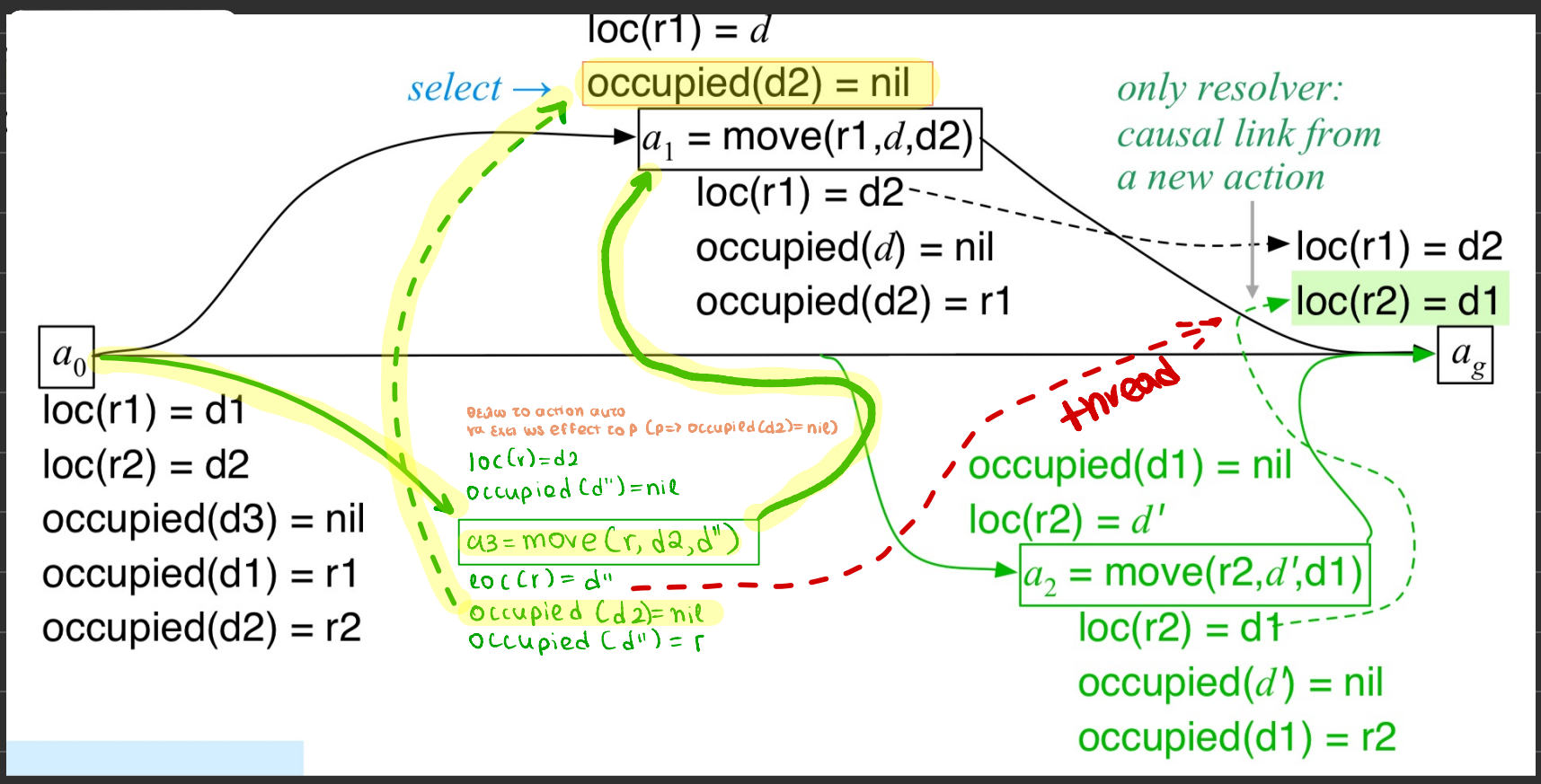


· 4 open goals
· no threats
=> p => loc(r2) = d1

→

· 5 open goals
· 1 threat

now we select this (bc it is a p with no causal link)



now action a3 προξενεσα εναν threat στο a2 και ag => threat

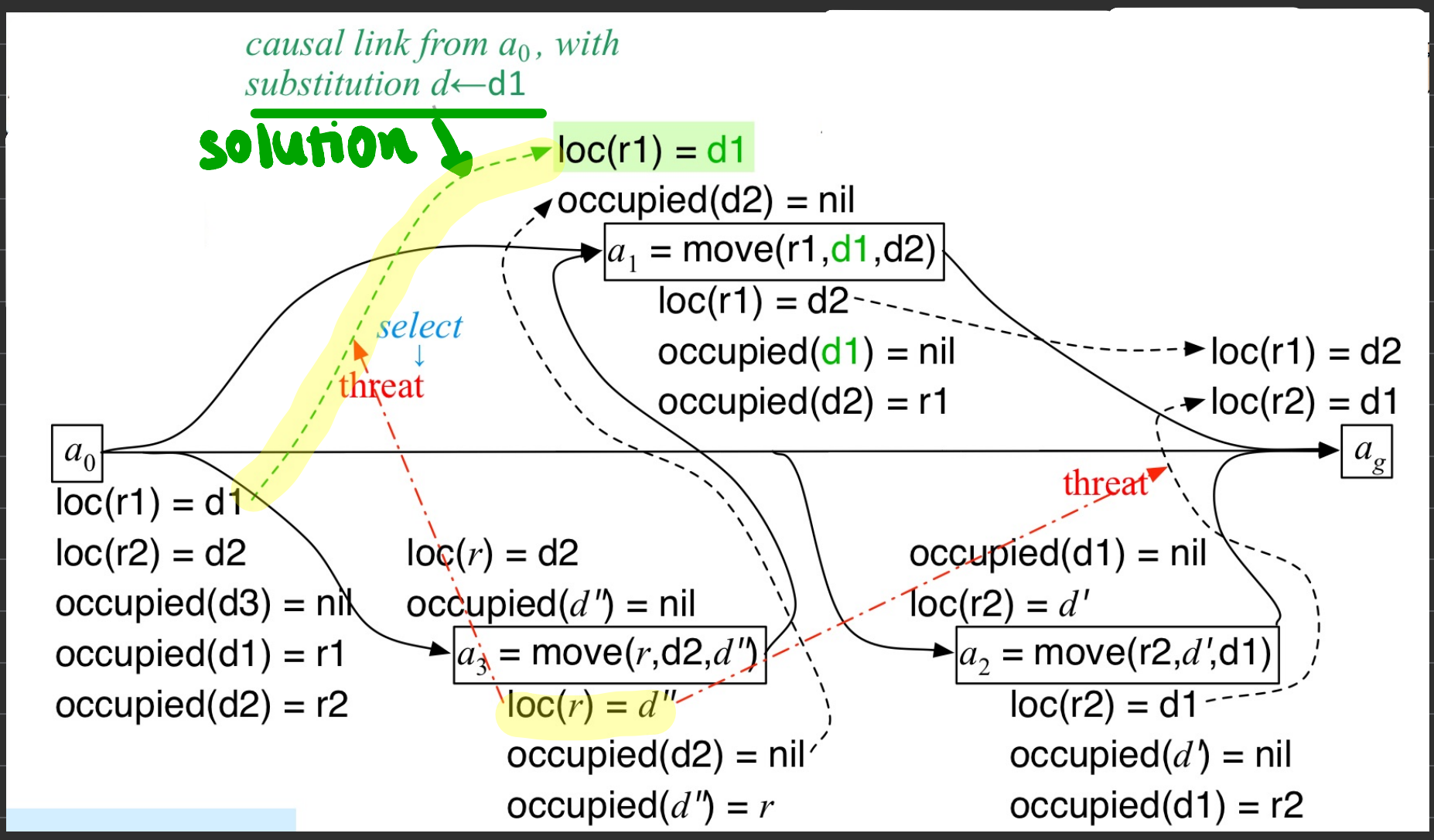
- "*c* may come between *a* and *b*" means the plan's current ordering constraints don't prevent it
  - plan doesn't already have $c \prec a$ or $b \prec c$

allows c to do that :)

↳ we have to do this!

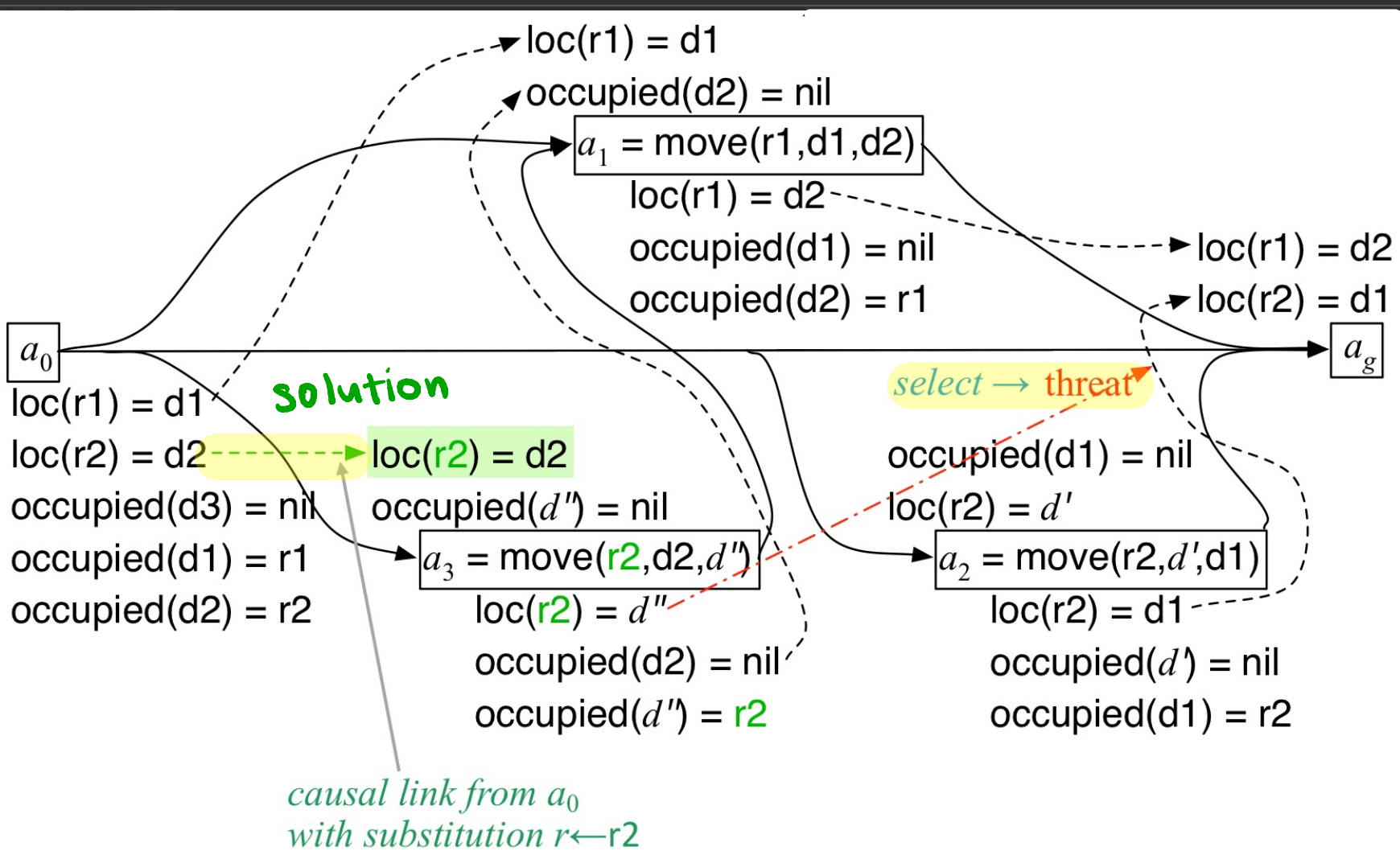
- "*c* may affect *p*" means
  - can substitute values for variables such that *c*'s effects either make *p* true or make *p* false :)

- 4 open goals
- 2 threats

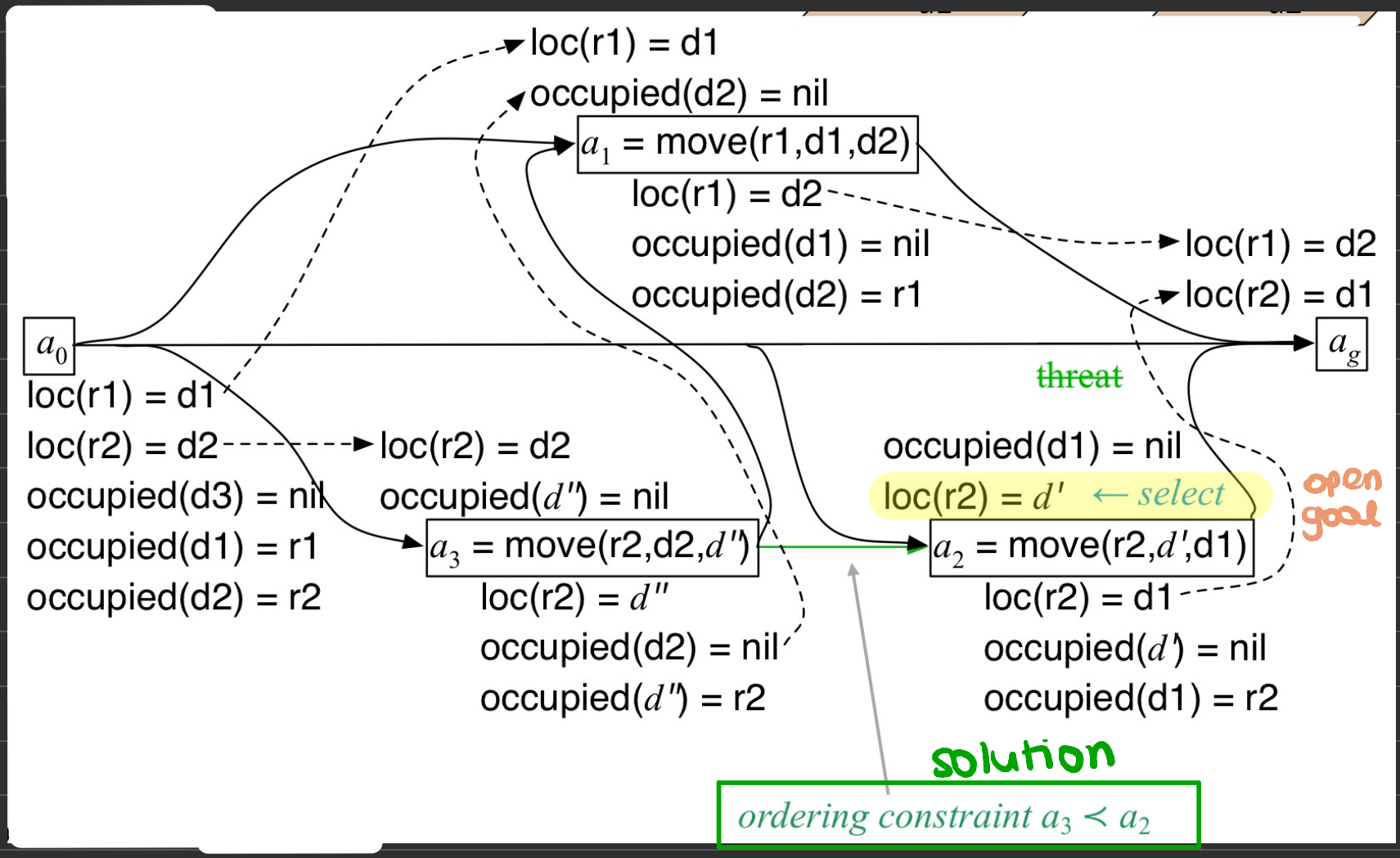


- 4 open goals
- 1 threat

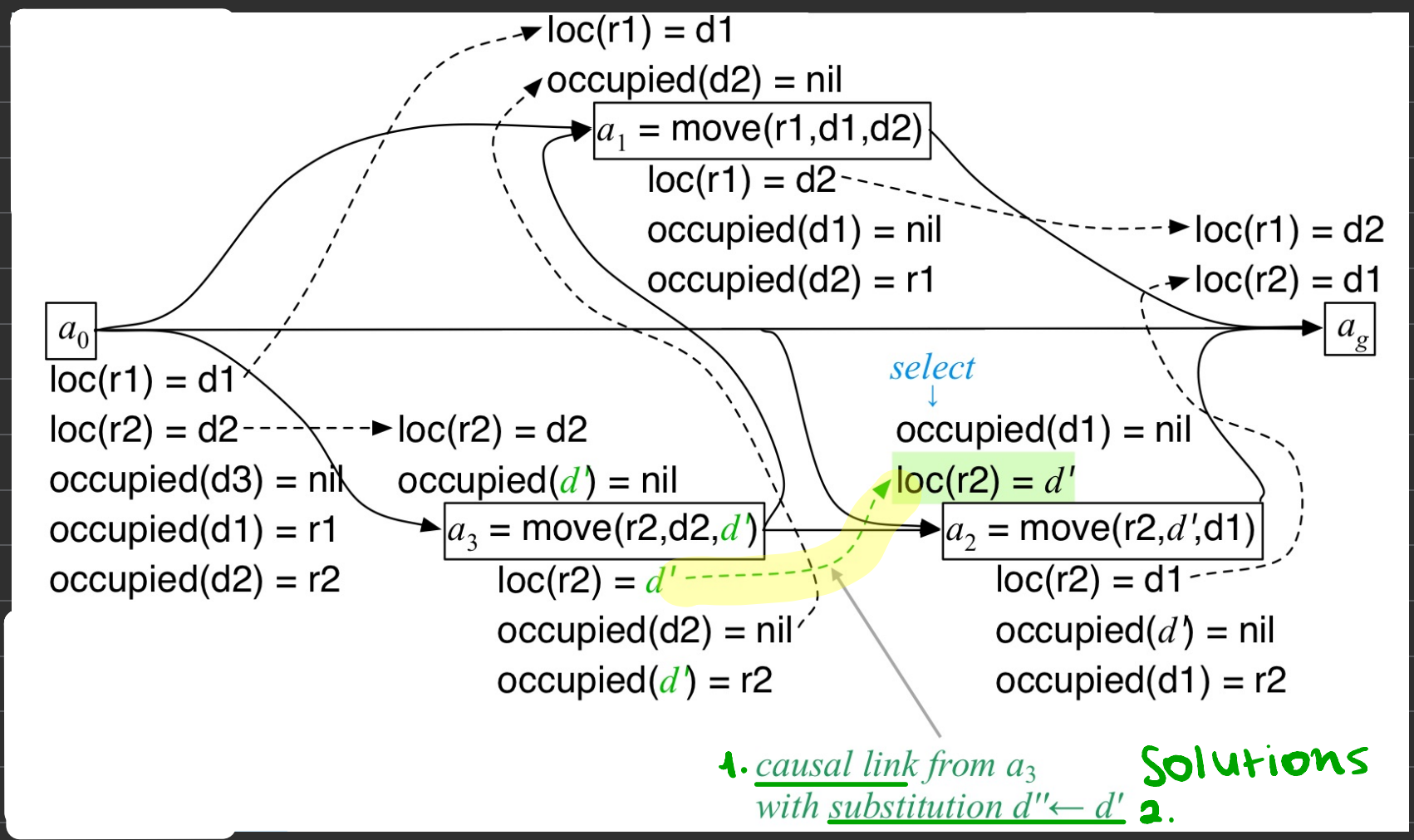
constraint : r ≠ r1



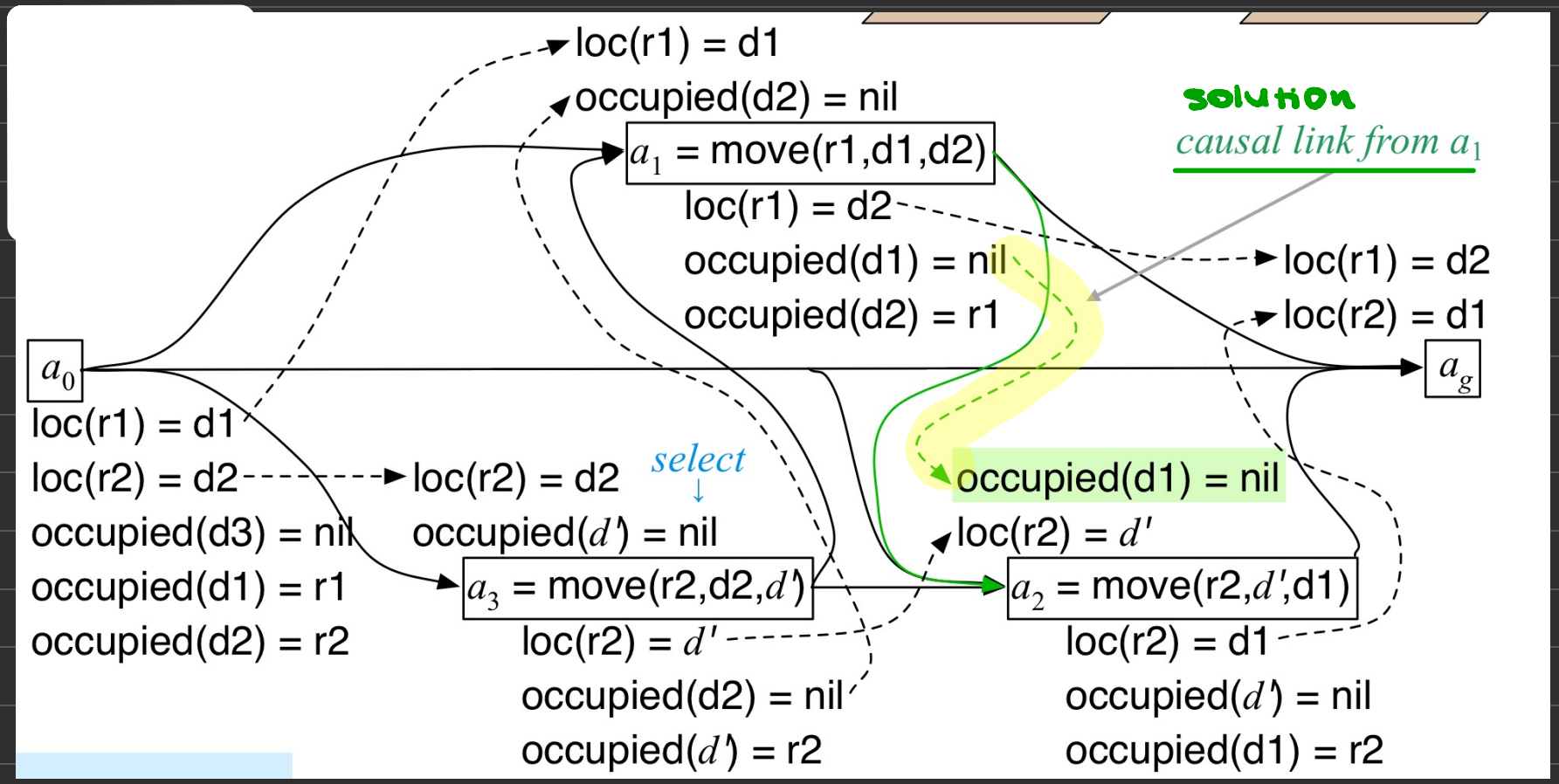
· 3 open goals
· no threats



· 2 open goals
· no threats

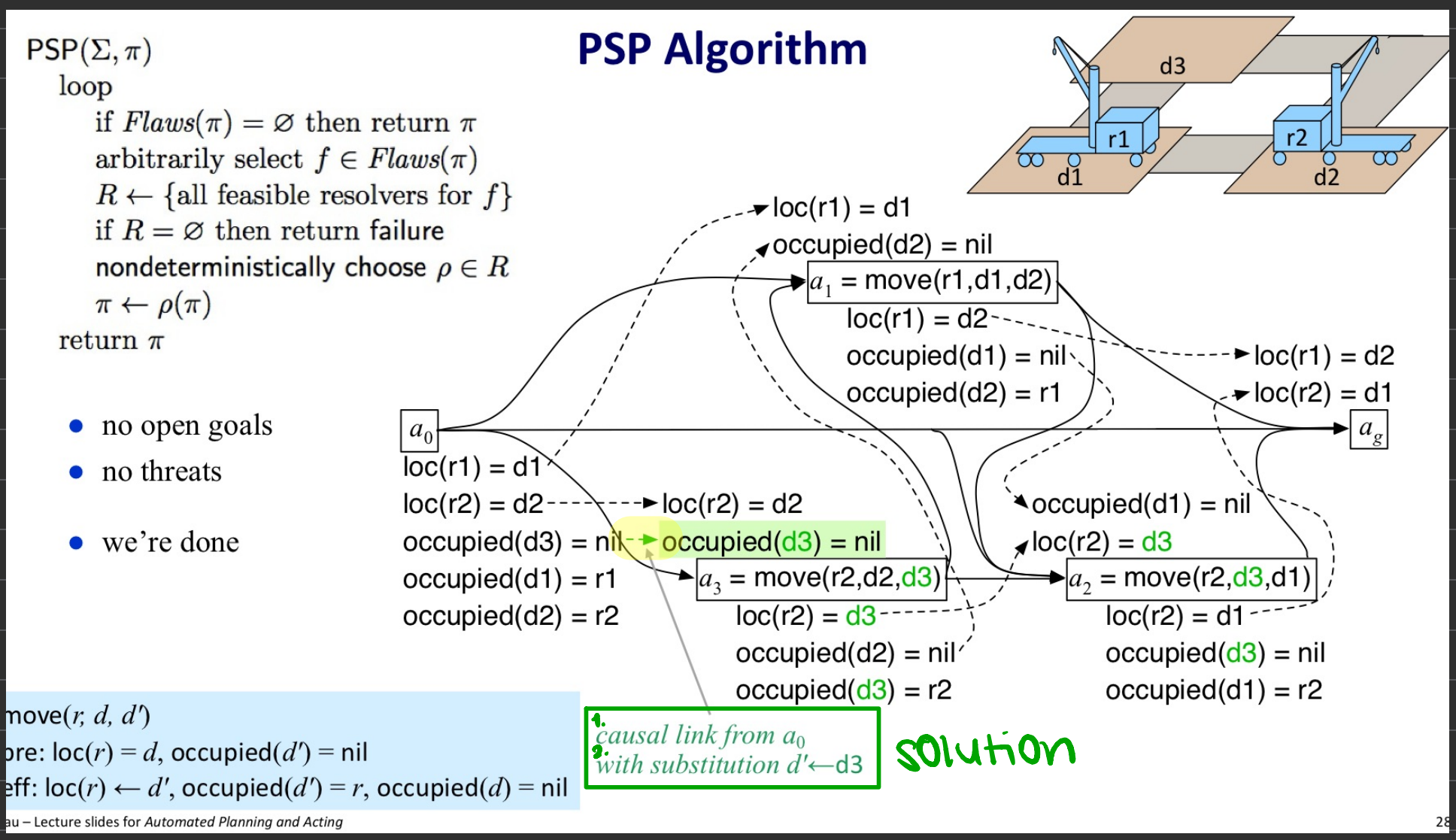


- 1 open goal
- no threats



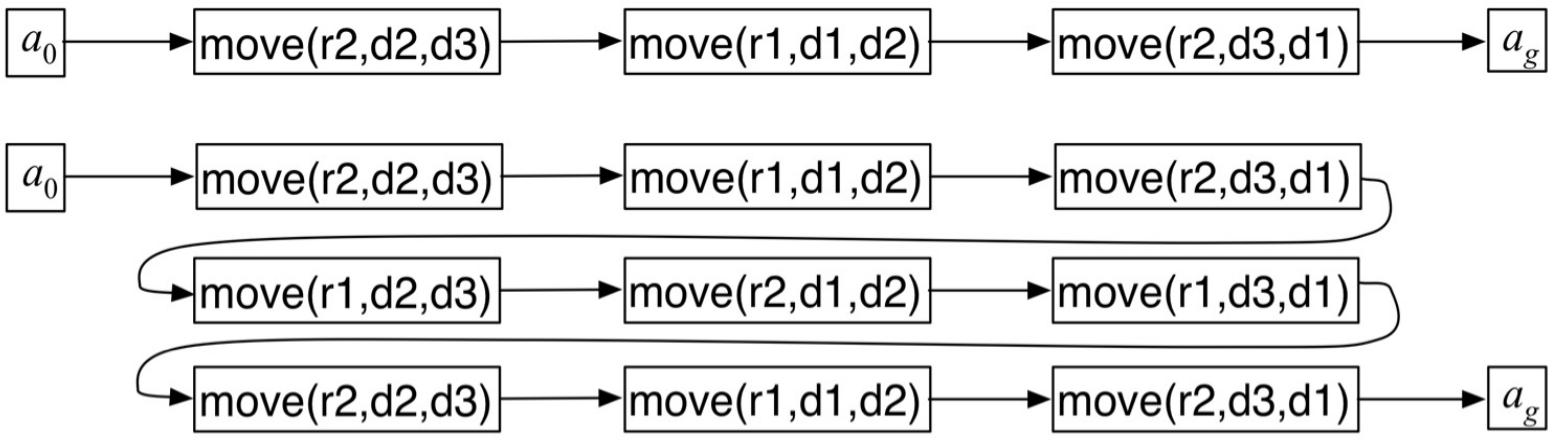
Last step!

- no open goals / threats

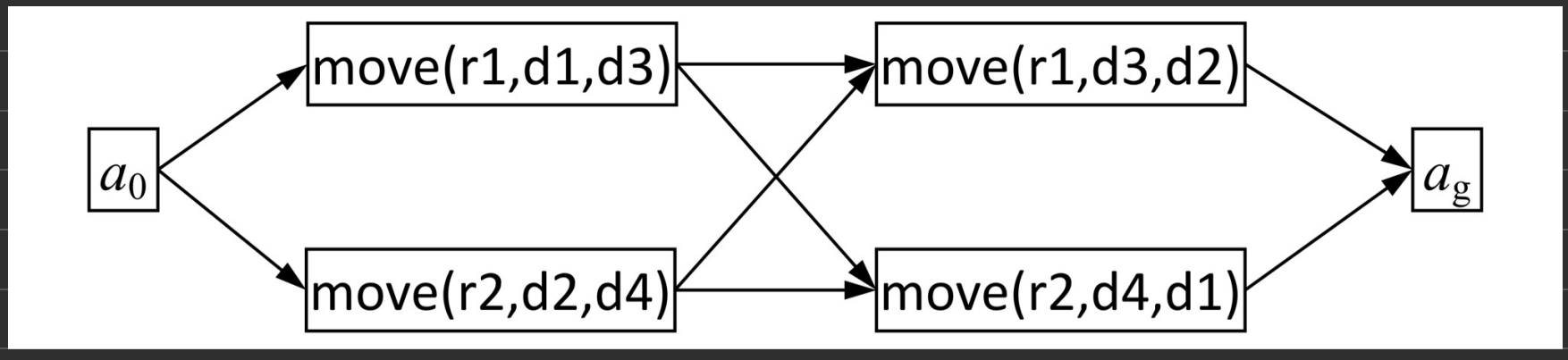


Solutions

- The solution we found:
- Another:
- Infinitely many others



- Add another location to the initial state
- This time, PSP also can return PARTIALLY-ORDERED solutions



selecting a flaw

selecting a flaw to resolve in PSP

≈

selecting a variable to instantiate in a CSP constraint satisfaction problem

↳ AND-branch in both cases

→ Fewest Alternatives First (FAF)

- select flaw w fewest resolvers

=> MRV (min remaining values) heuristic for CSPs.

choosing a resolver

chosing a resolver for a flaw

≈

assigning a value to a variable in CSP

↳ OR-branch in both cases

- Least Constraining Resolver (LCR)

→ prefer resolver that rules out the fewest resolvers for the other flows

- ~~LCV (Least Constraining Value) heuristic for CSPs.~~

problem in PSP

* keeps adding new actions forever

this might work ANA (Avoid New actions) heuristic

- prefer resolvers that don't add new actions

- use LCR as tie-breaker

BUT :

- Problem: ANA will prefer these two choices:
  - For loc(r1)=d in a_1, use action a_0 with substitution d←d1
  - For loc(r2)=d' in a_2, use action a_0 with substitution d'←d2
  - a_1 = move(r1,d1,d2); a_2 = move(r2,d2,d1)
    - Makes the problem unsolvable ⇒ need to backtrack
- Perhaps use ANA anyway?

Summary

- 2.5 Plan-Space Search
  - Definitions
    - Partially ordered plans and solutions
    - partial plans
    - causal links
  - flaws:
    - open goals
    - threats
  - resolvers
  - PSP algorithm
    - long example
    - brief discussion of node-selection heuristics, pruning techniques

chap 3

Planning & Acting

Planning

Acting

Prediction / Search

- Search over predicted states, possible organizations of tasks and actions
- Uses *descriptive* models (e.g., PDDL)
  - predict *what* the actions will do
  - don't include instructions for performing it

Performing

- Dynamic, unpredictable, partially observable environment
  - Adapt to context, react to events
- Uses *operational* models
  - instructions telling *how* to perform the tasks
  - usually hierarchical

REPRESENTATION

Tasks & Methods

Task -> activity for an actor to perform
taskname (arg1, ...)

foreach task → 1 or more: refinement methods

operational models telling how to perform the task

methodName (arguments)
- tasks
- preconditions
- body

method-name(arg_1, …, arg_k)
task: *task-identifier*
pre: *test*
body:
a program

- assignment statements
- control constructs:
  - if-then-else, while, … .
- tasks
  - can extend this to include events, goals
- commands to the execution platform

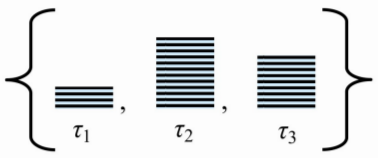
ACTING (RAE)

RAE [Refinement Acting Engine]

performs multiple tasks in parallel → reactive / no lookahead

for each task or event τ, a refinement stack → execution stack

- *Agenda* = {all current refinement stacks}



- Refinement stack for a task τ
  ⇔ current ~~path~~ in RAE's search tree for τ

refinement tree

procedure RAE:
loop:
for every new external task or event τ do
choose a method instance *m* for τ
create a refinement stack for τ, *m*
add the stack to *Agenda*
for each stack σ in *Agenda*
call Progress(σ)
if σ is finished then remove it

* Extensions to RAE
- Methods for events
  - e.g., an emergency
- Methods for goals
  - special kind of task: achieve(goal)
  - sets up a monitor to see if the goal has been achieved
- Concurrent subtasks

FULL explained example:
10 - 23 slides

Pages 9-10: Introduction and Problem Setup

- Environment: The problem involves two robots (r1, r2) in a partially observable environment with containers (c1, c2) and locations (loc0, loc1, loc2, loc3, loc4).
- Tasks: The primary task for the robots is to fetch containers.
- Commands: Robots can move, perceive their surroundings, take containers, and put them down.

Page 11: Fetch Task and Methods

- Tasks and Methods: Two methods for the fetch task are described:
- m-fetch1: Used when the container's location is unknown. The robot moves to unviewed locations and perceives them until it finds the container.
- m-fetch2: Used when the container's location is known. The robot directly moves to the location and takes the container.

Pages 12-13: Example Execution

- RAE Loop: The Refinement Acting Engine (RAE) processes tasks by selecting appropriate methods, creating refinement stacks, and progressing through these stacks.
- Example: The task fetch(r0, c2) is executed with method m-fetch1 as the container location is initially unknown. The robot perceives various locations to find the container.

Pages 14-15: Refinement Tree and Execution

- Refinement Tree: Shows the hierarchical structure of tasks and methods. The fetch task is decomposed into smaller actions like moving and perceiving.
- Execution: The robot executes the steps in m-fetch1, moving to locations and perceiving them until it finds the container and takes it.

Pages 16-17: Progressing and Handling Failures

- Progress Function: Manages the execution of tasks by checking if the current method step is a command, executing it, and updating the state.
- Handling Failures: If a method step fails (e.g., sensor failure), the RAE retries the task using a different method or a different refinement stack.

Pages 18-19: Example with Sensor Failure

- Sensor Failure Scenario: If the robot's sensor fails while perceiving a location, the task is retried with a different method or by re-executing the current method with different parameters.
- Progress Update: The state is updated accordingly, and the RAE continues processing until the task is successfully completed or all options are exhausted.

Pages 20-21: Simulation and Monte Carlo Rollouts

- Simulating Commands: To choose the best method instance, multiple simulated executions (Monte Carlo rollouts) are performed, and the method with the highest expected utility is selected.
- Expected Utility: Each method's expected utility is estimated based on the simulated outcomes, helping the RAE make informed decisions.

Page 22: Monte Carlo Rollout Details

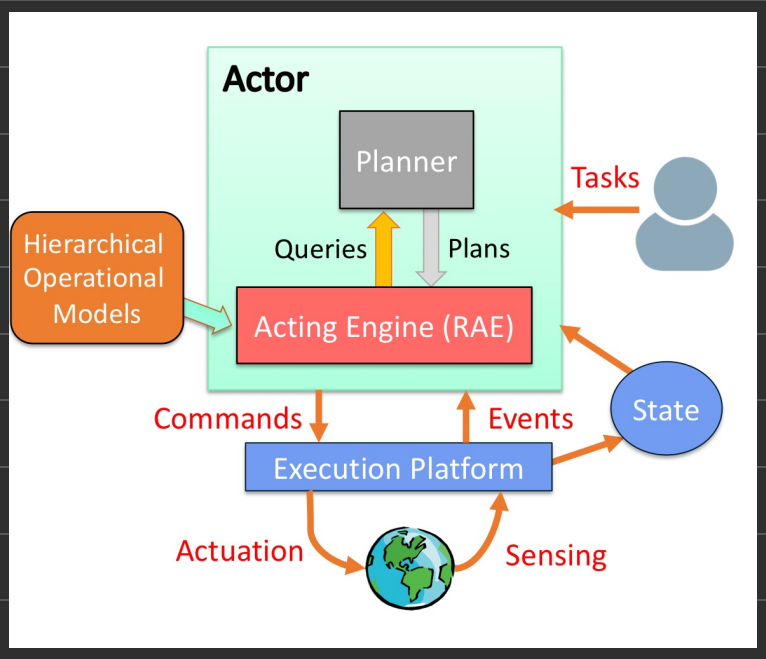
- Rollout Process: Each call to the simulation involves a sequence of actions and their outcomes. The process is repeated multiple times to gather statistical data.
- Multi-arm Bandit Problem: The exploration vs. exploitation dilemma is addressed using algorithms like UCB (Upper Confidence Bound) to balance between trying known successful actions and exploring new ones.

Page 23: Backtracking and Comparison to Search Algorithms

- Backtracking Search: The RAE's retry mechanism resembles a backtracking search, where failed attempts are revisited with alternative methods or parameters.
- Comparison: The approach is compared to classical search algorithms, highlighting the RAE's ability to dynamically adjust to failures and changing environments.

This summary provides a high-level overview of the algorithm example, focusing on the primary tasks, methods, and the process of executing and refining tasks in a dynamic environment.

Planning for RAE




```
procedure RAE:
    loop:
        for every new external task or event τ do
            choose a method instance m for τ
            create a refinement stack for τ, m
            add the stack to Agenda
        for each stack σ in Agenda
          call Progress(σ)
          if σ is finished then remove it
```

✓ 4 places where RAE & progress choose a method instance for a task

↓

☹ Bad choice may lead to:
- more costly solution
- failure [need to recover / unrecoverable]

☺ Solution:
- call a PLANNER, choose the method instance it suggests

Planning & Acting Integration

Τα action models των planners είναι abstractions

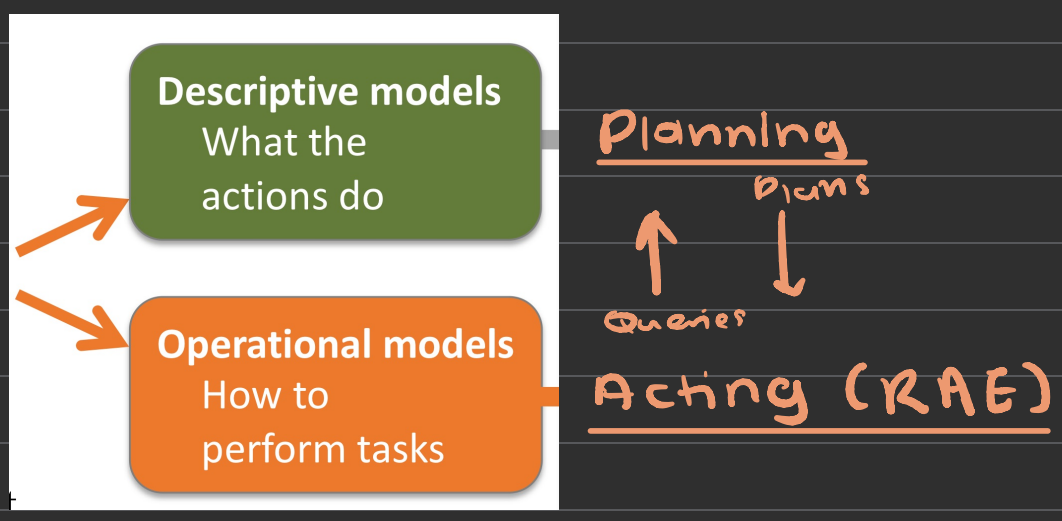
- The planned actions are tasks for the actor to refine

→ Consistency problem: how to get action models that describe what the actor will do?

→ Solution: ACTOR & PLANNER $\overset{\text{use}}{\Longrightarrow}$ SAME REPRESENTATION

- Must be operational; descriptive models too abstract
- Need planning algorithms that can use operational models

Consistent?



idea 1

- Planner uses Rae's tasks and refinement methods
- For each of Rae's commands, have a classical action model
- DFS or GBFS search among alternatives to see which works best

SeRPE (Sequential Refinement Planning Engine)

slide 29+ Problems slide 30, 31

1. Difficult to implement
2. Limitations of classical action models

Planning for RAE

```
procedure RAE:
    loop:
        for every new external task or event τ do
            choose a method instance m for τ
            create a refinement stack for τ, m
            add the stack to Agenda
        for each stack σ in Agenda
            call Progress(σ)
            if σ is finished then remove it
```

idea 2.

- Idea 2: simulation with multithreading or multiprocessing
  - Run Rae in simulated environment
    - Simulate the commands (see next page)
  - To choose among method instances, try all of them in parallel
- Planner returns the method instance *m* having the highest expected utility (≈ least expected cost)

Simulating commants

- simplest case:

Probabilistic action template

$a(x_1, \ldots, x_k)$

pre: …

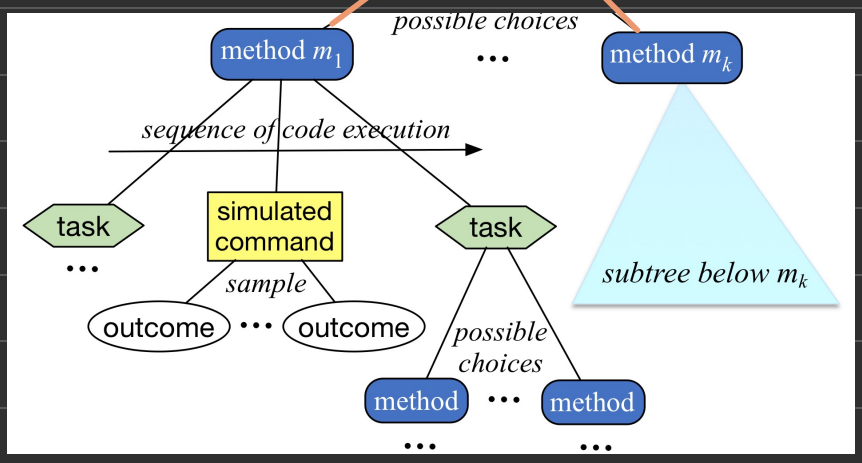
(p_1) effects$_1$: $e_{11}, e_{12}, \ldots$

…

(p_m) effects$_m$: $e_{m1}, e_{m2}, \ldots$

→ choose effects i at random with probability Pi and use it to update the current state

- More general:
  - Arbitrary computation, e.g., physics-based simulation
  - Run the code to get simulated effects



idea 3.

- Idea 3: simulation with Monte Carlo rollouts
  - Multiple runs
    - Random choices and outcomes in each run
  - Maintain statistics to estimate each choice's expected utility
  - Return the method instance *m* that has the highest estimated utility

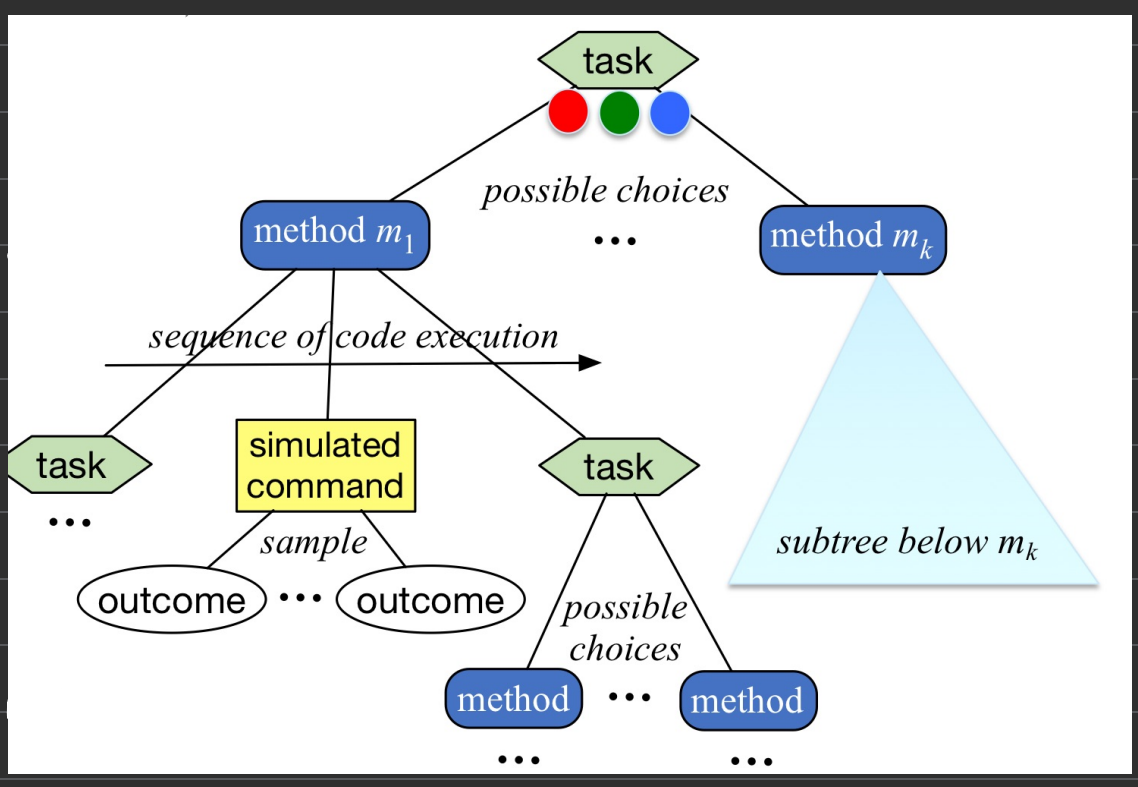
Planner

Plan-with-UPOM (task τ):
  Candidates ← {method instances relevant for τ}
  for i ← 1 to n
    call UPOM(τ)
    update estimates of methods' expected utility
  return the $m \in$ *Candidates* that has
    the highest estimated utility

UPOM(τ):
  choose a method instance m for τ
  create refinement stack σ for τ and m
  loop while Simulate-Progress(σ) ≠ *failure*
    if σ is completed then return (*m, utility*)
  return *failure*

→ each call to UPOM does a Monte Carlo rollout
  - simulated execution of RAE on τ

Monte Carlo Rollouts



RAE
Refinement Acting Engine

PlRAct Integr

idea 1: Planner uses RAE's task & refinem. methods
↓
for each RAE's commands
↓
classical action model

DFS & GBFS chose which works best → SeRPE
seq.
refin.
Plann.
engine

idea 2: Simulation ≠ multithreading
&
multiprocessing

Run RAE & simul. environm.
↓
simul. commands

To chose among methods → try them all in parallel having

Planner returns method highest exp. utility
= least expected cost

idea 3: simulation with Monte Carlo Rollouts
↓
multiple runs
↓
random choices & outcomes in each run
↓
statistics → to estimate each choice's expected utility
↓
return method instance that has highest estimated utility

Digression (απόκλιση): Monte Carlo Rollouts

- Exploitation vs exploration dilemma:
  - ***Exploitation***: choose an action that has given you high rewards in the past
  - ***Exploration***: choose an action that's less familiar, in hopes that it might produce a higher reward

UCB (Upper Confidence Bound) Algorithm

Assume all rewards are between 0 and 1

Για κάθε action a:

- $r(a)$ = average reward you've gotten from a
- $n(a)$ = number of times you've tried a
- $n_t = \sum_a n(a)$
- $Q(a) = r(a) + \sqrt{2(\ln n_t)/n(a)}$

UCB:

if there are any untried actions:
  $\tilde{a} \leftarrow$ any untried action
else:
  $\tilde{a} \leftarrow \text{argmax}_a\, Q(a)$
perform $\tilde{a}$
update $r(\tilde{a})$, $n(\tilde{a})$, n_t, $Q(\tilde{a})$

Theorem (given some assumptions)

As the number of calls to UCB → ∞
UCB's choice at each call → optimal choice

UCT Algorithm

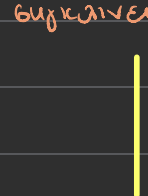
MDP: state space με όποιο κάθε action έχει probabilistic outcomes

UCT Algorithm: Monte Carlo Rollouts on a MDP

- At each state s,
  - Use UCB to choose an action at random
    - Balances exploration vs exploitation at s
  - Action's outcome ⇒ next state s

how to use UCT?
call it many times, return action with highest expected utility

Theorem:
as number of calls to UCT → ∞
choice converges to optimal



Convergence

UCT algorithm

- Monte Carlo rollouts on MDPs
- Call it many times, choice converges to optimal

UPOM search tree more complicated

- tasks, method instances, commands, code execution

if no exogenous events
→ can map it to UCT search of a complicated MDP
→ Proof of convergence to optimal

Acting with Planning (RAE + UPOM)

RAE + UPOM

```
procedure RAE:
    loop:
        for every new external task or event τ do
          * choose a method instance m for τ
            create a refinement stack for τ, m
            add the stack to Agenda
        for each stack σ in Agenda
          call Progress(σ)
          if σ is finished then remove it
```

whenever RAE needs to chose a method instance *
=> call **Plan-with-UPOM**
, use the method instance it returns

Can we use UPOM with Run-Lookahead / Run-Lazy-Lookahead?
slides 43-44-45

comparison

=>
- Rae + UPOM has tighter coupling between planning and acting
  - works better than Run-Lazy-Lookahead + UPOM'

Learning

Motivation

Plan-with-UPOM given as called acro to RAE, runs online

→ time constraints might not allow complete search

case 1 no time to search at all

- need a choice function

case 2 enough time to do partial search

- Receding horizon
  - Cut off search at depth d_{max} or when we run out of time
  - At leaf nodes, use heuristic function to estimated expected utility

Learning algorithms:

LearnΠ : learns a choice function
LearnH : learns a heuristic function

Integration with learning

- Gather training data from acting & planning traces of RAE and Plan-with-UPOM
- Train classifiers

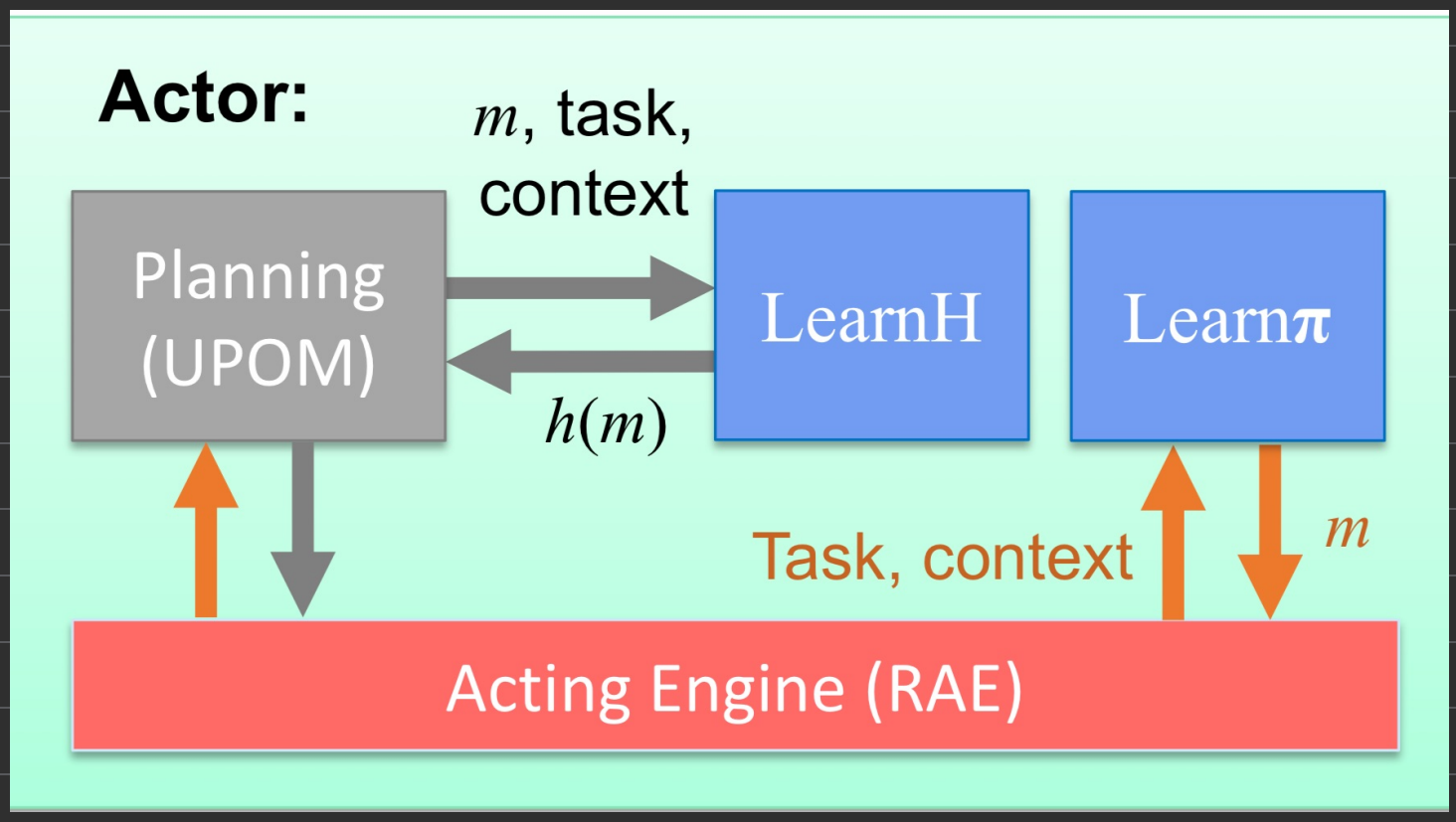
LearnH

Learns heuristic function to guide UPOM's search

↓

UPOM can use it to estimate expected utility at leaf nodes

=> useful if there isn't enough time to search all the way to the end



Evaluation & Application

Experimental Evaluation

Domain	$\|\mathcal{T}\|$	$\|\mathcal{M}\|$	$\|\overline{\mathcal{M}}\|$	$\|\mathcal{A}\|$	Dynamic events	Dead ends	Sensing	Robot collaboration	Concurrent tasks
S&R	8	16	16	14	✓	✓	✓	✓	✓
Explore	9	17	17	14	✓	✓	✓	✓	✓
Fetch	7	10	10	9	✓	✓	✓	–	✓
Nav	6	9	15	10	✓	–	✓	✓	✓
Deliver	6	6	50	9	✓	✓	–	✓	✓

- Five different domains, different combinations of characteristics
- Evaluation criteria: efficiency (reciprocal of cost), successes vs failures
- Result: Planning and learning help
  - RAE operates better with UPOM or learning than without
  - RAE's performance improves with more planning

Prototype Application

Software-defined networks

- Decoupled control and data layers
- Prone to high-volume, fast-paced online attacks
- Need automated attack recovery

Prototype Solution using RAE & UPOM

- expert writes recovery procedures as refinements methods

Experimental Results

- Improved efficiency, retry ratio, sucess ratio, resilience compared to human expert

Summary

- 3.1 Operational models
  - ξ versus s, tasks, events,
  - Commands to the execution platform
  - Extensions to state-variable representation
  - Refinement method
    - name, task/event, preconditions, body
  - Example: fetch a container
- 3.2 Refinement Acting Engine (RAE)
  - Purely reactive: select a method and apply it
  - Rae: input stream, *Candidates*, Instances, *Agenda*, refinement stacks
  - Progress:
    - command status, nextstep, type of step
  - Retry: *Candidates* \ *tried*
    - comparison to backtracking
  - Refinement trees
- 3.3 Refinement planning
  - plan by simulating Rae on a single external task/event/goal
  - SeRPE uses classical action models
  - UPOM simulates the actor's commands, does Monte Carlo rollouts
- 3.4 Acting and planning
  - Rae + UPOM
  - Comparison: Run-Lazy-Lookahead + UPOM′
  - A little about learning, experimental evaluation, prototype application
- Open-source Python implementation of Rae and UPOM:
  - https://bitbucket.org/sunandita/RAE/

chap 3 (HTN)

Refinement based Planning

idea of refining a task into smaller subtusks also used for planning

HTN -> hierarchical task network planning:
process of providing "recipes" for breaking down / refining complex tasks into smaller ones

- 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.

HTN - Hierarchical Task Network - based planning

For many planning problems -> we have ideas of how to find solutions: eg slide 3

- 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 & actions
- tasks (activities)
- HTN methods (ways to refine tasks)

format of a method:

method-name (*args*)
Tarea: *task-name*(*args*)
Prec: *preconditions*
Sub: *list of subtasks*

2 types of Subtasks

primitives: name of an action

composed: needs to be broken down [= refined] using **methods**

- HTN planning domain → 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

we apply methods for composed tasks
we apply 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

eg. 5-6-7

Integration of hierarchical planning & acting

For planning & acting ⟶ necessary that HTN methods can recover from unexpected problems

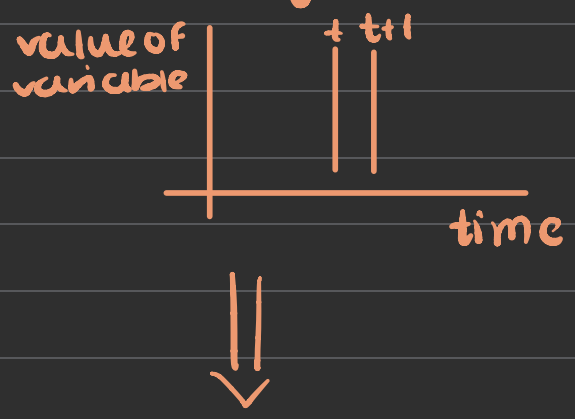
chap4

Notes from video

. Restrictions of duration in the actions

. We work with timelines

↳ they tell us how the value of the variable changes over time



Time line : a pair (T, C)

- T — temporal assertions
  - change
  - persistence
- C — Constraints

temporal assertions are the effects of the actions

notes from Proffesor

* specific parts of the algorithm in the exam:
  - working with a timeline to check if it is consistent or secure ✓
  - to provide casual support to temporal assertions at a given timeline ✓
  - Primitive actions & refinement method representation in temporal planning
  - Asks us to define a temporal action or method
    ↳ we should know how to define it (setting its temporal assertions, restrictions, etc)

PROPERTIES

Consistency

* Timeline is consistent if the 'temporal Assertions' → T satisfy all the 'constraints' → C that have been set &

no state variable has more than one value at a specific instant of time

- (T, C) is *consistent* if it has *at least one* consistent ground instance

↓

(T', C') is consistent if *

Security

*Timeline (T, C) is secure if:

- ✓ it is consistent (atleast 1 consistent ground instance)
- ✓ every gr. instance that satisfies the constraints is consistent

* how to make a timeline secure?

✓ by adding more constraints

↓

separation constraints!

Logico!!

Union of Multiple Timelines

- Timelines for k different state variables, all of which are fully ground: ← book omits this part
  - $(T_1, C_1), \ldots, (T_k, C_k)$
- Union is (T, C):
  - $T = T_1 \cup \ldots \cup T_k$
  - $C = C_1 \cup \ldots \cup C_k$
- If every (T_i, C_i) is secure, then $(T_1, C_1) \cup \ldots \cup (T_k, C_k)$ is also secure

casual support

what ensures the correct starting value?

action on the temporal assertion

↓

gives casual support to another temporal assertion

or

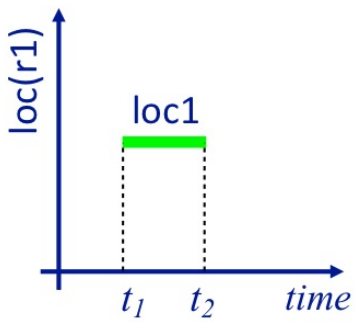
it is part of the initial state (priori casual support)

Causal support

- Consider the assertion $[t_1,t_2]$ loc(r1) = loc1
  - How did r1 get to loc1 in the first place?

Temporal Assertions

- Let α be either $[t_1,t_2]\ x = v_1$ or $[t_1,t_2]\ x : (v_1, v_2)$
  persistent change
- *Causal support* for α
  - Information saying α is supported *a priori*
  - Or another assertion that produces $x = v_1$ at time t_1
    - $[t_0,t_1]\ x = v_1$
    - $[t_0,t_1]\ x : (v_0, v_1)$



- A timeline $\mathcal{T}$ is *causally supported* if every assertion α in $\mathcal{T}$ has a causal support

- Three ways to modify a timeline to add causal support …

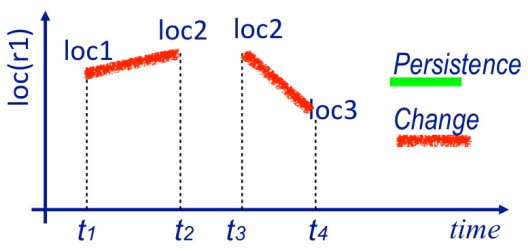
3 ways to modify timeline to add casual support

① Add a persistence assertion

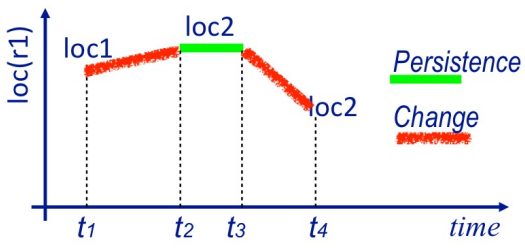
(1) Add a persistence assertion

$\mathcal{T}$ = {$[t_1,t_2]$ loc(r1):(loc1,loc2),
$[t_3,t_4]$ loc(r1):(loc2,loc3)}

C = {$t_1 < t_2 < t_3 < t_4$}



- Add $[t_2,t_3]$ loc(r1) = loc2
  - Supported by the first temporal assertion
  - Supports the second one

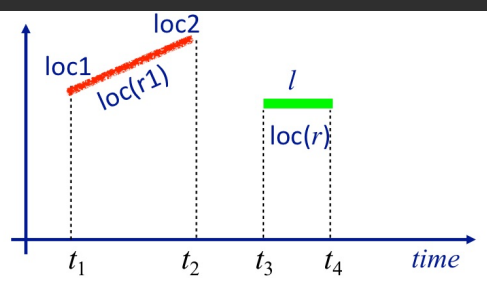


② Add constraints

(2) Add constraints

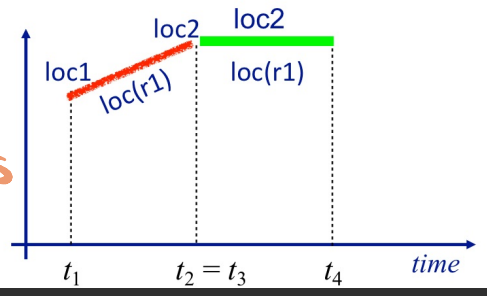
$\mathcal{T}$ = {$[t_1,t_2]$ loc(r1):(loc1,loc2),
$[t_3,t_4]$ loc(r) = l}

C = {$t_1 < t_2$, $t_3 < t_4$}



- Add $t_2 = t_3$, r = r1, l = loc2

+ assigning specific values to variables

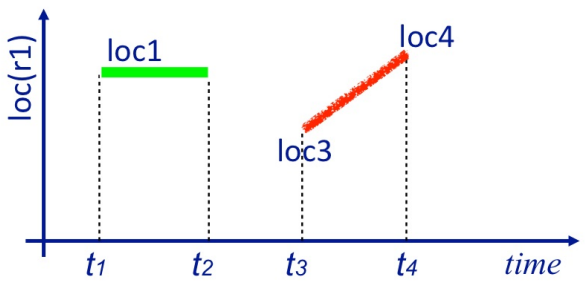


③ Add a change assertion (by adding an action)

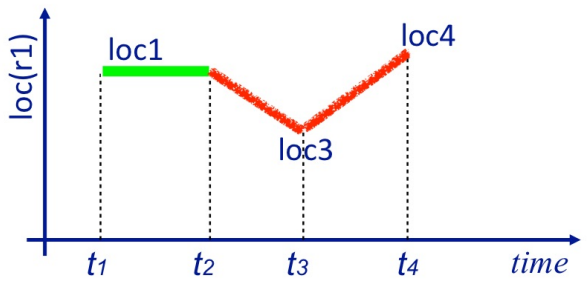
(3) Add a change assertion
(by adding an action)

$\mathcal{T}$ = {[t_1,t_2] loc(r1) = loc1,
[t_3,t_4] loc(r1):(loc3,loc4)}

C = {$t_1 < t_2 < t_3 < t_4$}



- Add an action that includes
[t_2,t_3] loc(r1):(loc1,loc3)



⇓

the casual support is the essence of the algorithm (later discussed)

How the actions are defined?
How do they include temporal assertions?

Temporal Planning → Planning & Refinement

Actions
/ \
Primitive — can be resolved through refinement methods (subdividing them into smaller actions)

↓

Primitive TASKS (actions)

(head, T, C)

head → name, parameters

T, C → (T, C) union of set of timelines

parameteres → ts, te

- Always two additional parameters
  - starting time t_s , ending time t_e
- In each temporal assertion in $\mathcal{T}$,
  - left endpoint is like a precondition
    - ⇔ need for causal support
  - right endpoint is like an effect

T : prec effect

temp. assert

Refinement Methods

Methods & Tasks

how is the tasks refined into primitive actions or other subtasks

Tasks and Methods

- Task: move robot r to dock d
  - $[t_s, t_e]$ move(r,d)

adds additional parameters (objects) that are going to appear somewhere in the refinement

- Method:

m-move1(r,d,d',w,w')

task: move(r,d)

refinement:

$[t_s, t_1]$ leave(r,d',w')
$[t_2, t_3]$ navigate(r,w',w)
$[t_4, t_e]$ enter(r,d,w)

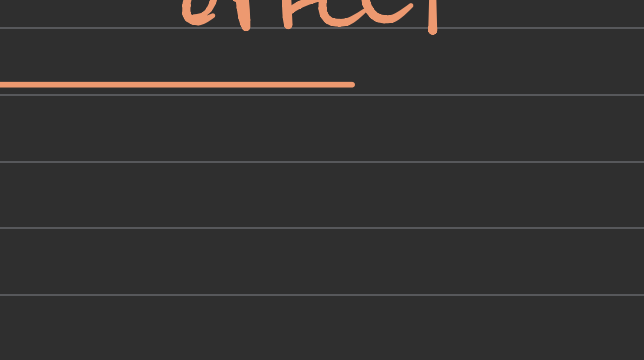
book omits r

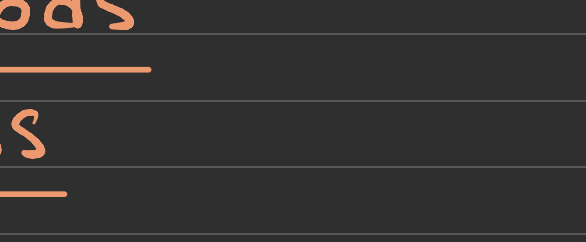
assertions:

$[t_s, t_s+1]$ loc(r) = d'

constraints:

adjacent(d,w),
adjacent(d',w'), $d \neq d'$,
connected(w,w'),
$t_1 \leq t_2$, $t_3 \leq t_4$





+ + +

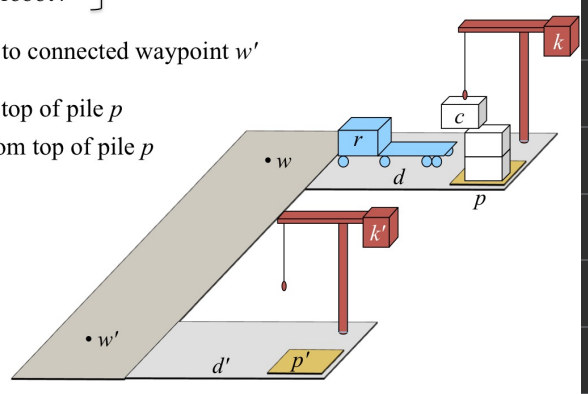
apa στην ουσία έχουμε

primitive actions

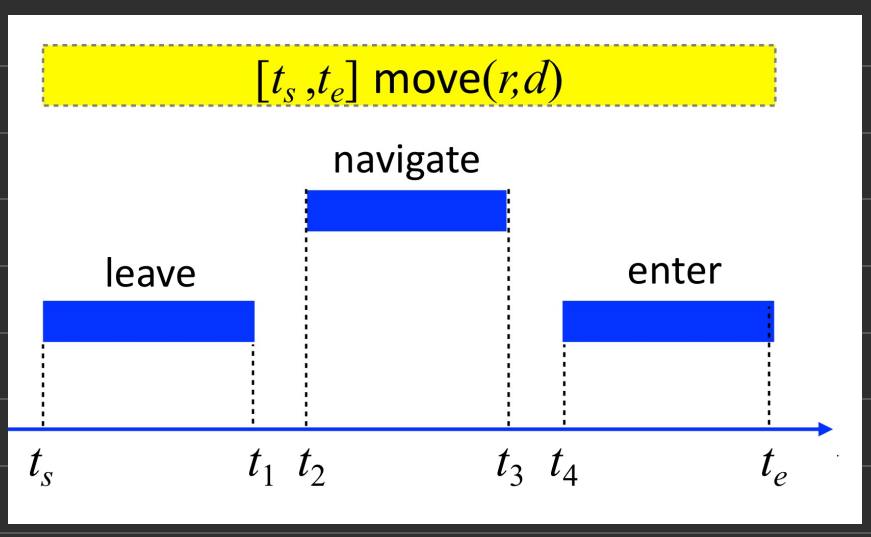
αυτα:

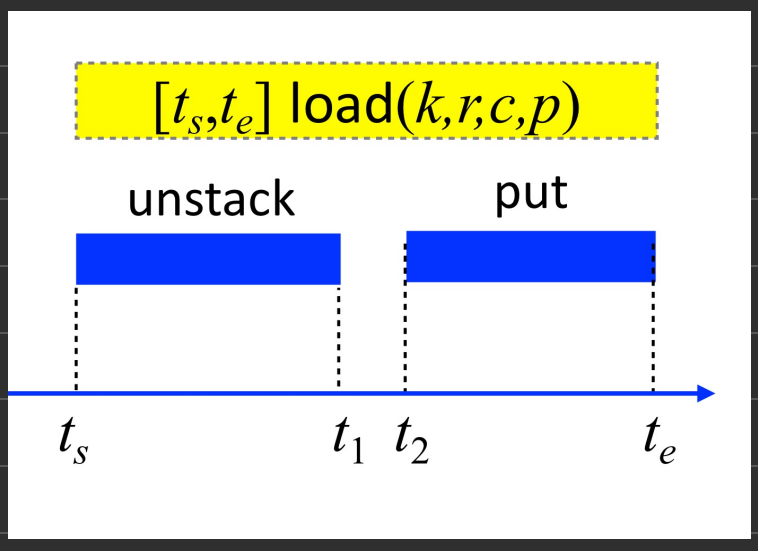
- leave(r,d,w) robot r leaves dock d to an adjacent waypoint w
- enter(r,d,w) r enters d from an adjacent waypoint w
- take(k,c,r) crane k takes container c from robot r
- put(k,c,r) crane k puts container c onto robot r
- navigate(r,w,w') r navigates from waypoint w to connected waypoint w'
- stack(k,c,p) crane k stacks container c on top of pile p
- unstack(k,c,p) crane k takes a container c from top of pile p

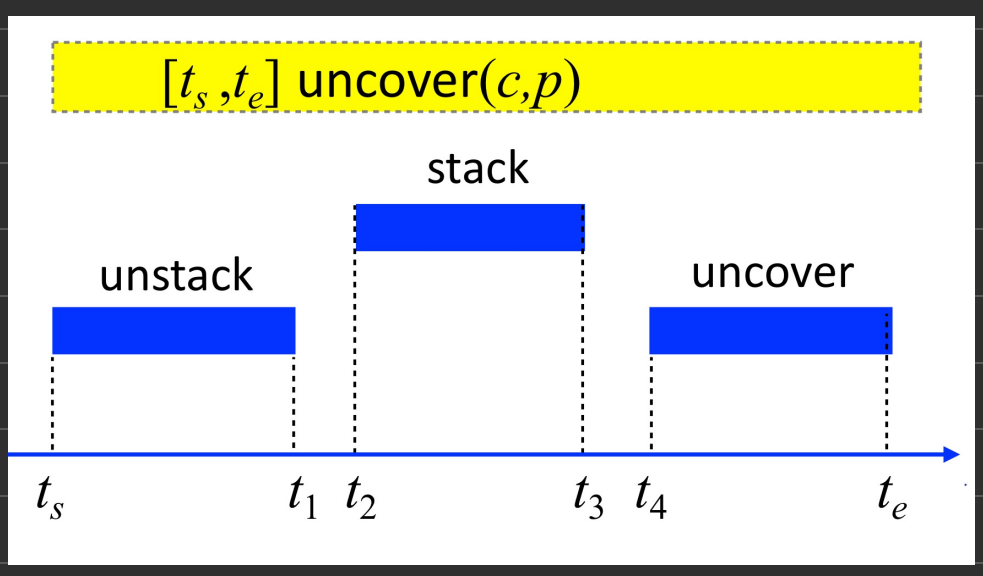
c, c' - containers
d, d' - loading docks
k, k' - cranes
p, p' - piles
r - robot
w, w' - waypoints

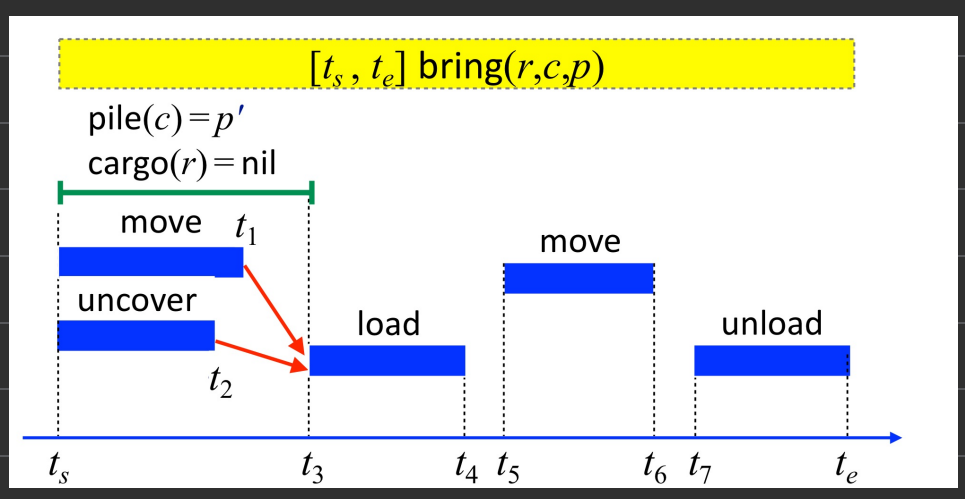


και methods που τα χρησιμοποιούν









Summed up

primitive actions (head, T, C)

head → enter (r, d, w)

T → assertions:
[ts, te] loc(r) : (w, d)
[ts, te] occupand : (empty, r)

C → constraints:
$te \leq ts + \delta_2$
adjacent(d, w)

methods

what includes

m-move (r, d, d', w, w')

task move(r, d)

refinement [list of primitive actions may take time]
[ts, t1] leave (r, d', w') ...

assertions [ts, ts+1] loc(r) = d' // kati gia current state C?)

constraints adjacent(d, w)
$t1 \leq t2$, $t3 \leq t4$

chronicles

$\varphi = (A, S, T, C)$

- $\mathcal{A}$: temporally qualified tasks
- S: *a priori* supported assertions
- $\mathcal{T}$: temporally qualified assertions
- C: constraints

"a problem to be solved"

constraint management

slide 53

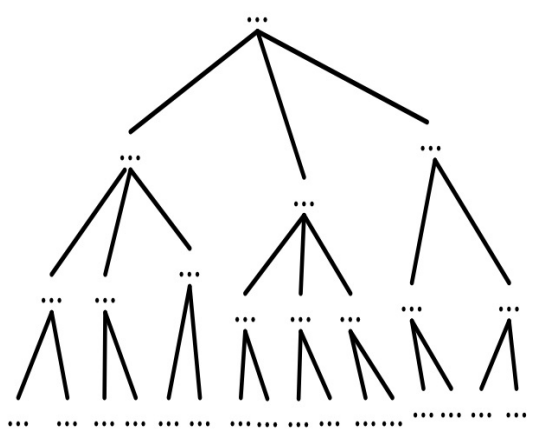
Consistency of C

- C contains two kinds of constraints
  - Object constraints
    - $\text{loc}(r) \neq l_2, \quad l \in \{\text{loc3}, \text{loc4}\}, \quad r = \text{r1}, \quad o \neq o'$
  - Temporal constraints
    - $t_1 < t_3, \quad a < t, \quad t < t', \quad a \leq t' - t \leq b$
- Assume object constraints are independent of temporal constraints and vice versa
  - exclude things like $t <$ distance (r, r')
- Then two separate subproblems
  - (1) check consistency of object constraints
  - (2) check consistency of temporal constraints
  - C is consistent iff both are consistent

(1)

Object Constraints

- Constraint-satisfaction problem (CSP) – NP-hard
- Can write an algorithm that's *complete* but runs in exponential time
  - If there's an inconsistency, always finds it
  - Might do a lot of pruning, but spend lots of time at each node
- Instead, use a technique that's incomplete but takes *polynomial* time
  - arc consistency, path consistency*
- Detects some inconsistencies but not others
  - Runs much faster, but prunes fewer nodes



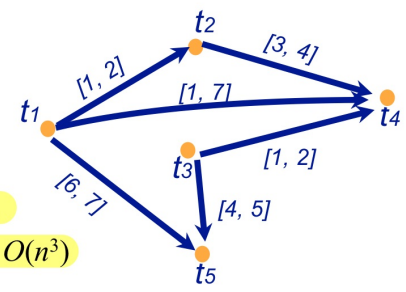
*See Russell & Norvig, Artificial Intelligence: A Modern Approach

(2)

Time Constraints

To represent time constraints:

- Simple Temporal Networks (STNs)
  - Networks of constraints on time points
- Synthesize incrementally them starting from ϕ_0
  - Templan can check time constraints in time $O(n^3)$
- Incrementally instantiated at acting time
- Kept consistent throughout planning and acting



STN: (V, E)

- $E \subseteq V^2$ set of arcs
- V: set of temporal variables $t_1, \ldots, t_n$

> each arch $[t_i, t_j] \rightarrow$ interval $[a, b]$ ↓ represents a constraint

$t_j - t_i \in [a, b]$

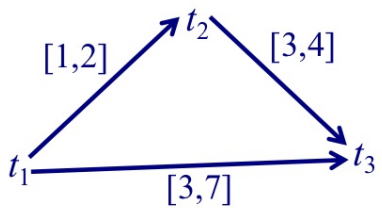
$t_i - t_j \in [-b, -a]$

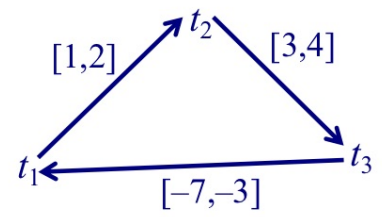
> unary constraints: $t_0 \equiv 0$

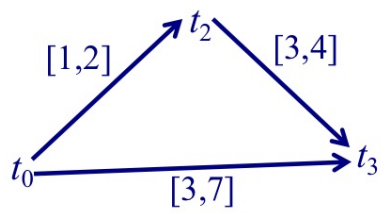
$r_{oi} = [a, b] \iff t_i - 0 \in [a, b]$

Time Constraints

- *Simple Temporal Network* (STN):
- a pair $(\mathcal{V},\mathcal{E})$, where
  - $\mathcal{V}$ = {a set of temporal variables $\{t_1, \ldots, t_n\}$}
  - $\mathcal{E} \subseteq \mathcal{V}^2$ is a set of arcs
- Each arc (t_i,t_j) is labeled with an interval $[a,b]$
  - Represents constraint $t_j - t_i \in [a, b]$
  - Or equivalently, $t_i - t_j \in [-b, -a]$
- Notation: instead of $t_j - t_i \in [a, b]$, write $r_{ij} = [a,b]$
- To represent unary constraints:
  - Dummy variable $t_0 \equiv 0$
  - Arc $r_{0i} = [a,b]$ represents $t_i - 0 \in [a,b]$







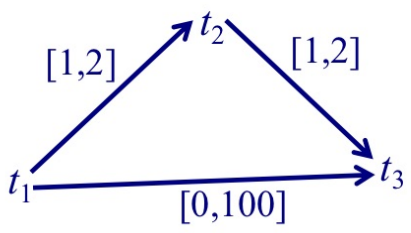
Solution to an STN: να βρώ integer values στα timepoints έτσι ώστε όλα τα constraints να ικανοποιούνται

↓

consistent STN: has a solution ✓

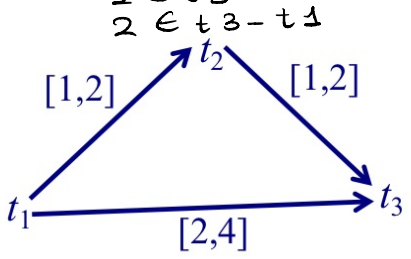
Time Constraints

- *Solution* to an STN:
  - any assignment of integer values to the time points such that all the constraints are satisfied
- *Consistent* STN: has a solution



- Solutions:
  - $(t_2–t_1, t_3–t_2, t_3–t_1) \in$ {(1,1,2), (1,2,3), (2,1,3), (2,2,4)}

↳ $1 \in t_2 - t_1$
$1 \in t_3 - t_2$
$2 \in t_3 - t_1$



- *Minimal* STN:

  for every arc (t_i,t_j) with label $[a,b]$,

  for every $t \in [a,b]$,

  there's at least one solution such that $t_j - t_i = t$

  - If we make any of the time intervals shorter, we'll exclude some solutions

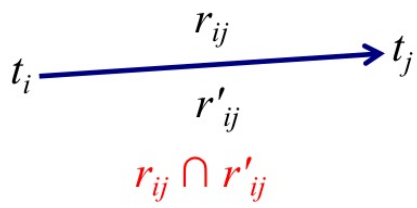
Operations on STNs

- Intersection, $\cap$

  $t_j - t_i \in r_{ij} = [a_{ij}, b_{ij}]$

  $t_j - t_i \in r'_{ij} = [a'_{ij}, b'_{ij}]$

  Infer $t_j - t_i \in r_{ij} \cap r'_{ij} = [\max(a_{ij}, a'_{ij}), \min(b_{ij}, b'_{ij})]$



$r_{ij} \cap r'_{ij}$

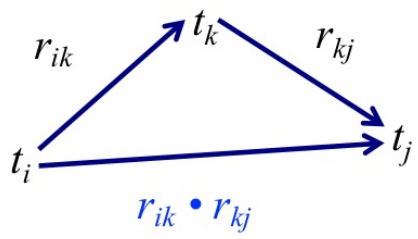
- Composition, $\bullet$

  $t_k - t_i \in r_{ik} = [a_{ik}, b_{ik}]$

  $t_j - t_k \in r_{kj} = [a_{kj}, b_{kj}]$

  Infer $t_j - t_i \in r_{ik} \bullet r_{kj} = [a_{ik} + a_{kj},\ b_{ik} + b_{kj}]$

  Reason: shortest and longest times for the two intervals

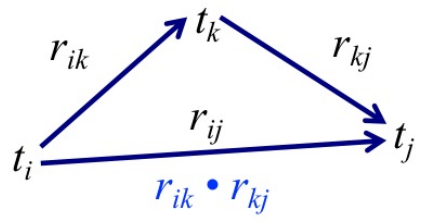


$r_{ik} \bullet r_{kj}$

- Consistency checking

  not empty

  - r_{ik}, r_{kj}, r_{ij} are consistent only if $r_{ij} \cap (r_{ik} \bullet r_{kj}) \neq \emptyset$
- Special case for networks with just three nodes
  - Consistent iff if $r_{ij} \cap (r_{ik} \bullet r_{kj}) \neq \emptyset$



$r_{ij} \cap (r_{ik} \bullet r_{kj})$

$r_{ij} \cap (r_{ik} \bullet r_{kj}) \neq 0$

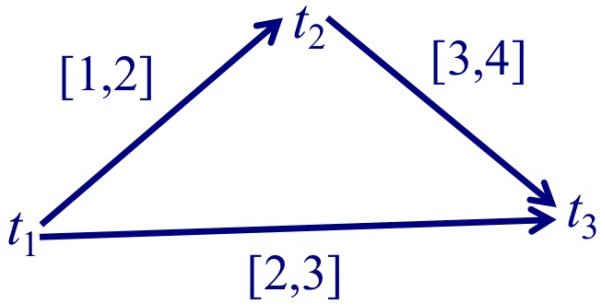
↓

consistency checking

- intersection → finds common interval between 2 constraints

- composition → combines 2 constraints through a variable to form a direct constraint

examples:



- STN $(\mathcal{V},\mathcal{E})$, where
  - $\mathcal{V} = \{t_1, t_2, t_3\}$
  - $\mathcal{E} = \{r_{12}=[1,2],\ r_{23}=[3,4],\ r_{13}=[2,3]\}$
- Composition:
  - $r'_{13} = r_{12} \bullet r_{23} = [4,6]$
- Can't satisfy both r_{13} and r'_{13}
  - $r_{13} \cap r'_{13} = [2,3] \cap [4,6] = \emptyset$
- $(\mathcal{V},\mathcal{E})$ is inconsistent

do not intersect!

$V = \{t_1, t_2, t_3\}$

$E = \{r_{12} = [1,2]$

$r_{23} = [3,4]$

$r_{13} = [2,5]\}$

composition:

$r_{13}' = [1+3, 2+4]$

$= [4, 6]$

$r_{13}' \cap r_{13} =$

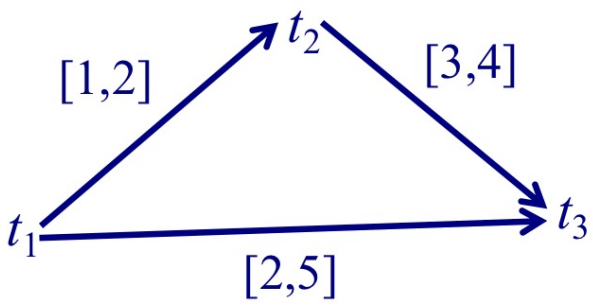
$[4,6] \cap [2,5] =$

$[4,5] \neq \emptyset$

max min

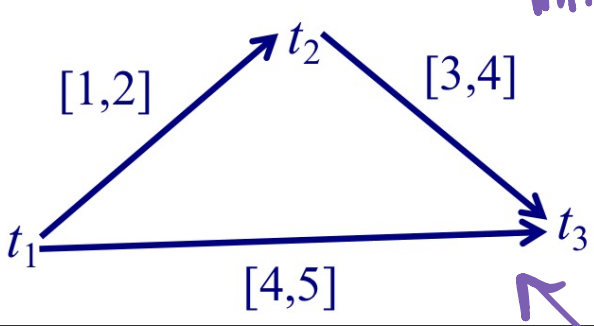
$\Rightarrow (V,E)$ consistent

$r_{13} \leftarrow [4,5]$ will make it minimal



- STN $(\mathcal{V},\mathcal{E})$, where
  - $\mathcal{V} = \{t_1, t_2, t_3\}$
  - $\mathcal{E} = \{r_{12}=[1,2], r_{23}=[3,4], r_{13}=[2,5]\}$
- As before, $r'_{13} = [4,6]$
  - $r_{13} \cap r'_{13} = [4,5]$
- $(\mathcal{V},\mathcal{E})$ is consistent
  - $r_{13} \leftarrow [4,5]$ will make it minimal
  - Same solutions as above

minimal



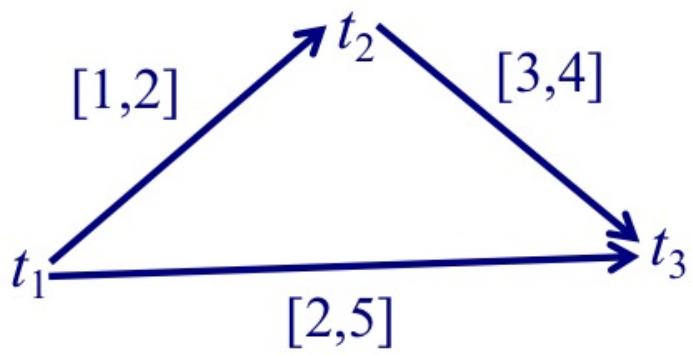
PATH CONSISTENCY ALGORITHM (PC)

- PC (*Path Consistency*) algorithm:
  - For every $i < j$, iterates over r_{ij} once for each k
  - Each time, tries to reduce interval, checks consistency
  - i, j, k each go from 1 to n

    => time $O(n^3)$

$O(n^3)$

Operations on STNs



- For $n = 3$, PC tries these triples:
  - k=1, i=2, j=3
  - k=2, i=1, j=3
  - k=3, i=1, j=2

- k=1, i=2, j=3
  - $r_{ik} = r_{21} = [-2,-1]$
  - $r_{kj} = r_{13} = [2,5]$
  - $r_{ij} = r_{23} = [3,4]$
  - $r_{ik} \bullet r_{kj} = [-2+2, -1+5] = [0,4]$
  - $r_{23} = r_{ij} \leftarrow [\max(3,0), \min(4,4)]$
    $= [3,4]$
  - No change

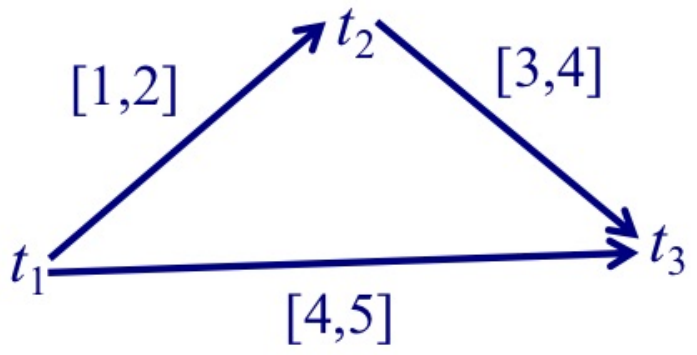
$[0,4] \cap [3,4] = [3,4]$

- k=2, i=1, j=3 (see previous slide)
  - PC reduces $r_{ij} = r_{13}$ to [4,5]

- k=3, i=1, j=2
  - $r_{ik} = r_{13} = [4,5]$
  - $r_{kj} = r_{32} = [-4,-3]$
  - $r_{ij} = r_{12} = [1,2]$
  - $r_{ik} \bullet r_{kj} = [4-4, 5-3] = [0,2]$
  - $r_{13} = r_{ij} \leftarrow [\max(1,-2), \min(2,2)]$
    $= [1,2]$
  - No change

$[0,2] \cap [1,2] = [1,2]$

- Minimal network:



- The PC algorithm iteratively refines the constraints in the network by considering all possible paths.
- The process ensures that all constraints are consistent with each other, resulting in a minimal network where all intervals are as tight as possible without introducing inconsistency.
- This helps in verifying whether a given STN is consistent and if not, adjusting it to achieve consistency.