

PowerPoint Presentation	3
PowerPoint Presentation	24
PowerPoint Presentation	44
PowerPoint Presentation	55
PowerPoint Presentation	82
PowerPoint Presentation	105
A.I. PLANNING	120
A.I. PLANNING	135
A.I. PLANNING	154
A.I. PLANNING	166
PowerPoint Presentation	176
PowerPoint Presentation	198
PowerPoint Presentation	217
PowerPoint Presentation	222
PowerPoint Presentation	243
Temporal Planning I - 01 - Introduction and Modelling	263
Temporal Planning I - 02 - Planning with PDDL 2.1 Challenges and Decision Epoch Planning	274
Temporal Planning I - 03 - Simple Temporal Networks and Crikey 3	282
Temporal Planning I - 04 - Heuristics for Temporal Planning - TRPG	298
RPG	304
Temporal Planning II - 01 - POPF Introduction and Overview	305
Temporal Planning II - 02 - POPF Details and Worked Example	312
Temporal Planning II - 03 - Compression Safety	327
PowerPoint Presentation	339
PowerPoint Presentation	351
PowerPoint Presentation	360
PowerPoint Presentation	379
PowerPoint Presentation	391
PowerPoint Presentation	406
PowerPoint Presentation	432

PowerPoint Presentation	453
PowerPoint Presentation	465
PowerPoint Presentation	483

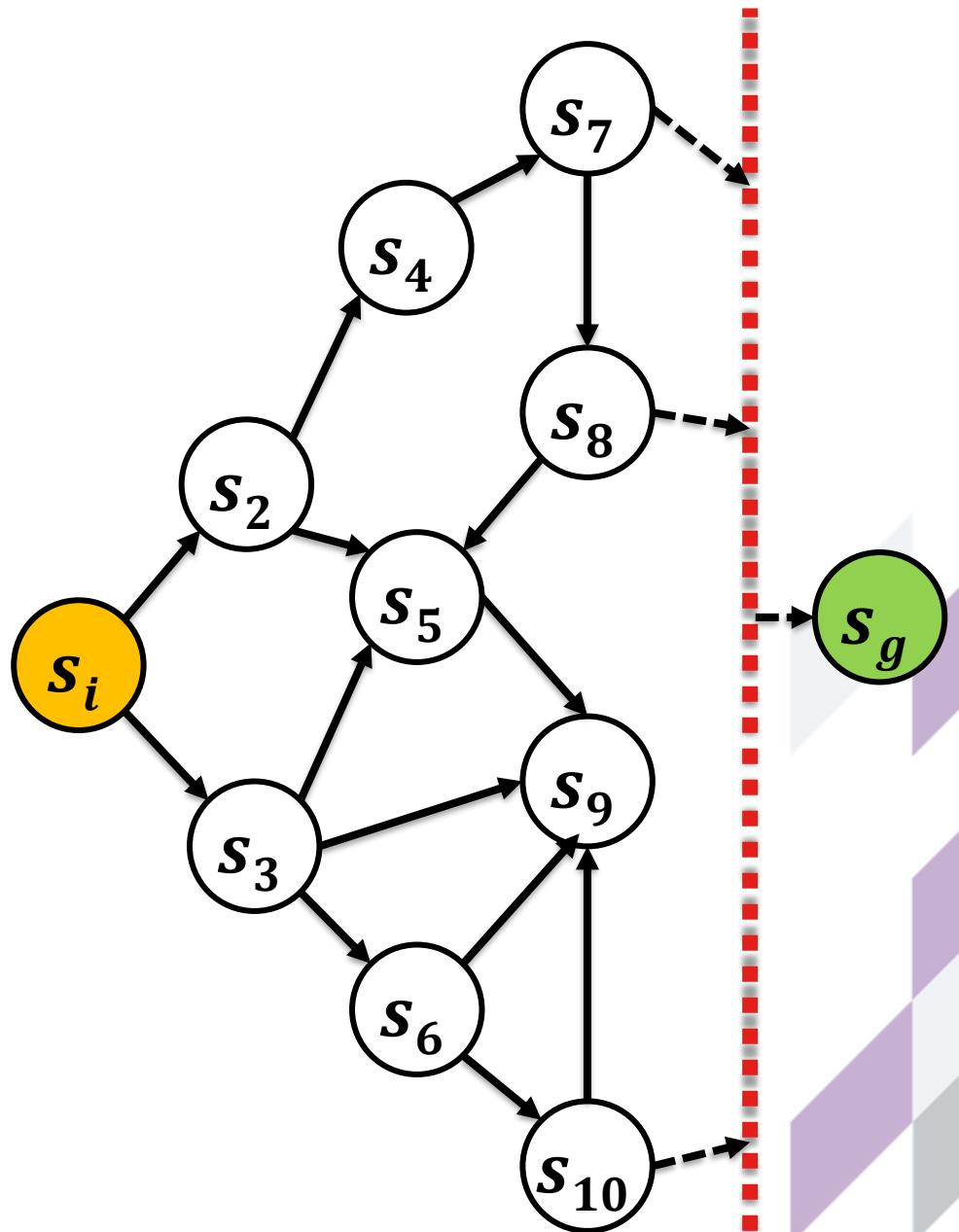
# Classical Planning Non-Forward Search Partial Order Planning

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



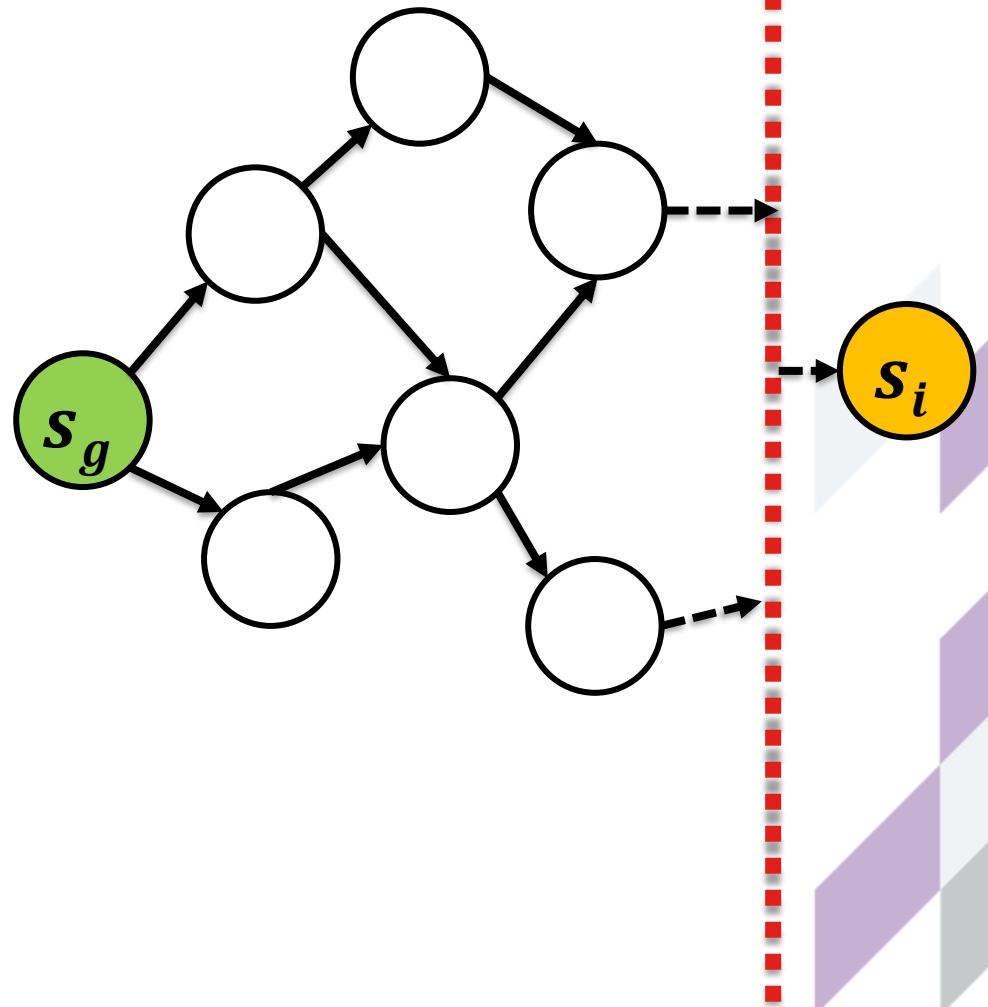
# Forward Search

- Searching forward from the initial to goal state can be problematic due to the nature of the problem space.
- We may wind up creating redundant steps in our plans.
  - This could be the fault of the heuristic.
  - Or is a reflection of a corner of the search space we fall into.
- We may find ourselves in a situation where we're caught in a loop.
  - Heuristic search should avoid this, but uninformed search is vulnerable.
- Depending on the algorithm, we can be at the mercy of the state space to actually find the solution.



# Regression Search

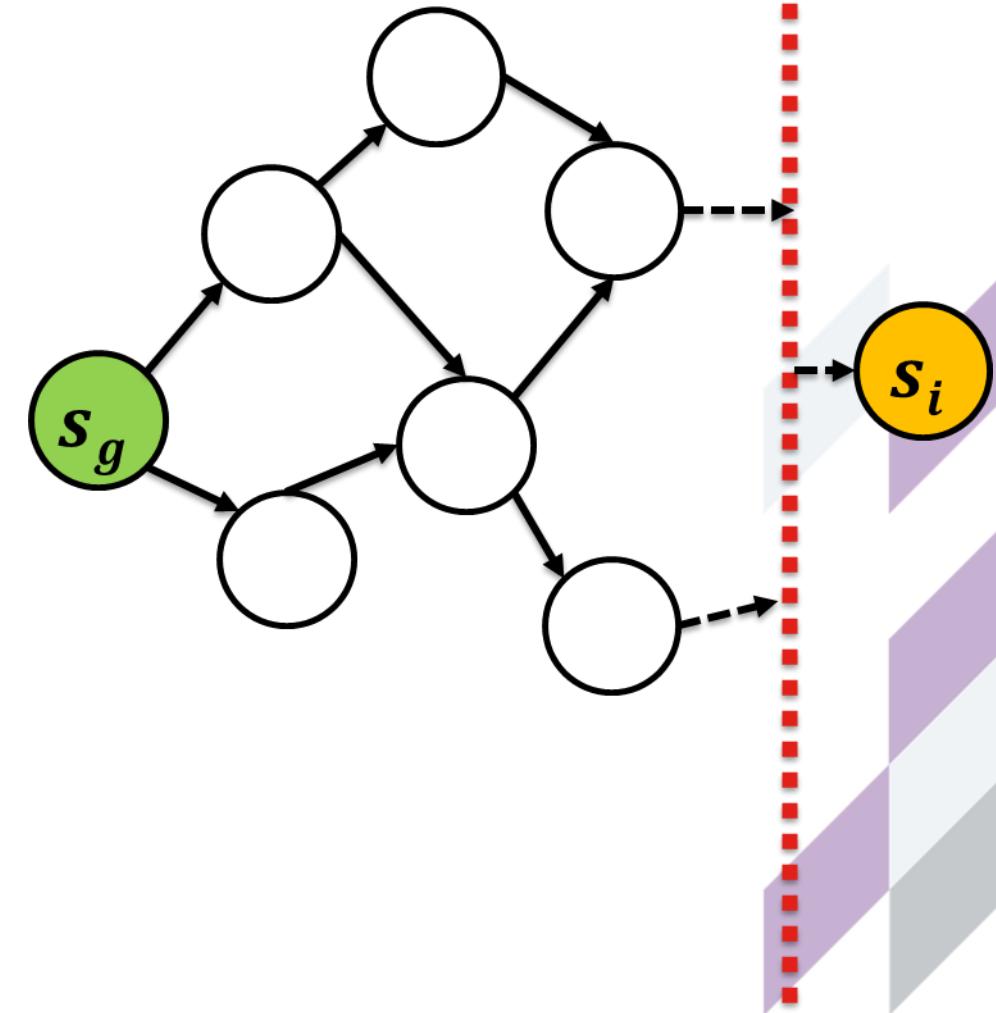
- Searching backwards from the goal state.
- Instead of aiming of searching for a state that holds all goals, we search for assignments of the goal facts.
- Find actions with add effects that create goal facts.
- Then add (unsatisfied) preconditions as subgoals to achieve.



# Regression Planning: More Formally

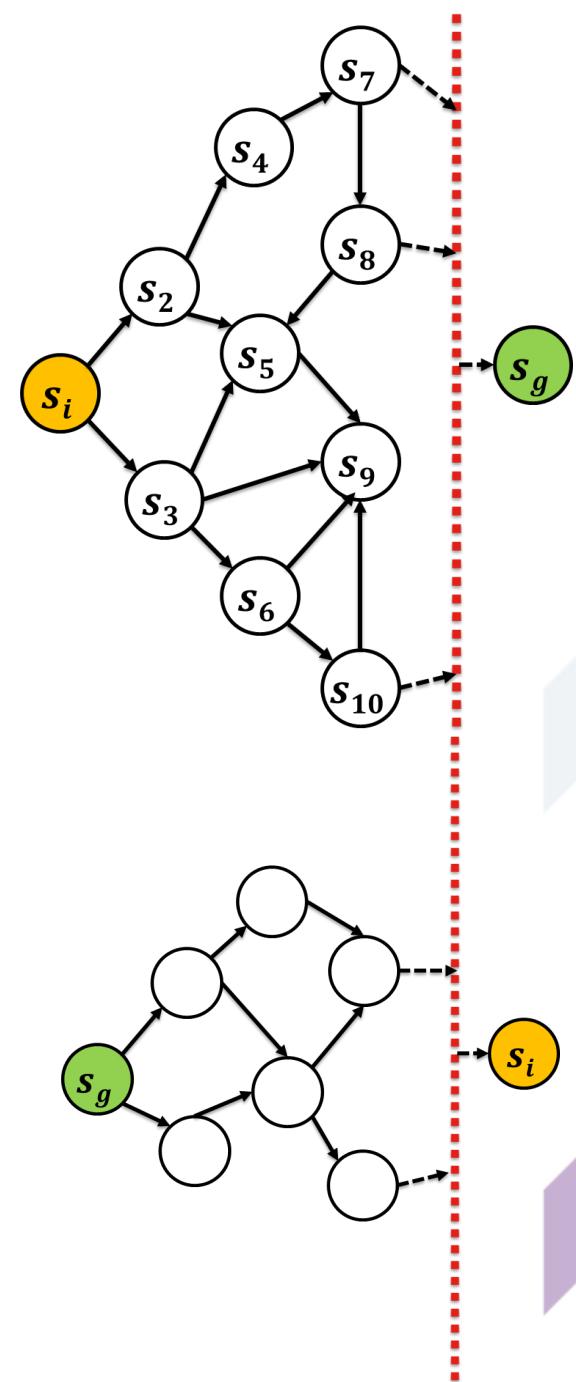
- The set of actions  $A(s)$  applicable in  $s$  are the operators  $op \in O$  that are relevant and consistent.
  - Namely, for which  $Add(op) \cap s \neq \emptyset$  and  $Del(op) \cap s = \emptyset$
  - Operators that add something in the state, and don't delete anything in the state.
- The state  $s = f(a, s)$  that follows the application of  $a \in A(s)$  is such that:

$$s = s - Add(a) + Pre(a)$$



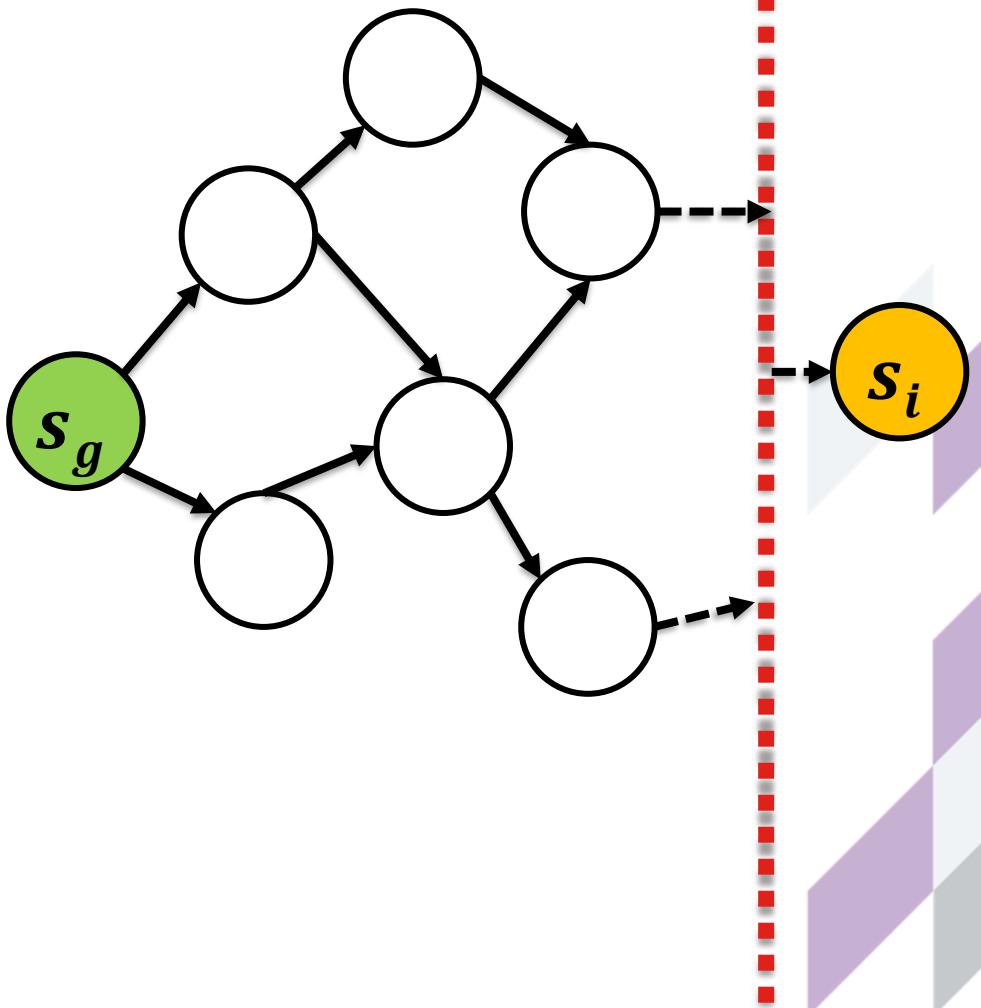
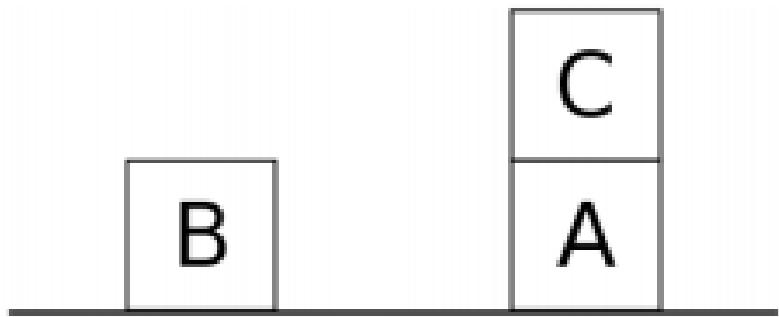
# Why Search Backwards

- But, why is searching backwards any different?
- States spaces when searching forwards vs backwards are not symmetric.
- **Forward Search:**
  - One initial state.
  - Apply action  $a$  to state  $s$ ? One valid unique successor  $s'$ .
- **Backward Search**
  - A set of goal states.
  - Multiple states where  $a$  can be applied to reach  $s'$ .

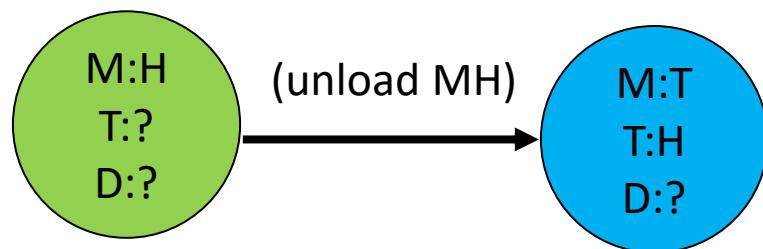


# Regression Search: STRIPS

- **STRIPS**
  - The **STanford Research Institute Problem Solver**
  - Arguably the first planning system dating back to 1971
- **Used regression search from the goal state.**
- **But... it was incomplete.**

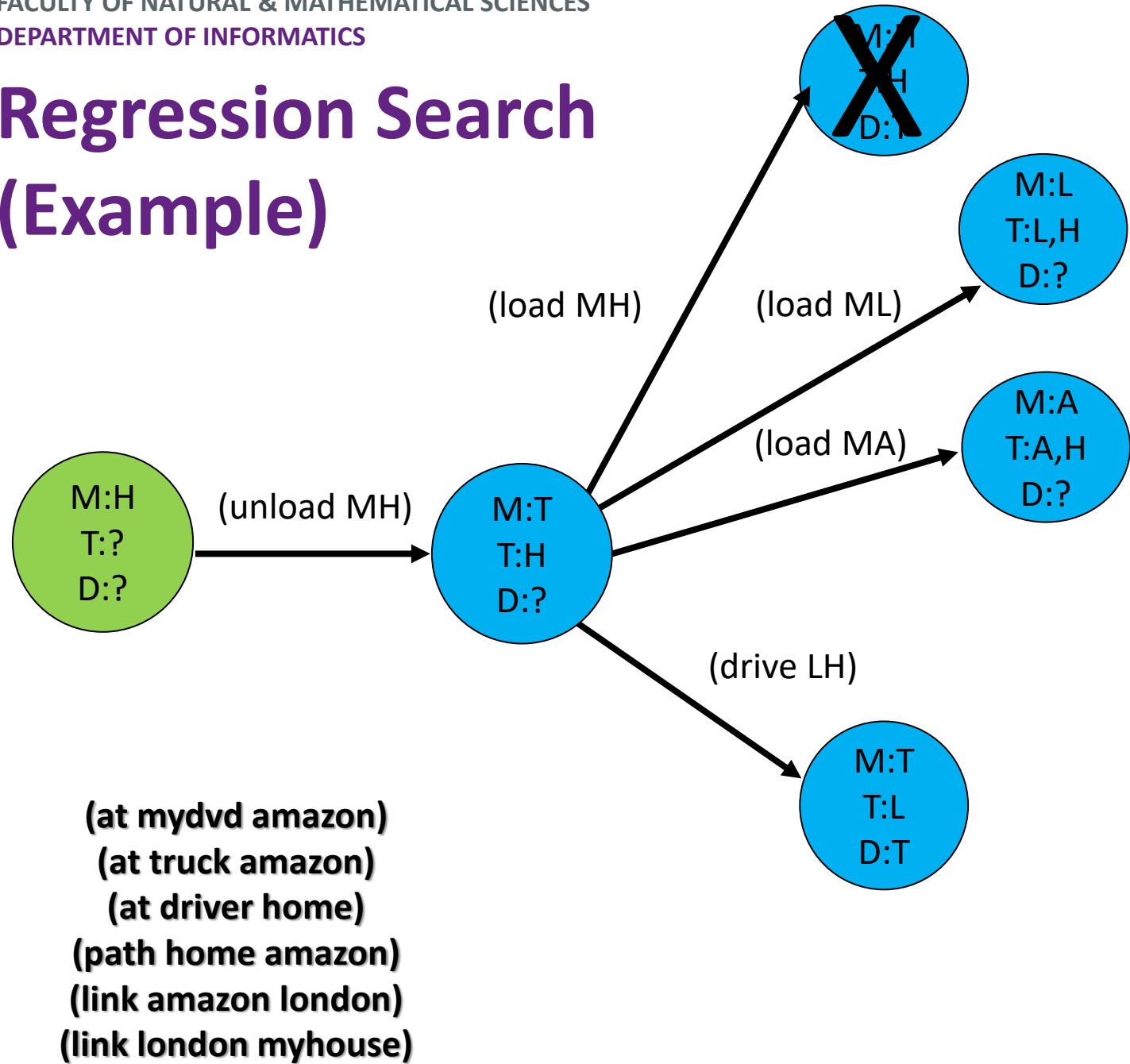


# Regression Search (Example)

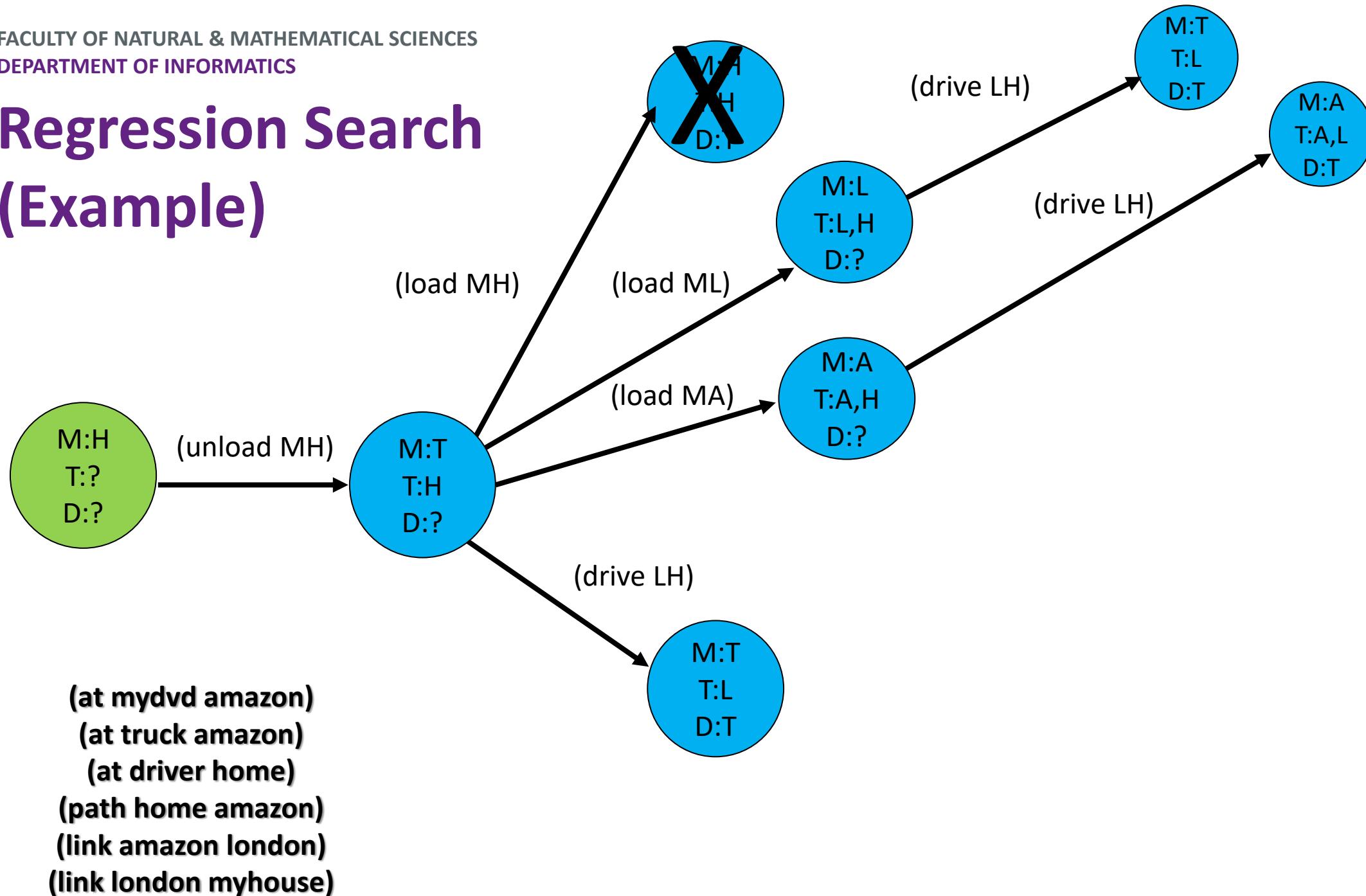


**(at mydvd amazon)**  
**(at truck amazon)**  
**(at driver home)**  
**(path home amazon)**  
**(link amazon london)**  
**(link london myhouse)**

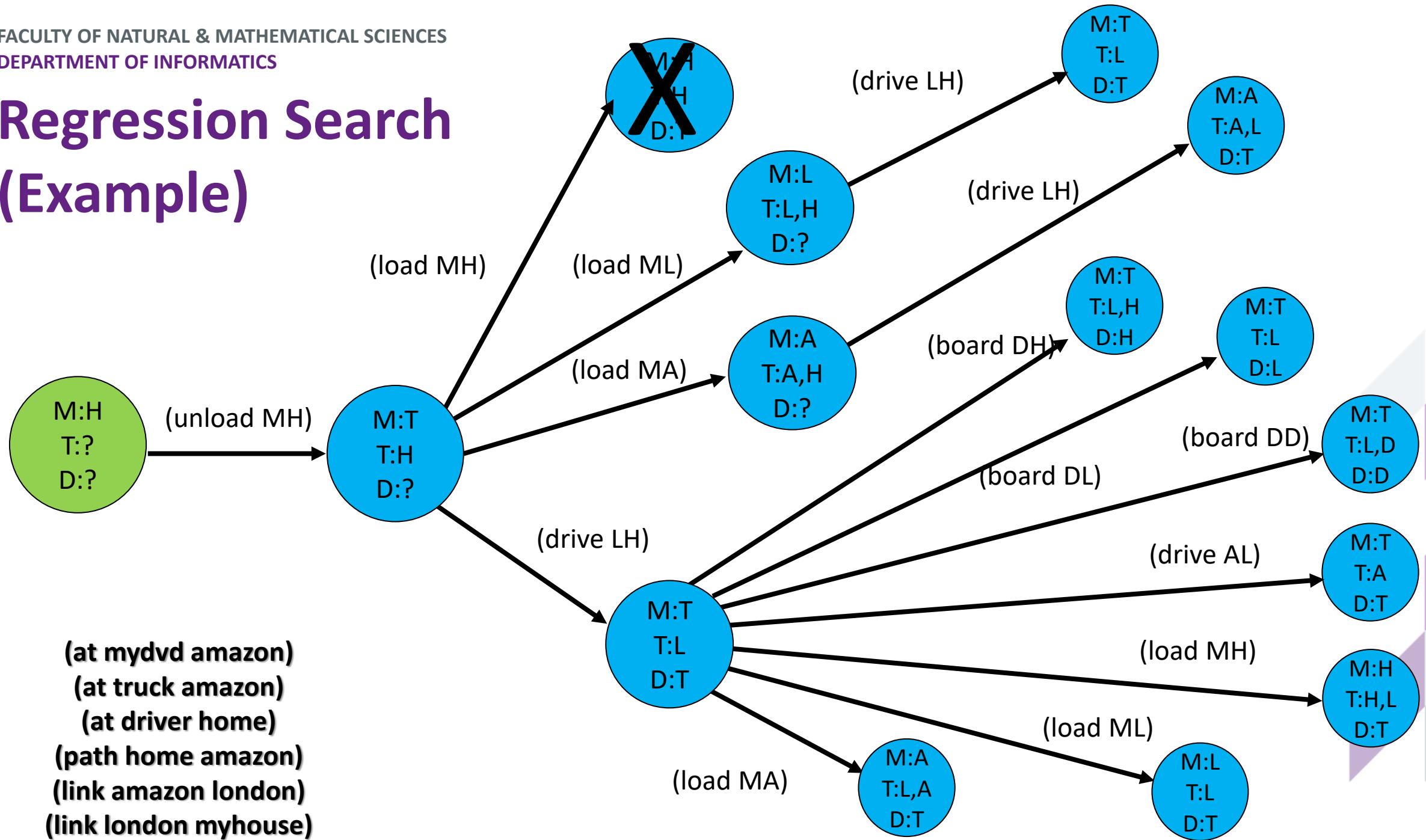
# Regression Search (Example)



# Regression Search (Example)



# Regression Search (Example)



# Partial Order Planning: Searching in Plan Space

- **Planning in plan space, using a lifted framework.**
- Initial plan, initial state, goal state.
- **Either:**
  - Pick an **unresolved goal**.
    - Select an **action** to achieve it (or the initial state)
    - Add to the plan
    - Add a **causal link** and add its preconditions as open goals.
  - Find an **unbound parameter** and select a **binding**.
  - Find a **threat**, resolve by **promotion** or **demotion**.

# POP Example

Init

Link LH    Link AL    At TA    at DVDA

At ?T ?L    In ?P ?T

Unload ?P ?L ?T

At DVD H

Goal

Ignoring the Driver

# POP Example

Init

Link LH    Link AL    At TA    at DVDA

At T H    In DVD T

Unload DVD H T

At DVD H

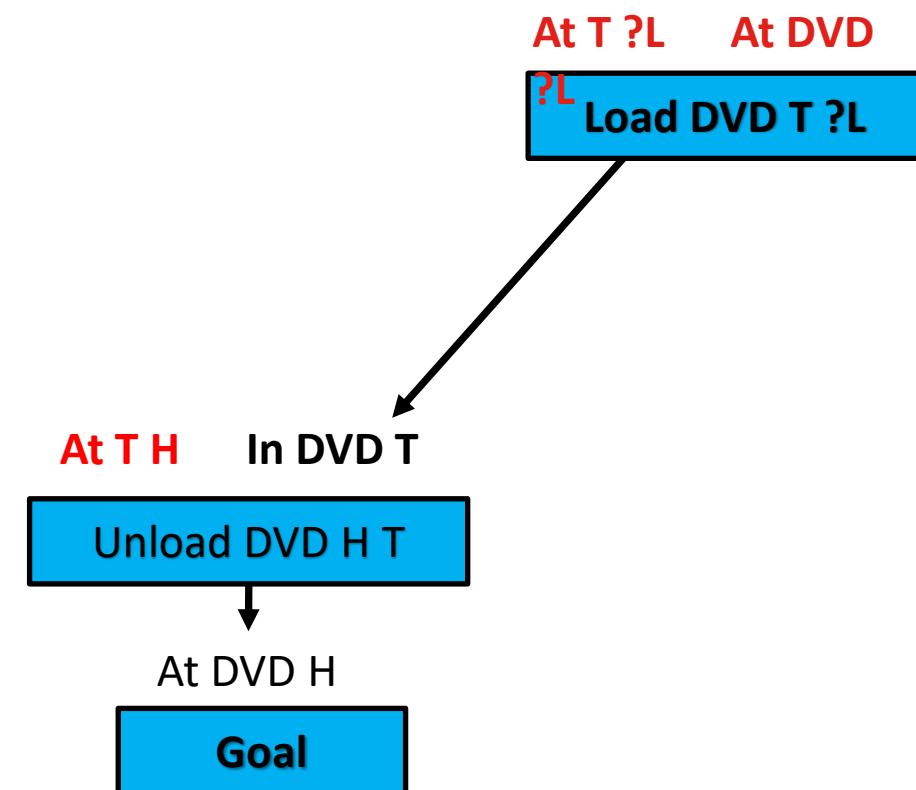
Goal

Ignoring the Driver

# POP Example

Init

Link LH    Link AL    At TA    at DVDA



Ignoring the Driver

# POP Example

Init

Link LH      Link AL      At TA      at DVDA

Link LH      At TL  
Drive T L H

At T H      In DVD T

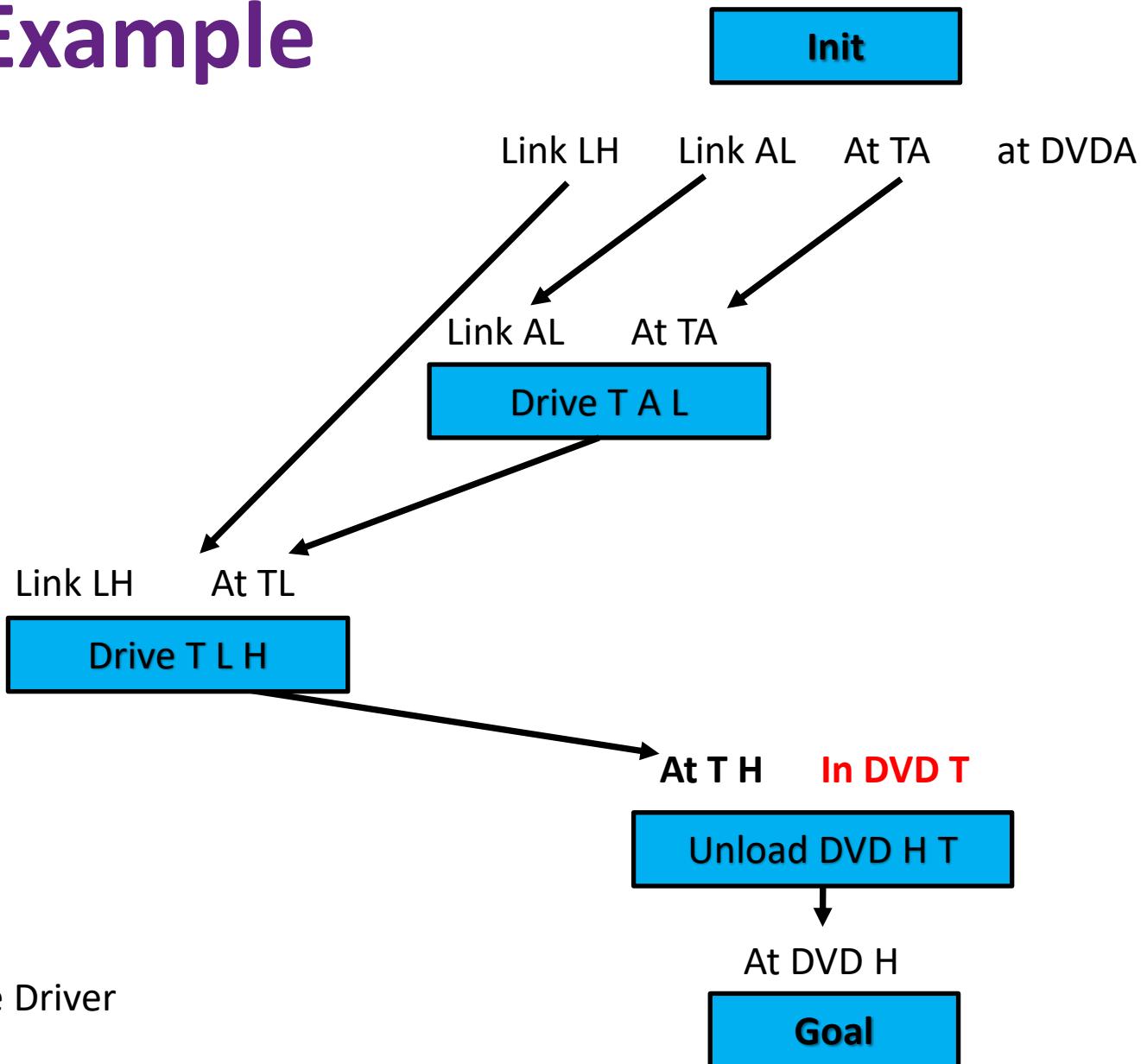
Unload DVD H T

At DVD H

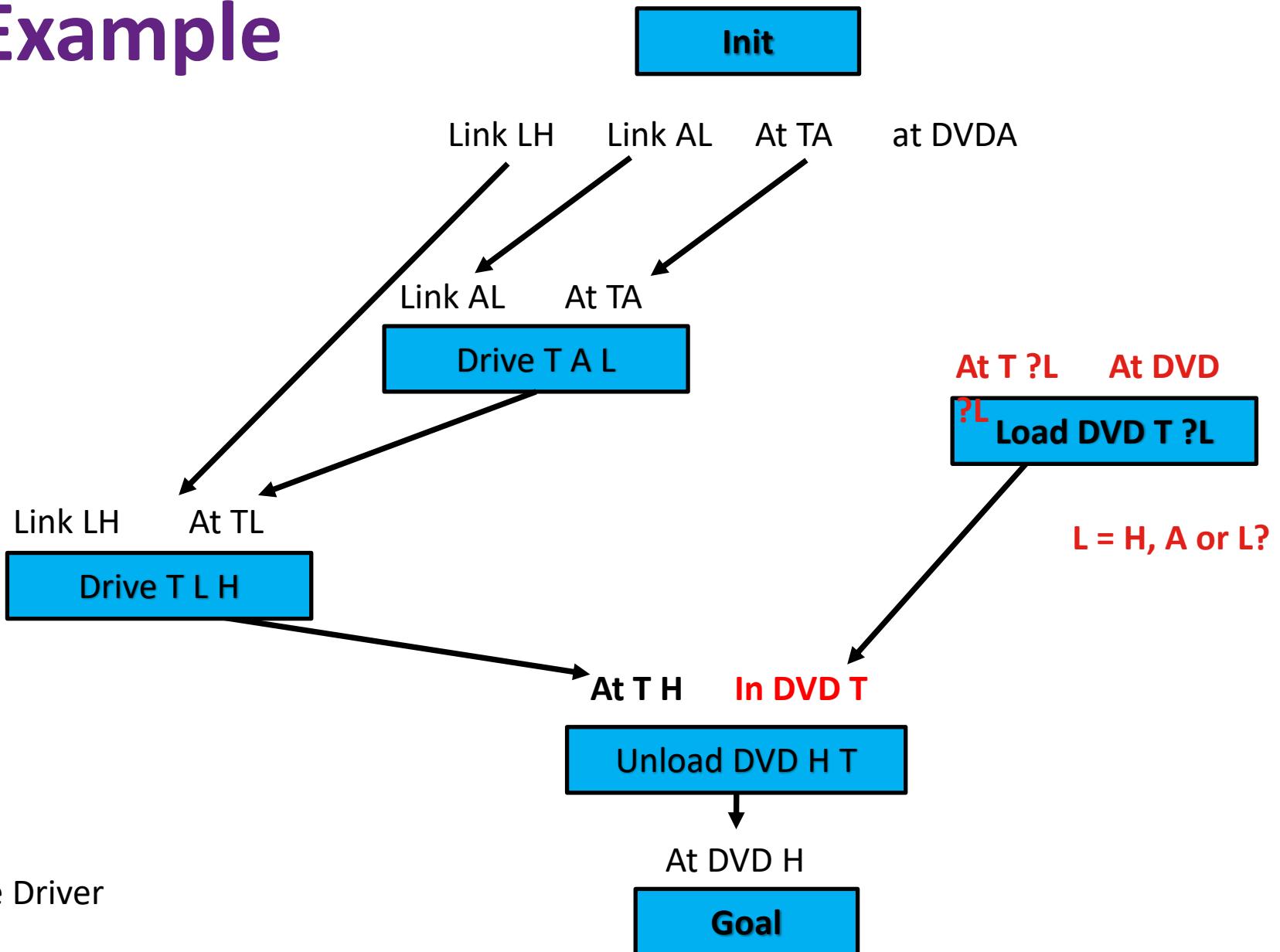
Goal

Ignoring the Driver

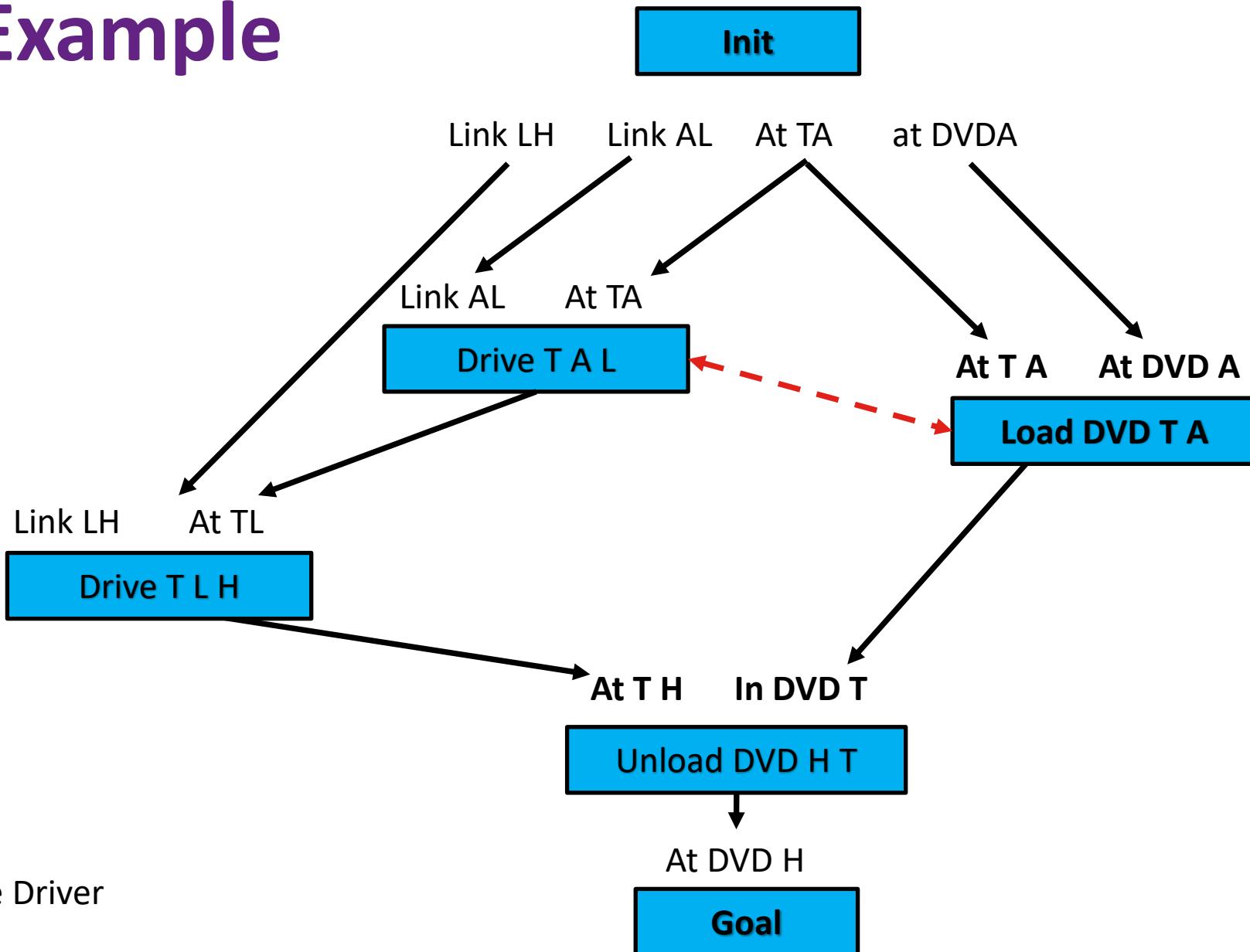
# POP Example



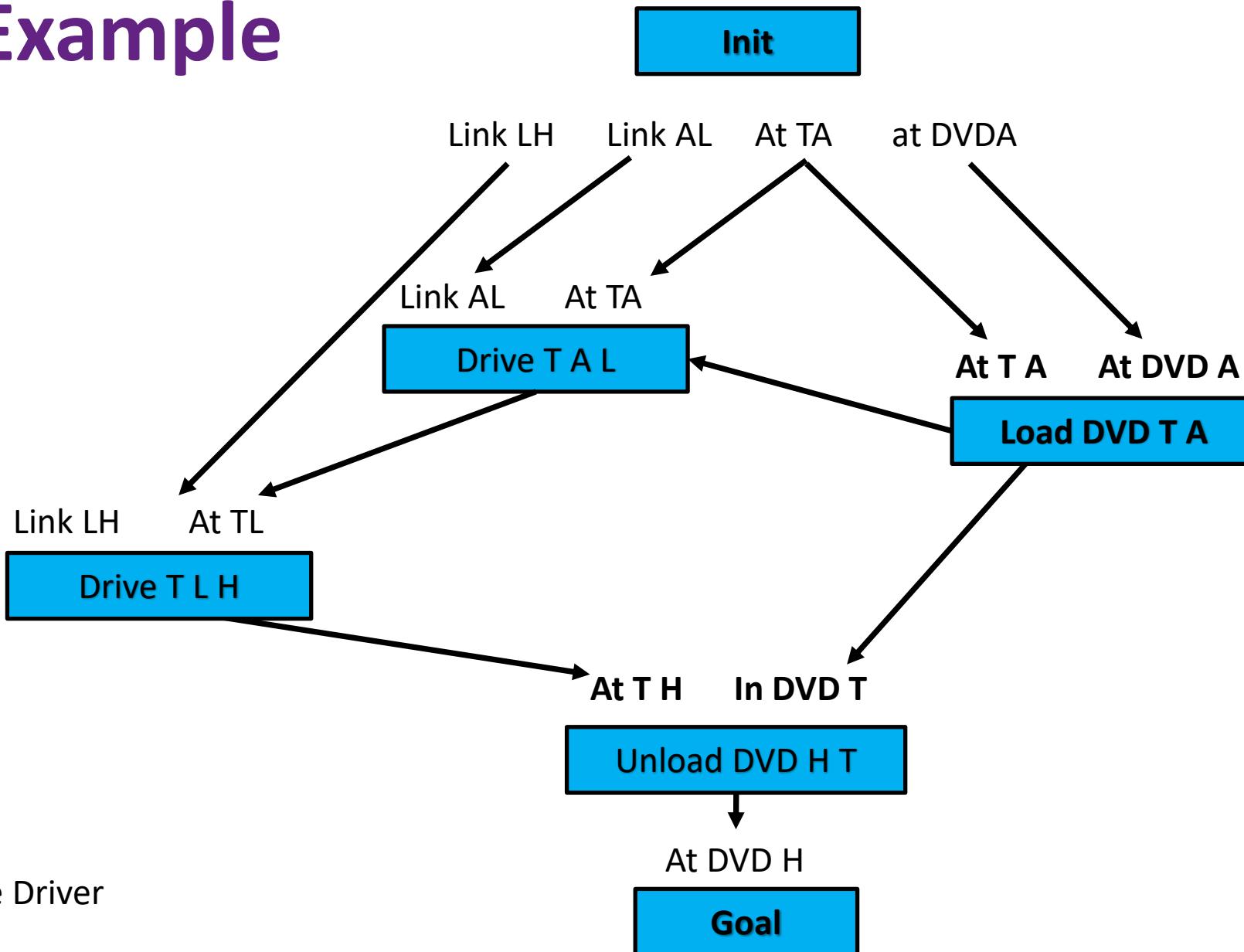
# POP Example



# POP Example

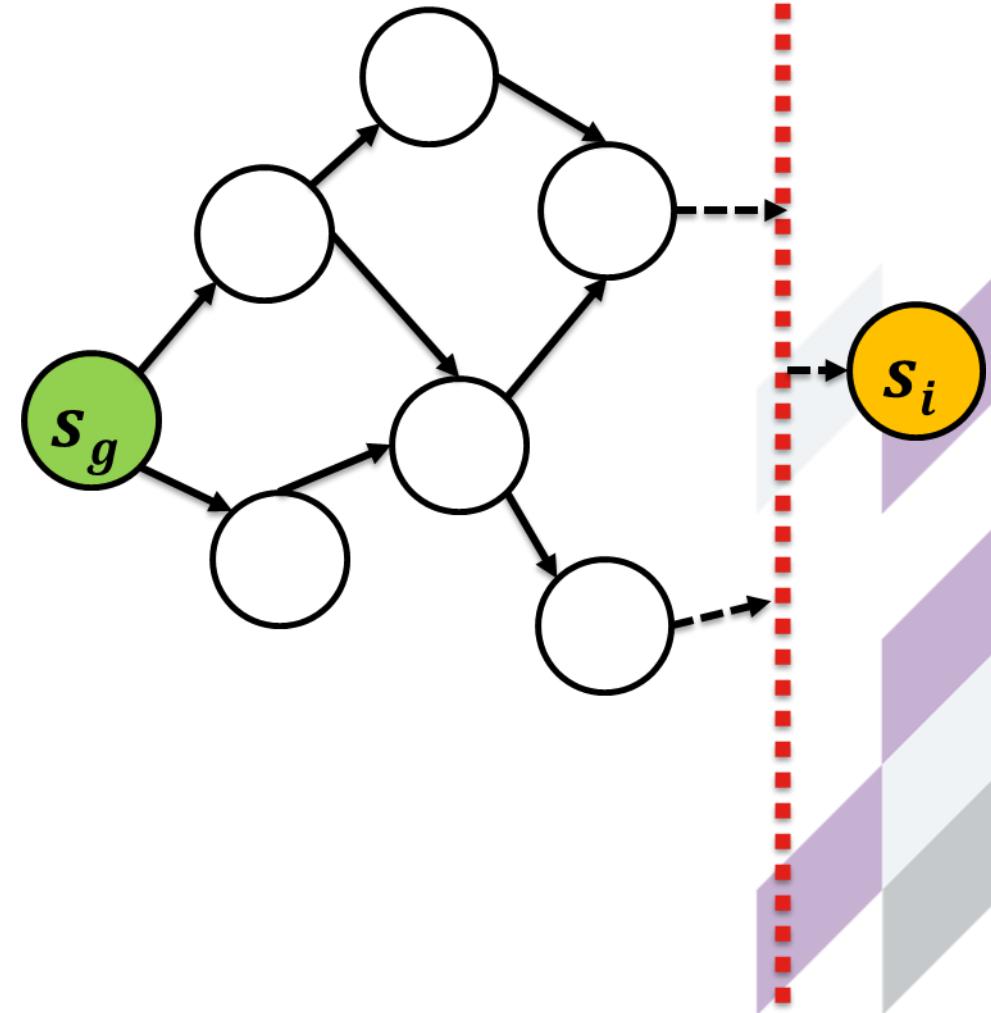


# POP Example



# Summary

- Forward search is the most popular, but not the only process to find solutions.
- Regression can help us reduce state spaces.
- Basis for Partial Order Planning (POP)
- But it also has its own issues.
  - High branching factor: threats, bindings, actions.
  - Hard to compute heuristics.
- Produces nice ‘least commitment’ plans though.



# Classical Planning Non-Forward Search Partial Order Planning

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



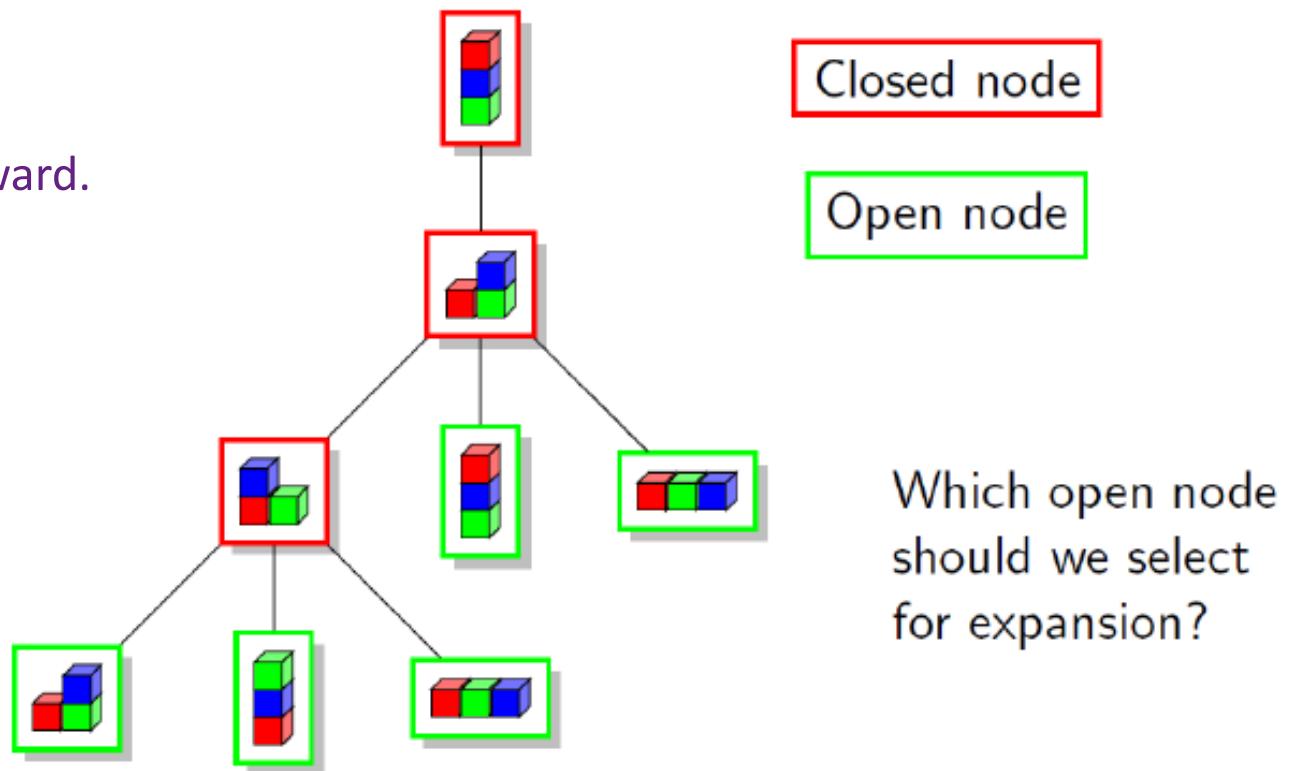
# Classical Planning Non-Forward Search Introduction to HTN Planning

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



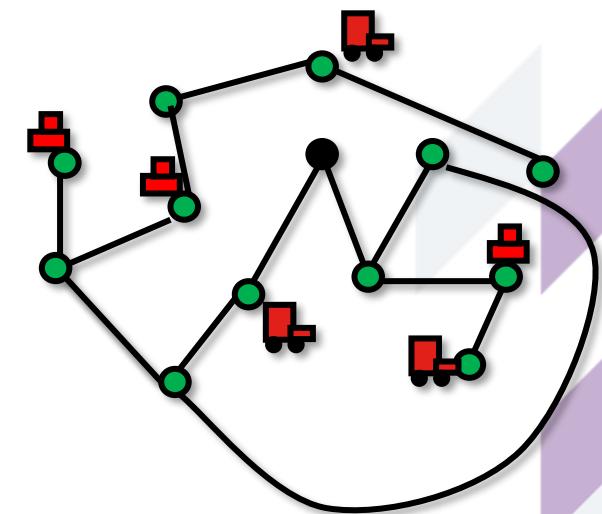
# Recap: Non-Forward Search

- Exploring planning techniques that deviate from the traditional forward-search approach.
  - Initial state, open list, closed list, heuristic etc.
- **Graphplan:**
  - Generate plan graphs forward, then search backward.
- **SAT Planning**
  - Planning as satisfiability
- **POP Planning**
  - Partial orders on plan execution.



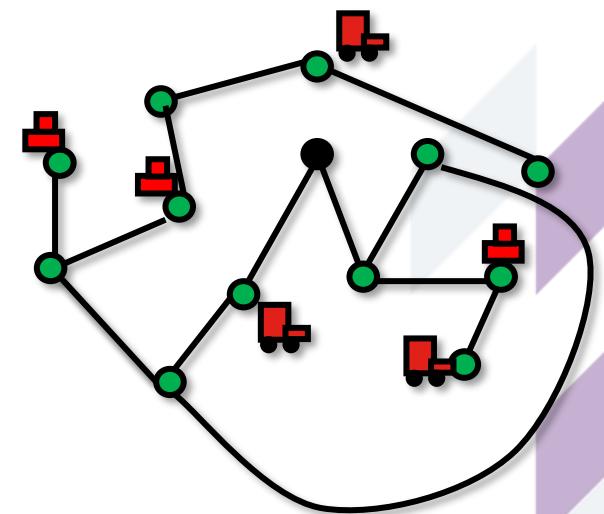
# Hierarchical Task Network (HTN) Planning

- In real life, we seldom plan down to the minutiae of detail, we instead think more abstractly.
  - E.g. “getting out of the house in the morning”:
    - waking up, washed, dressed, breakfast etc.
- In many planning problems, we – as the designer – may already have ideas about what good solutions might look like.
- E.g. driving to packages to pick them up, driving to destinations to drop them off.
- Basic Idea: What if we put this information directly into the domain?



# Hierarchical Task Network (HTN) Planning

- Reframe the planning problem:
  - Tasks to complete instead of goals to satisfy.
    - Tasks are little more abstract than a typical planning problem.
  - Methods to decompose tasks into subtasks.
    - Enables for gradual decompositions of complex task into smaller and more manageable tasks.
    - A subtask could be a single action, or another method in and of itself.
  - We enforce constraints of the problem in the method and actions.
  - Simplest version: Simple Task Network (STN) Planning



# HTN Planning: More Formally...

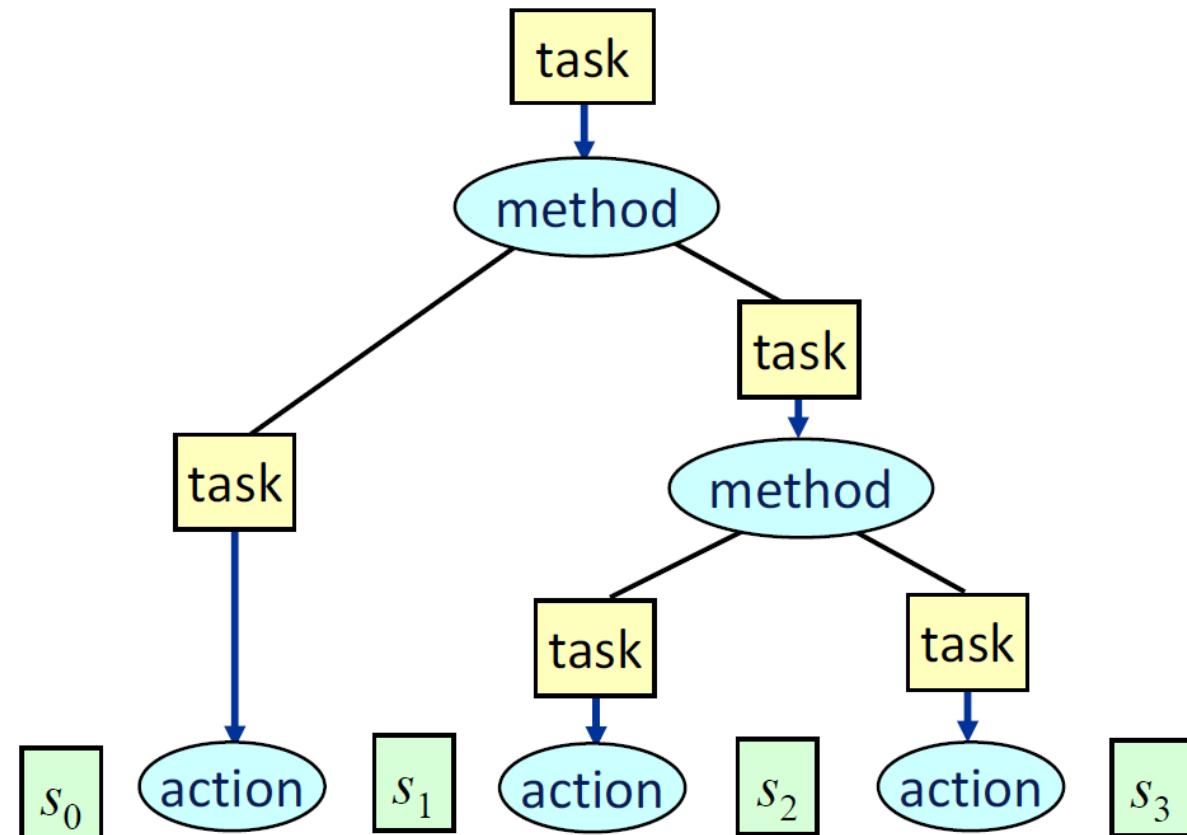
- **HTN Planning Problem:**  $(\Sigma, M, A, s_0, T)$

- $\Sigma$  - a state-variable planning domain.
- $M$  - a set of methods
- $s_0$  - the initial state of the problem
- $T$  - a list of tasks to complete

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

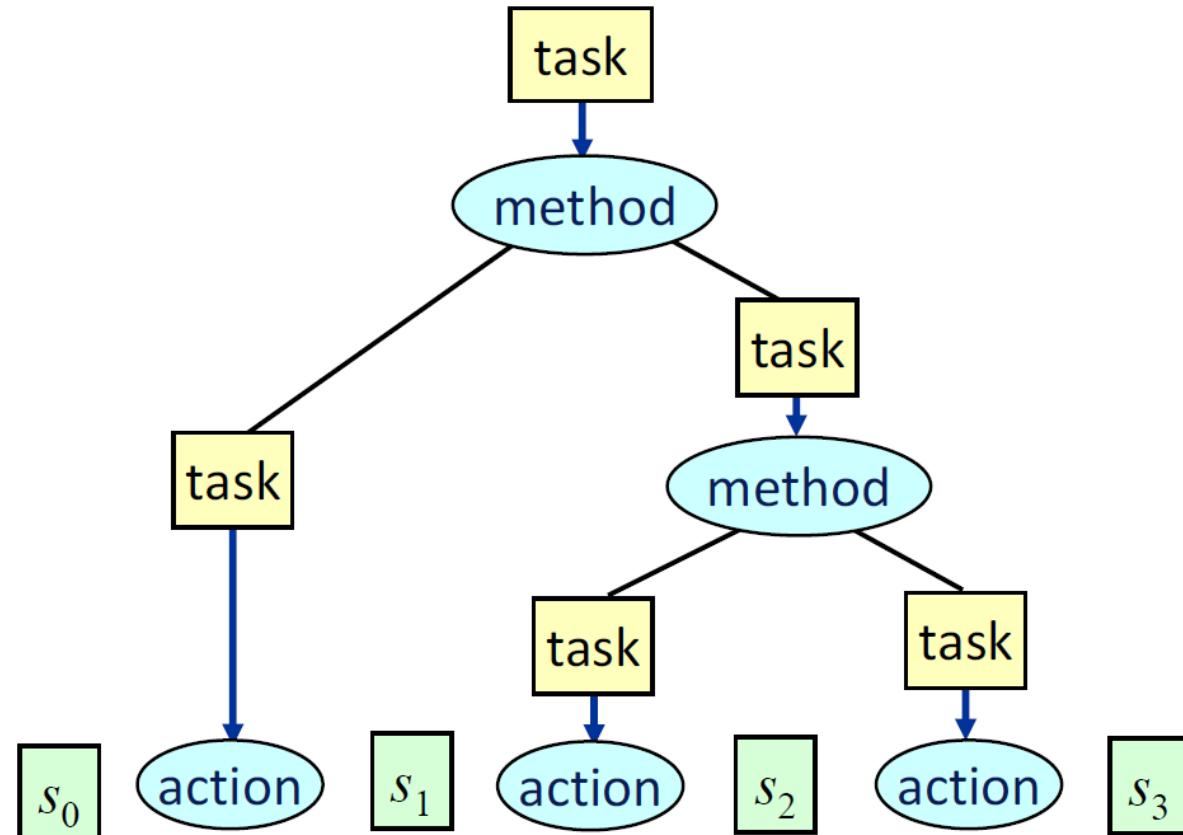
$$s \in S, a \in A$$

$$\gamma(s, a) \rightarrow s'$$

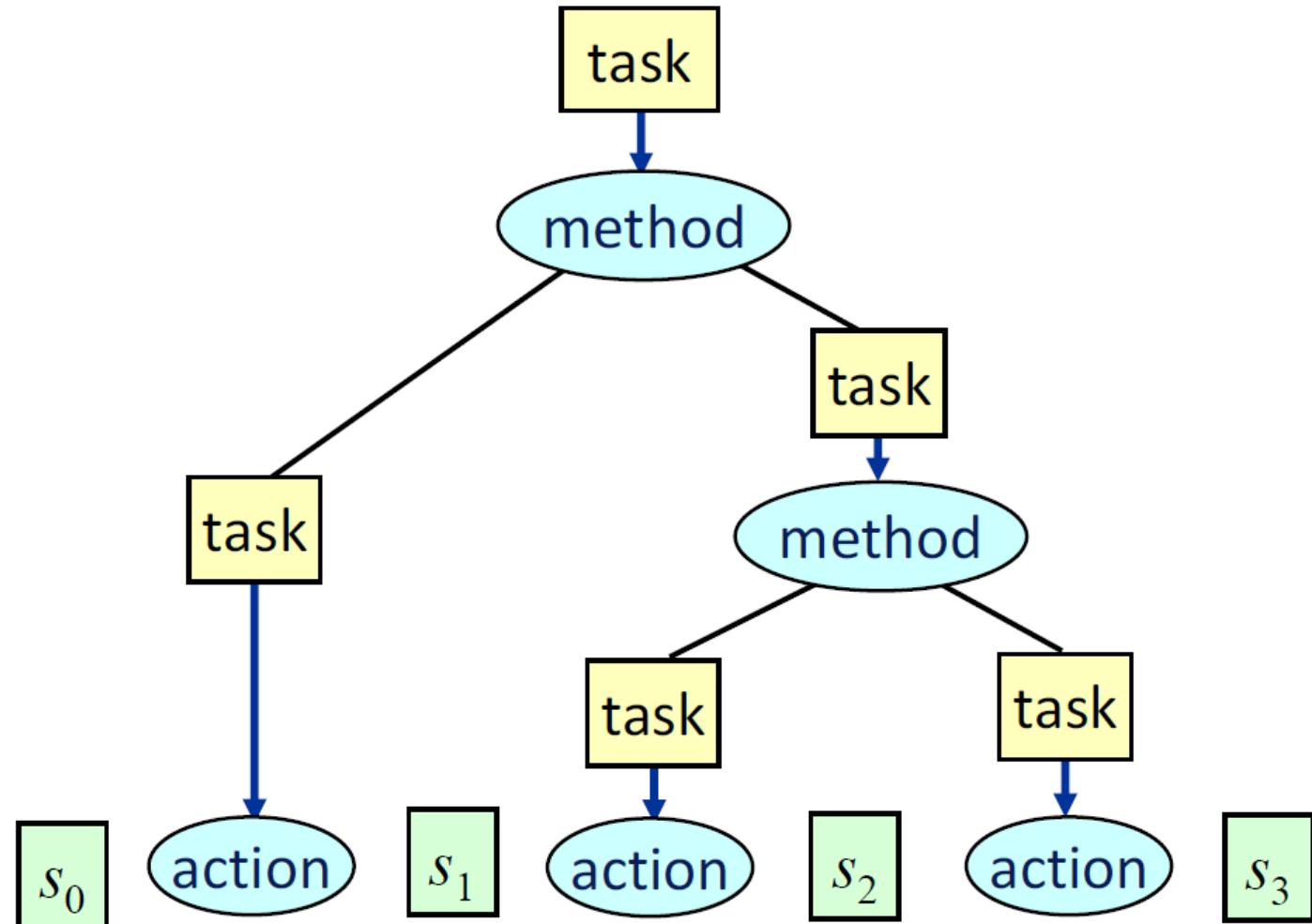


# Hierarchical Task Network (HTN) Planning

- HTN planning defines two types of tasks.
  - Primitive Tasks: Translate to single planning action.
  - Compound Tasks: Translate into a collection of one or more tasks.
- Planning Process:
  - Recursively refine each task into subtasks.
  - Ensure tasks can then be grounded with literals.

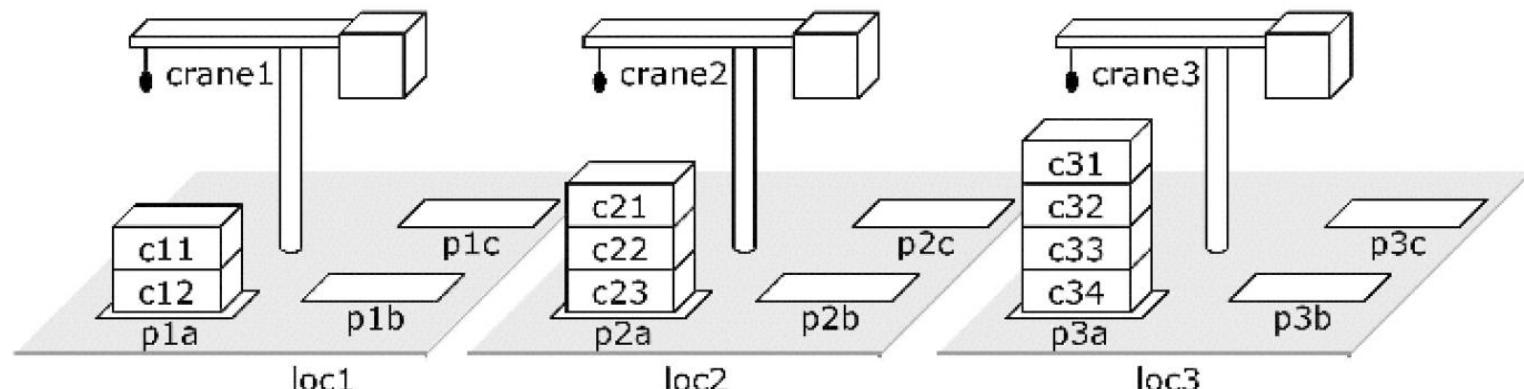


# Hierarchical Task Network (HTN) Planning

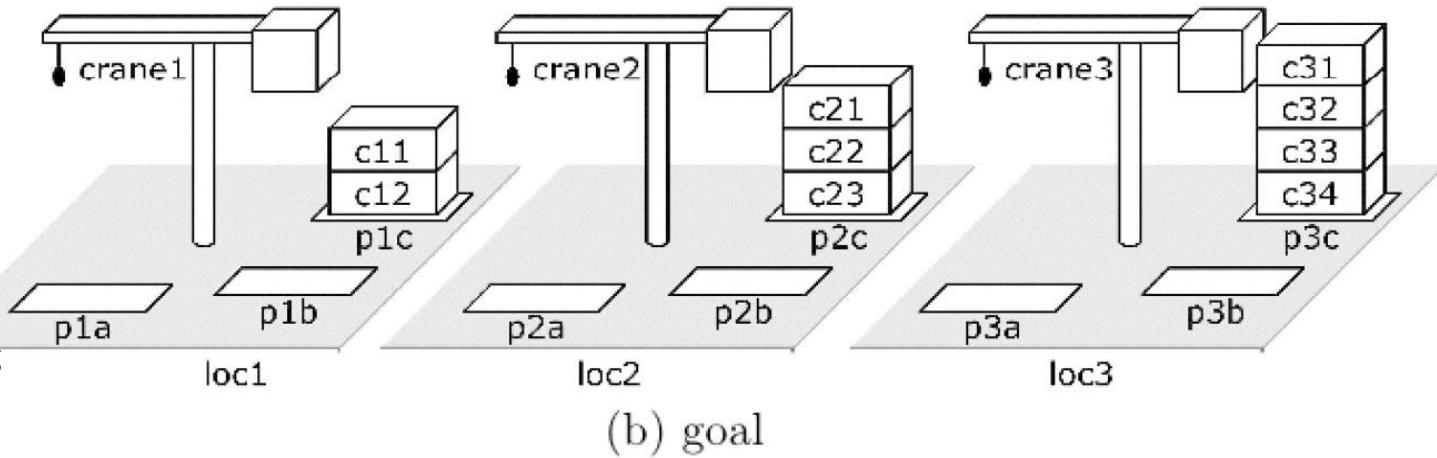


# HTN Example: Dockworker (DWR)

- **Task:** Moving containers from one pallet in the yard to another.
- **Goal:** Moving containers from original pallet X to destination Y, while preserving the order of the containers on the new pallet.



(a) initial state



(b) goal

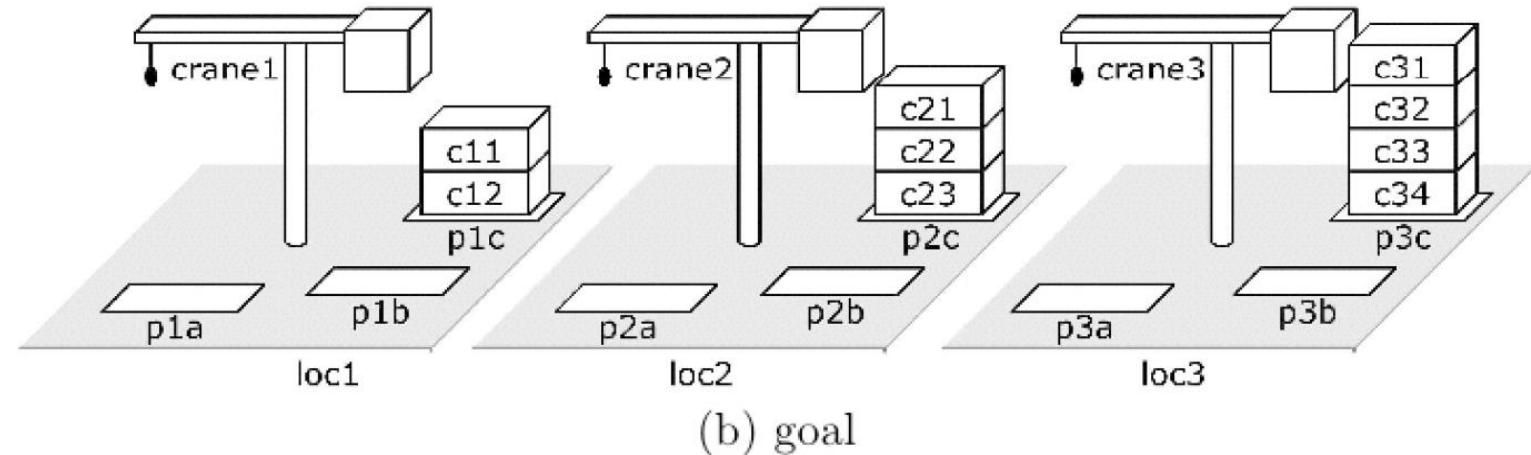
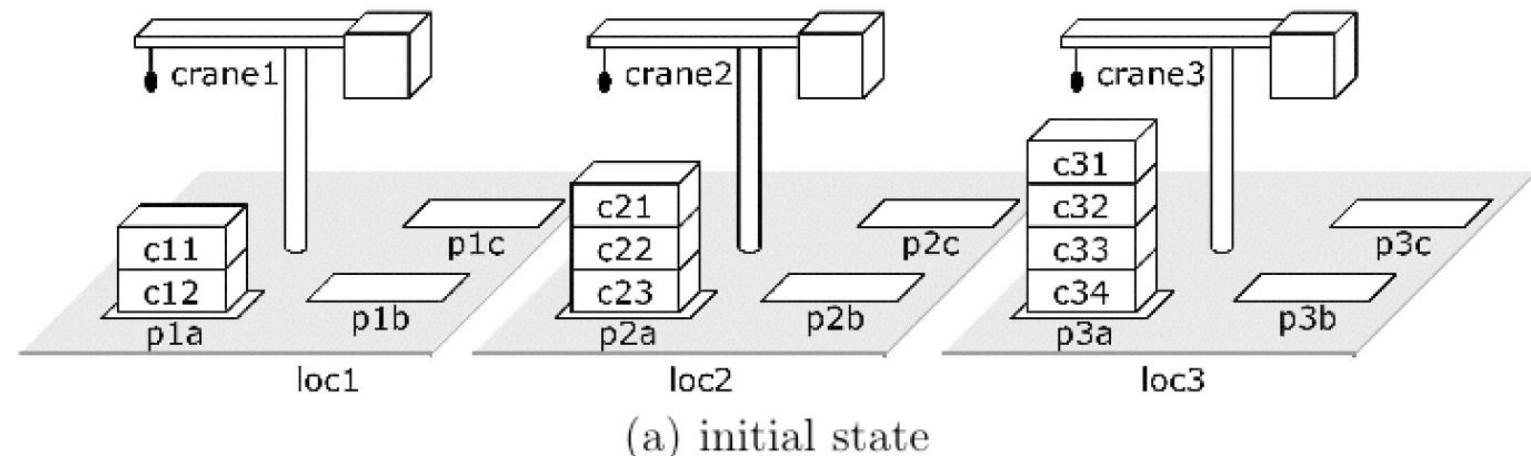
# HTN Example: Dockworker (DWR)

- Task:

$move - stack(p, q)$

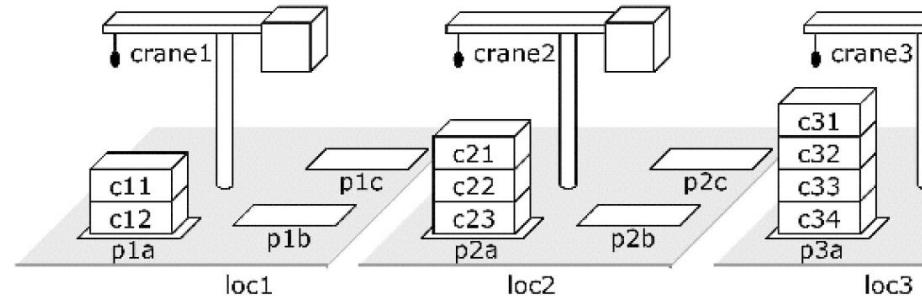
- Parameters:

- $p$  – The original location.
- $q$  – The destination location.

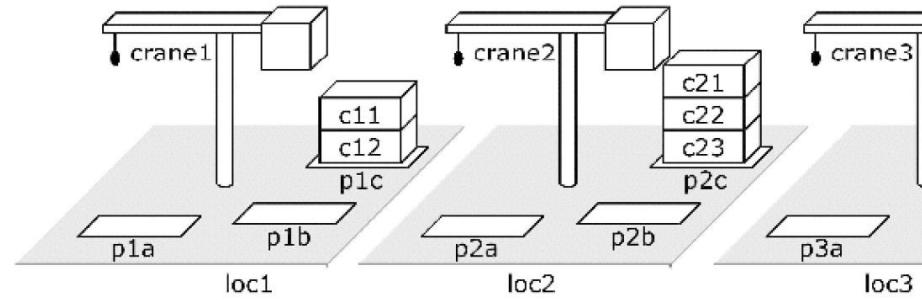


# Dockworker: Actions

- **Take  $\langle k, l, c, p, o \rangle$** 
  - Preconditions
    - $belong(k, l), attached(p, l), empty(k), in(c, p), top(c, p), on(c, o)$
  - Effects
    - $holding(k, c), top(o, p), not(in(c, p)), not(top(c, p)), not(on(c, o)), not(empty(k))$
- **Put  $\langle k, l, c, p, o \rangle$** 
  - Preconditions
    - $belong(k, l), attached(p, l), holding(k, c), top(o, p)$
  - Effects
    - $in(c, p), top(c, p), on(c, o), not(top(o, p)), not(holding(k, c)), empty(k)$



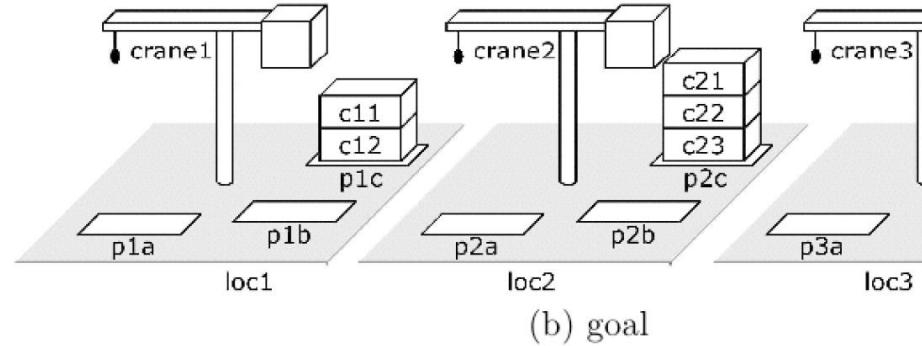
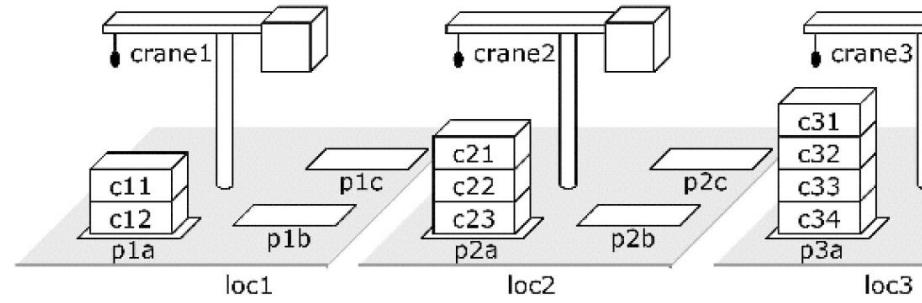
(a) initial state



(b) goal

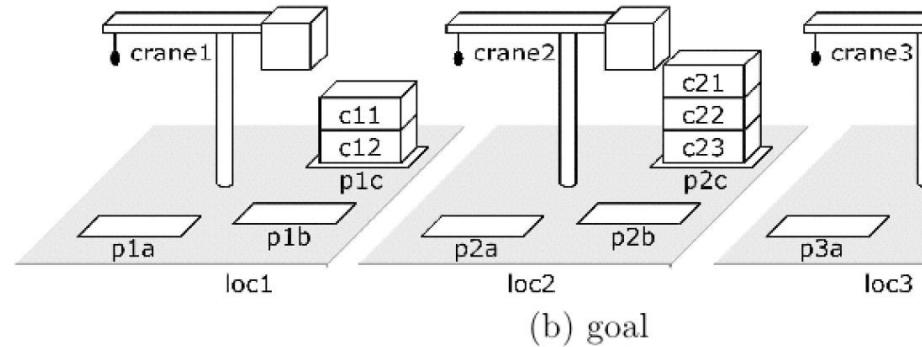
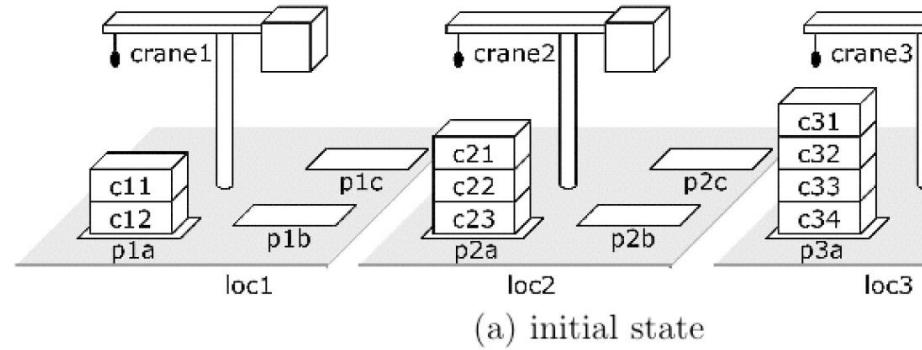
# Dockworker: Methods

- **Recursive-Move( $p, q, c, x$ )**
  - Task:  $move - stack(p, q)$
  - Pre:  $top(c, p), on(c, x)$
  - Subtasks:
    - $move - topmost - container(p, q)$
    - $move - stack(p, q)$
- **Do-Nothing( $p, q$ )**
  - Task:  $move - stack(p, q)$
  - Pre:  $top(pallet, p)$
  - Subtasks (none).

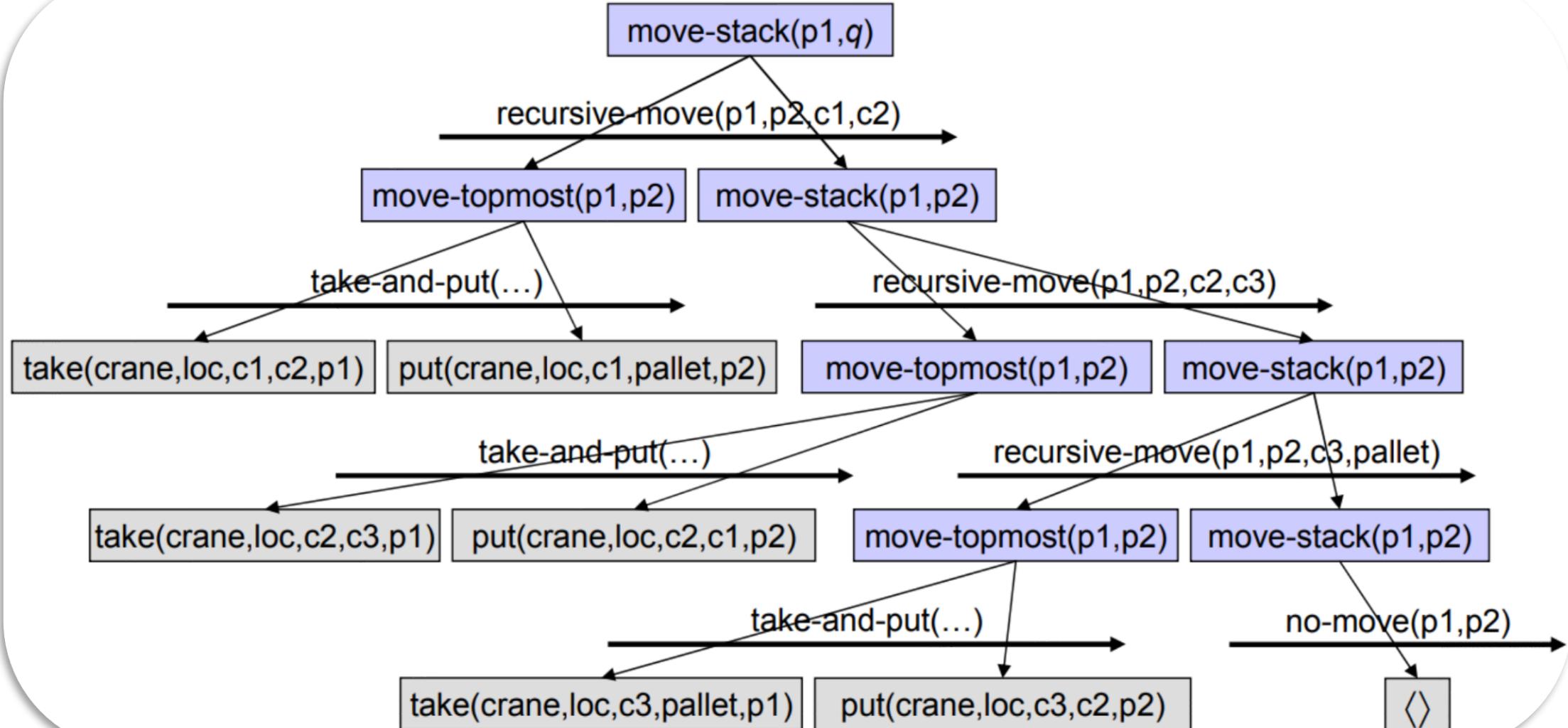


# Dockworker: Methods

- **Take-and-Put( $c, k, l_1, l_2, p_1, p_2, x_1, x_2$ )**
  - **Task:**  $move - topmost - container(p_1, p_2)$
  - **Pre:**  $top(c, p_1), on(c, x), attached(p_1, l_1), belong(k, l_1), attached(p_2, l_2), top(x_2, p_2)$
  - **Subtasks:**
    - $take(k, l, c, x_1, p_1)$
    - $put(k, l_2, c, x_2, p_2)$

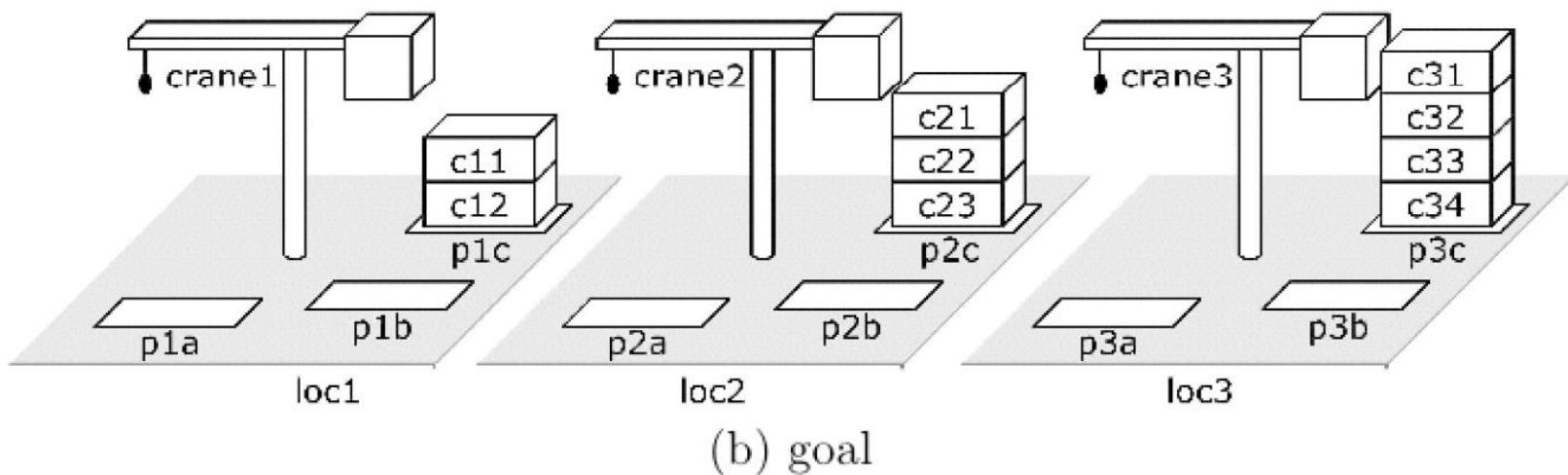
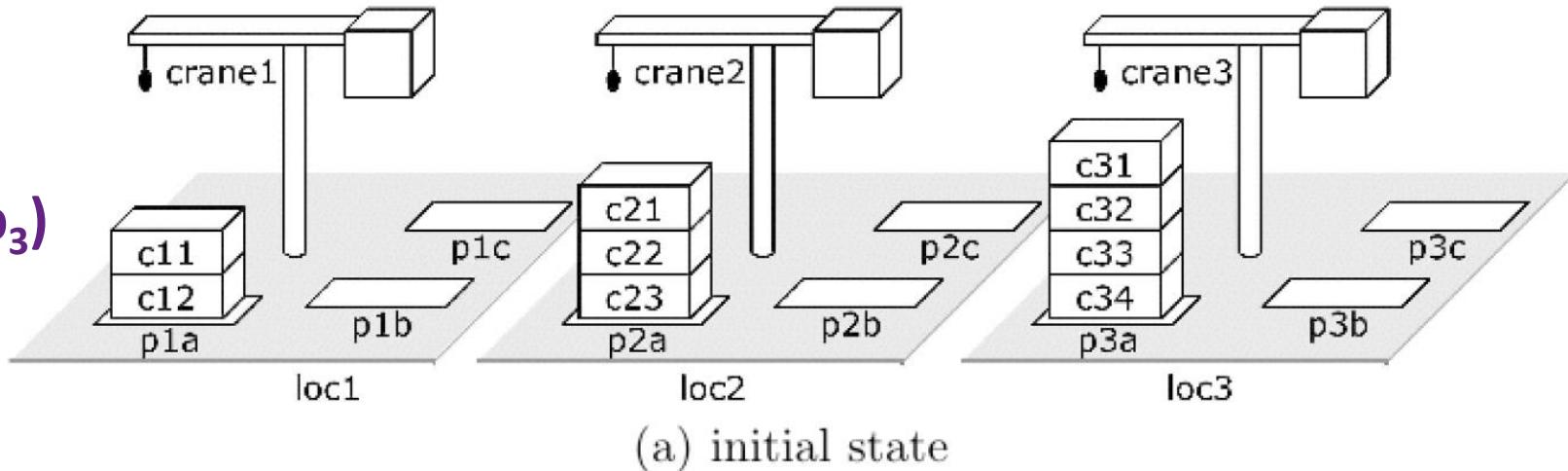


# Dockworker Example



# Dockworker: Methods

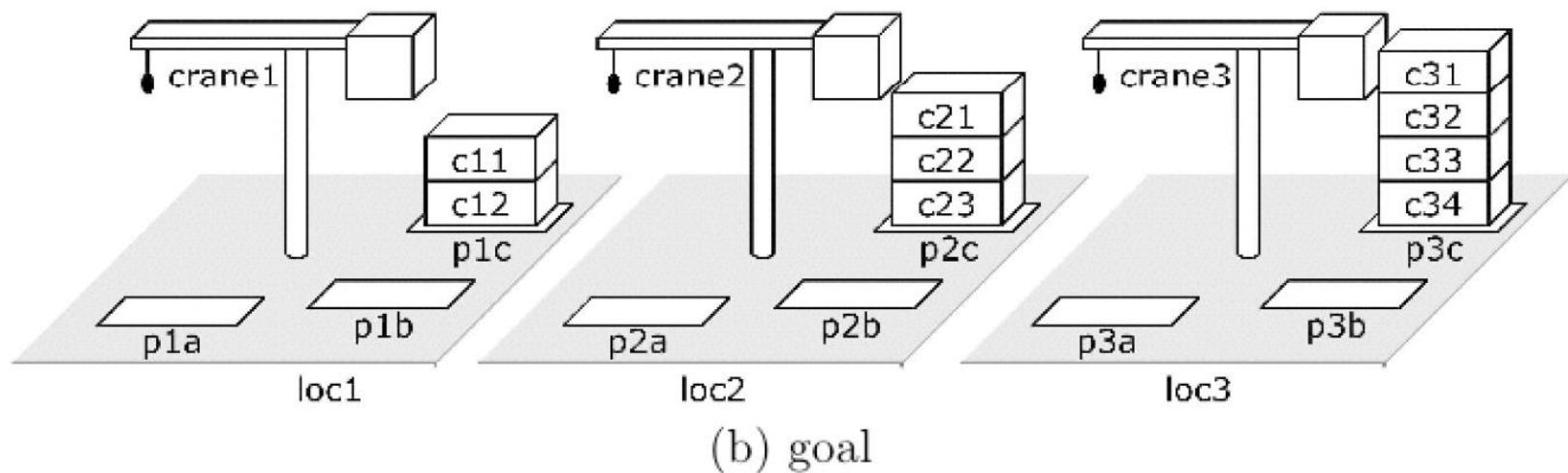
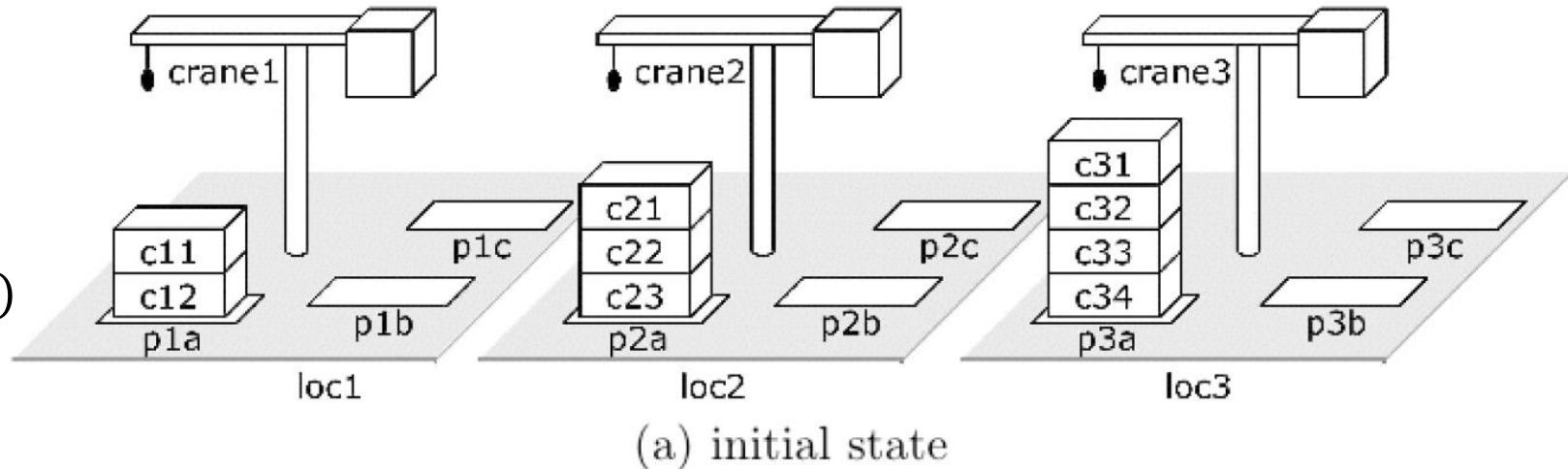
- **Move-Stack-Twice( $p_1, p_2, p_3$ )**
  - Task:  $move - stack(p_1, p_3)$
  - Pre: *None*
  - Subtasks:
    - $move - stack(p_1, p_2)$
    - $move - stack(p_2, p_3)$



# Dockworker: Methods

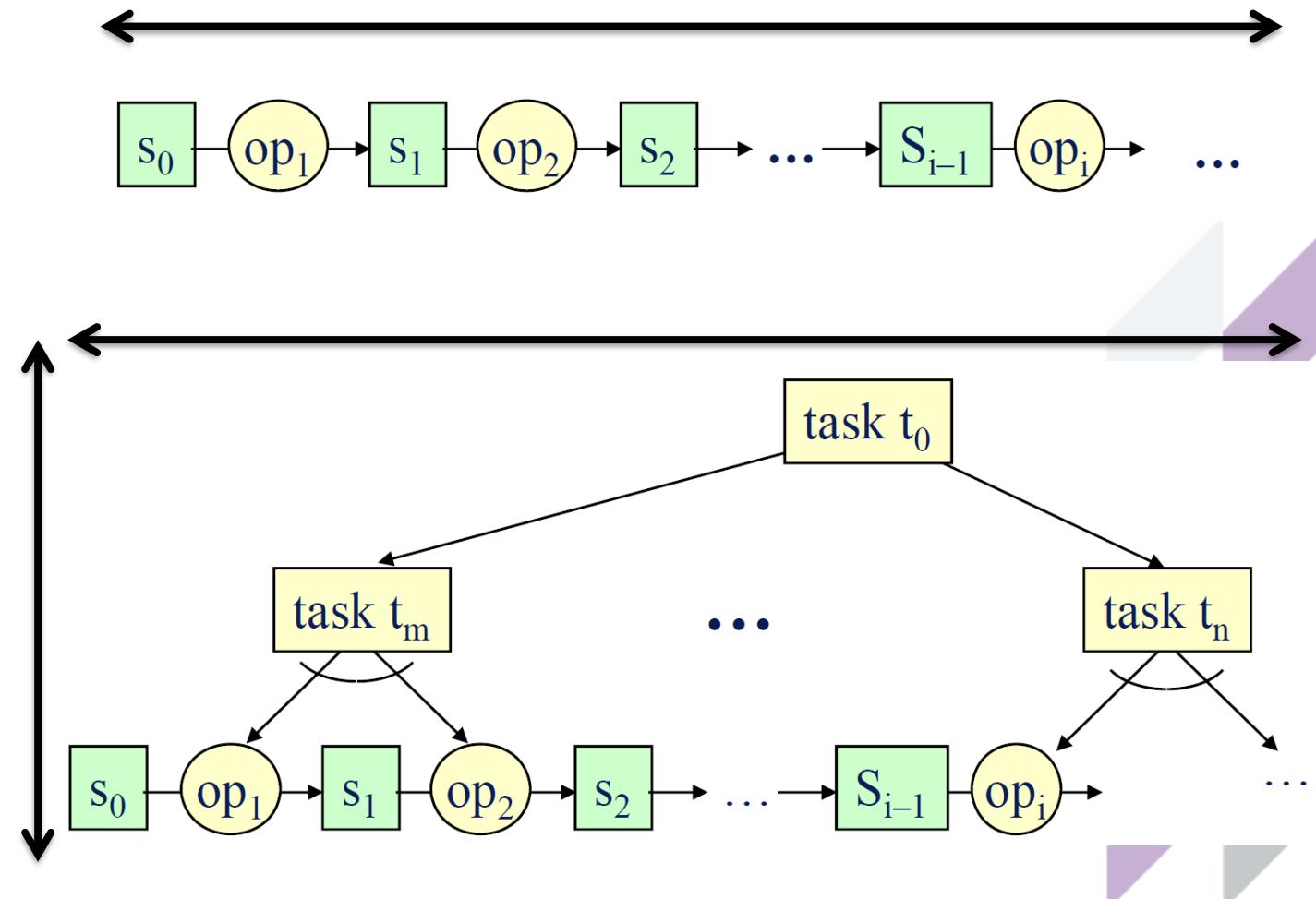
- **Move-Each-Twice()**

- Task:  $move - all - stacks()$
- Pre: *None*
- Subtasks:
  - $move - stack(p1a, p1b)$
  - $move - stack(p1b, p1c)$
  - $move - stack(p2a, p2b)$
  - $move - stack(p2b, p2c)$
  - $move - stack(p3a, p3b)$
  - $move - stack(p3b, p3c)$



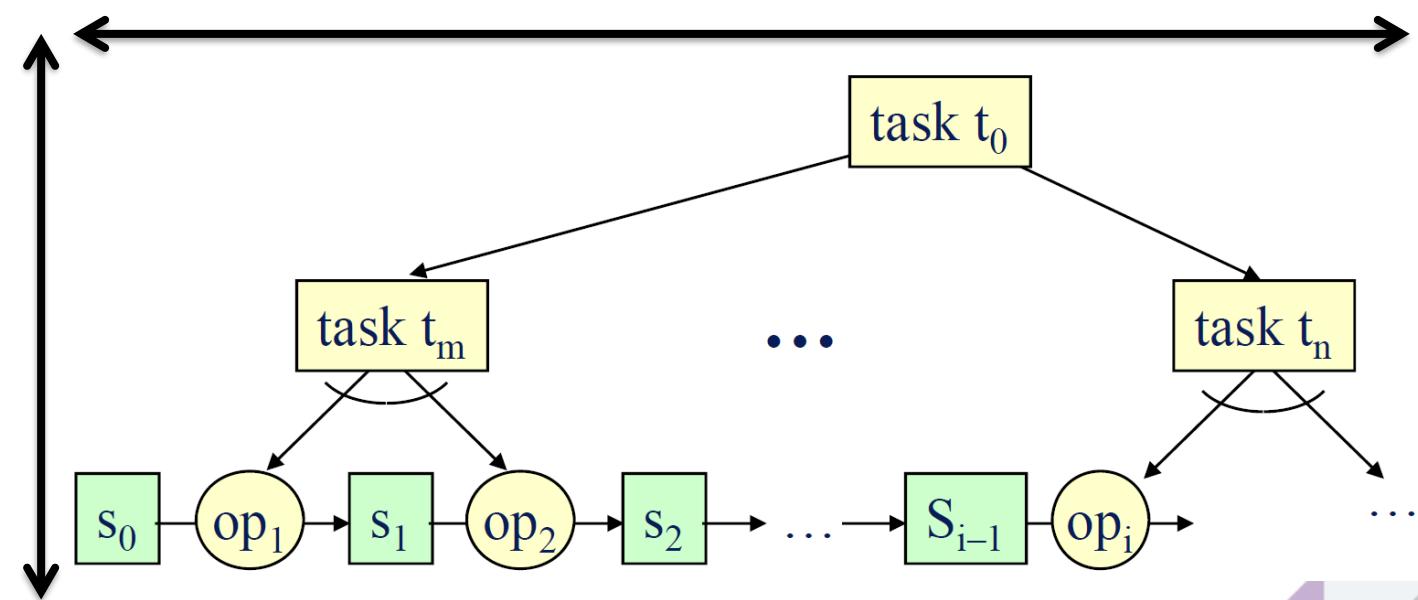
# Searching in HTN

- Unlike state-space planning, where you either search forward or backward, we have the option of searching in two directions in HTN:
  - Backward/Forward
    - Selecting tasks
  - Up/Down
    - Processing tasks into sub-tasks or action.



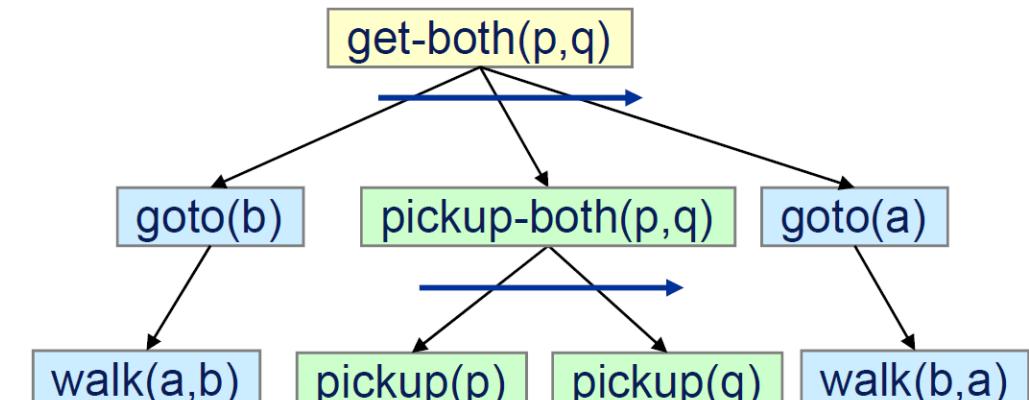
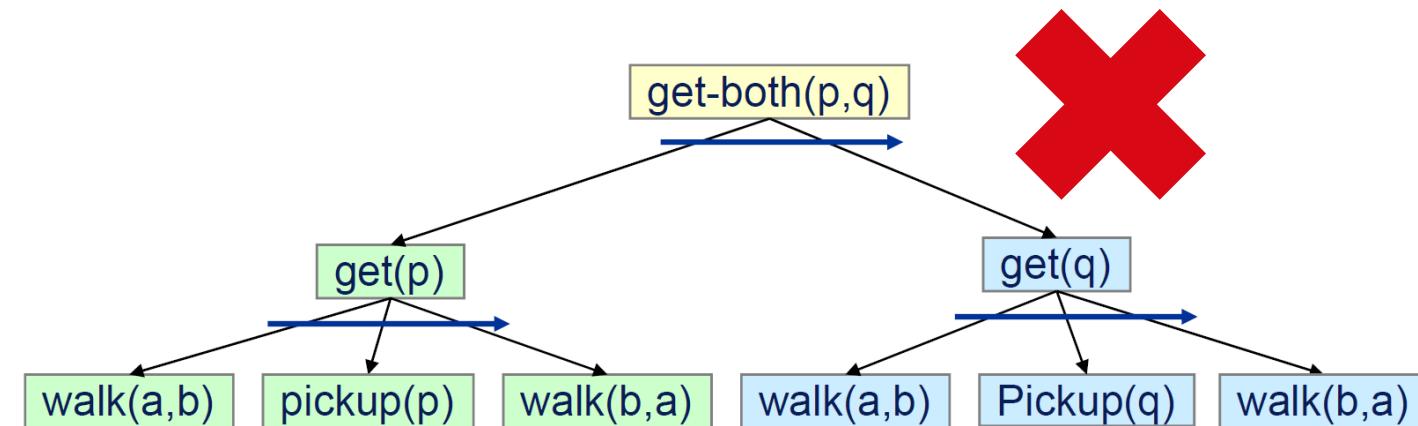
# Searching in HTN

- Important to ensure that when attempt to process a task, we know what the world state would be at this point.
- Hence in this instance, it's important that when a task is broken into subtasks, they're processed in order such that the world state is already planned in advance.



# Total vs Partial Ordering

- The simplest approach (STN planning) is reliant on ordering constraints.
- Forward search required given need to test for applicability of the state.
- Interweaving actions need to be written into a more concise formalism.
- That said, partial ordering is feasible, but requires a more complex algorithm to achieve it.



# STN vs HTN Planning

- HTN planning is a more general implementation of STN.
- Variety of formalisms and algorithms that are employed.
  - Forward Search – SHOP2
  - Plan-space Planning – O-Plan
    - Constraints can be associated with tasks and methods.
    - i.e. What facts are true at the before, during and after a task/method is executed.
- Use of goals/subgoals instead of tasks/subtasks.
  - GDP/GoDeL

# Classical Planning Non-Forward Search Introduction to HTN Planning

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Numeric Planning

# Planning with Numbers

- In the Scanalyzer application we saw:  
`(increase (total-cost) 3)`
- total-cost is a **state variable**
- We can create others:
  - Can have effects on them (already seen)
  - Can **condition** on them in the preconditions of actions.

# Numeric Variables in PDDL

```
(:predicates
  (at ?obj - locatable ?loc - location)
  (in ?obj1 - obj ?obj - truck)
  (driving ?d - driver ?v - truck)
  (link ?x ?y - location) (path ?x ?y - location)
  (empty ?v - truck)
)
(:functions (time-to-walk ?l1 ?l2 - location)
  (time-to-drive ?l1 ?l2 - location)
  (driven)
  (walked)
)
```

# Numeric Preconditions

- In principle, PDDL numeric preconditions comprise:
  - A comparison operator:  $>$ ,  $\geq$ ,  $=$ ,  $\leq$ ,  $<$ ; with...
  - A left- and right-hand side written using constants, the values of functions, and the operators  $+$ ,  $-$ ,  $*$ ,  $/$ .
- Examples:

```
(>= (fuel) (* 5 (distance ?from ?to)))
```

```
(<= (height truck) (height barrier1))
```

```
(>= (number-of-widgets) 3)
```

# Numeric Effects

- In principle, PDDL numeric effects comprise:
  - A variable to update
  - How to update it: assign (=), increase (+=), decrease (-=)
  - A right-hand-side formula, as in preconditions
- Examples:

```
(decrease (fuel) (* 5 (distance ?from ?to)))  
(increase (number-of-widgets) (capacity))  
(assign (battery-charge b) (max-charge b))
```

# Metric (Objective) Functions

- Now we have numeric state variables we can define a metric function for the planner to optimise.
- Examples:
  - (minimize (total-cost))
  - (minimize (+ (fuel-used) (\*2 (wages))))
- These metrics refer to the values of the variables after the plan has finished executing, i.e in the goal state.
- Defining a metric function is *optional* for a numeric planning problem.

# Planning with Numbers

- Search is straightforward these are discrete effects so:
  - We check numeric preconditions are satisfied before applying actions;
  - Update the values of the variables according to effects when we apply actions.
- But so far we've not seen anything about Heuristic Guidance.
- Metric relaxed planning graph, from Metric-FF (Hoffmann 2003).

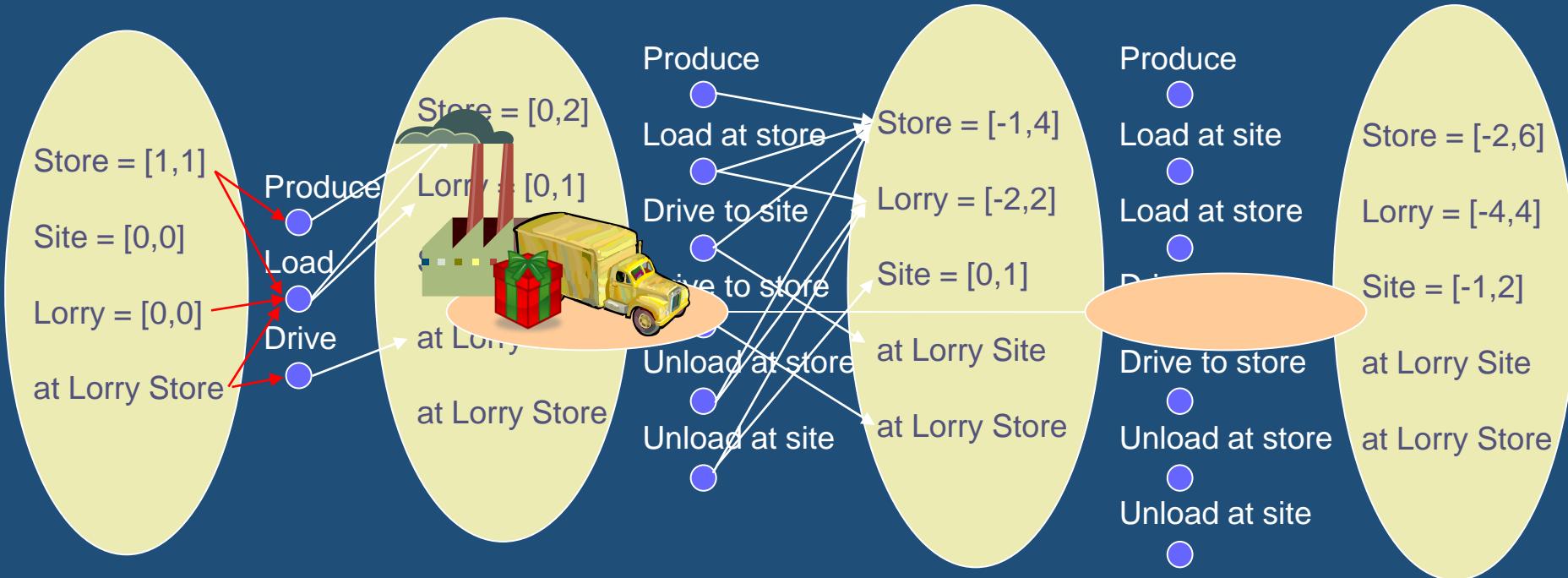
# Ignoring Numeric ‘Delete’ Effects

```
(:action move
  :parameters (?x - object ?from ?to -
location)
  :precondition (and
    (at ?x ?from)
    (>= (fuel ?x) 1)
  )
  :effect (and
    (at ?x ?to)
    (not (at ?x ?from))
    (decrease (fuel ?x) 1)
  )
)
```

# Relaxing Numeric Effects

- Is (decrease (fuel ?x) 1) a delete effect?
  - Conceptually yes, deletes a unit of fuel, needed to perform a move action;
  - Could ignore this in a relaxed problem.
- What about (decrease (undelivered-packages) 1) where the goal is (= (undelivered-packages) 0)?
  - Not a delete effect: we want fewer undelivered packages;
  - Ignoring this would make the problem unsolvable.
- How do we generalise relaxing numeric effects?
  - Maintain bounds: upper and lower bound on each numeric variable;
  - Use lower bound for preconditions  $v \leq c$  and upper bound for  $v \geq c$ .

# Bounds propagation



# Searching for Better Solutions

- Once a solution has been found we can continue searching;
- We can easily compute the cost of the plan so far, simply by evaluating the metric function in the current state.
- Assuming the cost monotonically increases as we add actions to the plan we can prune states for which  $\text{cost}(S) > \text{cost}^*$  where  $\text{cost}^*$  is the best cost we have found so far;
- Alternatively, we can just restart search, adding a goal ( $\text{cost} < \text{cost}^*$ ).
- Problem: heuristic is not necessarily providing guidance to find better solutions, it might keep sending us back to the same one.

# Classical Planning Optimal Planning with SAS+

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Classical Planning: Material Overview

- **Classical Planning: The fundamentals.**

- Non-Forward Search

- **Improving Search**

- Graphplan
- SAT Planning
- POP Planning
- HTN Planning

- **Optimal Planning**

- SAS+ Planning
- Pattern Databases

- **Planning Under Uncertainty**



# So Far...

- PDDL defines predicates for each domain
- A proposition is predicate with parameter bindings:
  - {(at truck1 city1)},
  - {(holding robot ball1)} , ...
- A proposition is either true or false, at any given point;
- One proposition exists for everything that could possibly be true.

# Propositions in Blocksworld

- Let's consider the propositions in a given state of BlocksWorld.

**$s(A\text{-on-}B) = T$**

**$s(A\text{-on-}C) = F$**

**$s(A\text{-on-table}) = F$**

**$s(B\text{-on-}A) = F$**

**$s(B\text{-on-}C) = F$**

**$s(B\text{-on-table}) = T$**

**$s(C\text{-on-}A) = F$**

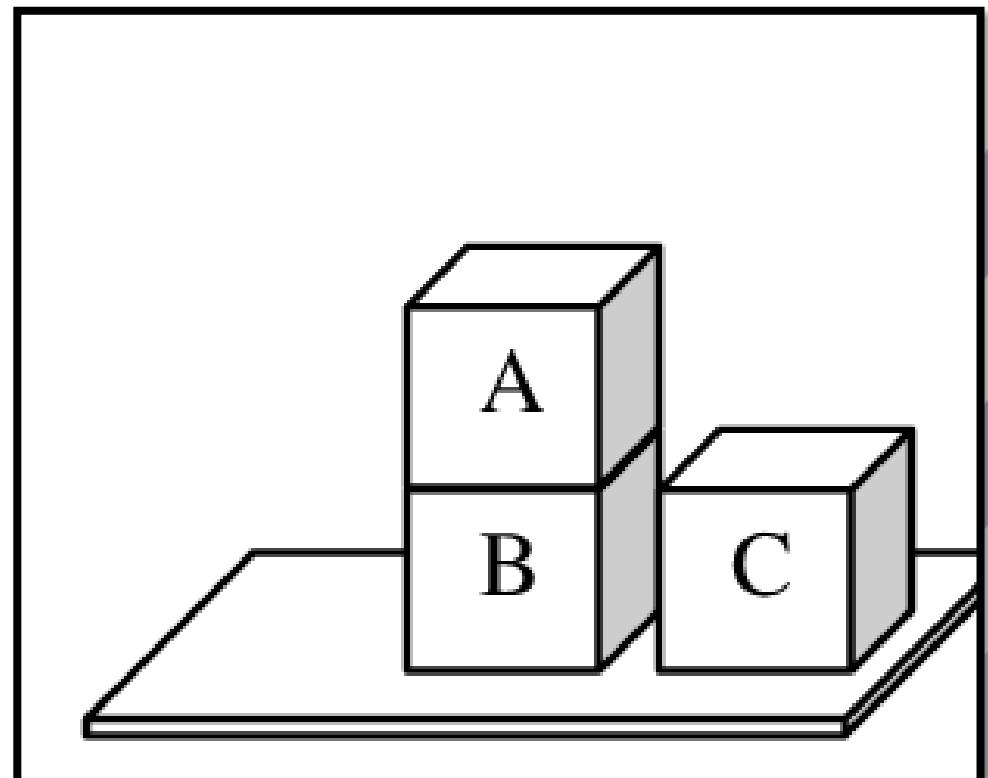
**$s(C\text{-on-}B) = F$**

**$s(C\text{-on-table}) = T$**

**$s(A\text{-clear}) = T$**

**$s(B\text{-clear}) = F$**

**$s(C\text{-clear}) = T$**



# Propositions in Blocksworld

- Let's consider the propositions in a given state of BlocksWorld.

$s(A\text{-on-}B) = T$

$s(A\text{-on-}C) = F$

$s(A\text{-on-table}) = F$

$s(B\text{-on-}A) = F$

$s(B\text{-on-}C) = F$

$s(B\text{-on-table}) = T$

$s(C\text{-on-}A) = F$

$s(C\text{-on-}B) = F$

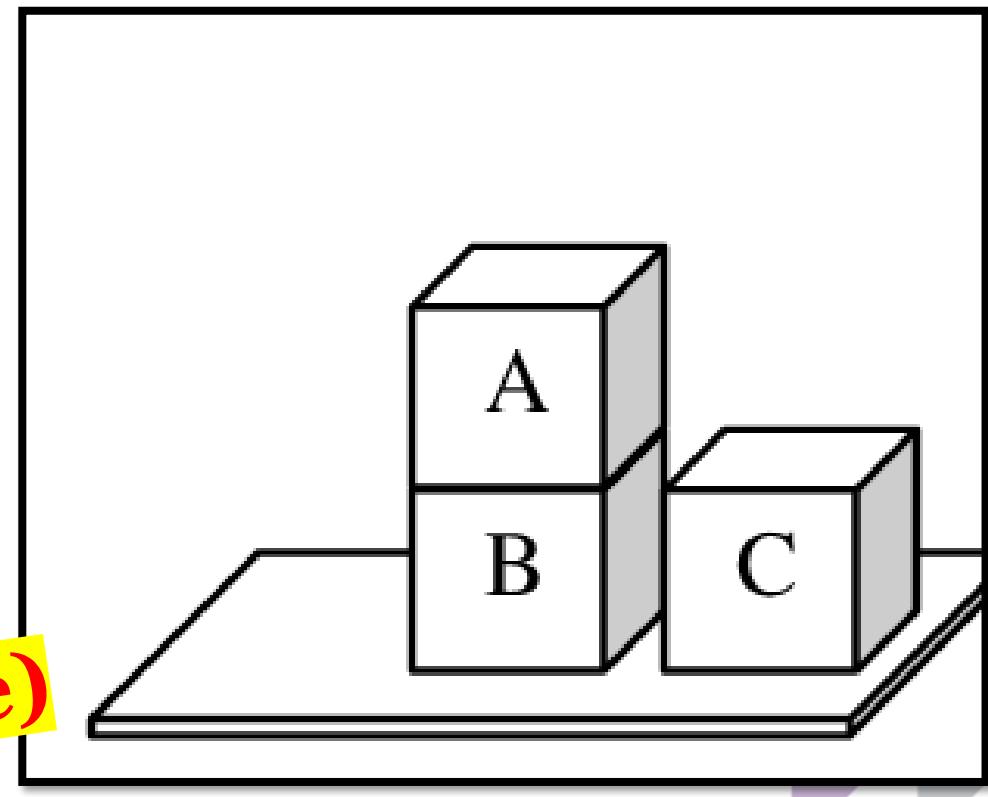
$s(C\text{-on-table}) = T$

$s(A\text{-clear}) = T$

$s(B\text{-clear}) = F$

$s(C\text{-clear}) = T$

$2^{12} = 4096$  states  
(not all reachable)



# Propositions in Blocksworld

- Let's consider the propositions in a given state of BlocksWorld.

$s(A\text{-on-}B) = T$

$s(A\text{-on-}C) = F$

$s(A\text{-on-table}) = F$

$s(B\text{-on-}A) = F$

$s(B\text{-on-}C) = F$

**$s(B\text{-on-table}) = T$**

$s(C\text{-on-}A) = F$

$s(C\text{-on-}B) = F$

**$s(C\text{-on-table}) = T$**

**$s(A\text{-clear}) = T$**

$s(B\text{-clear}) = F$

**$s(C\text{-clear}) = T$**

**$S(A\text{-on-}D) = F$**

**$S(B\text{-on-}D) = F$**

**$S(C\text{-on-}D) = F$**

**$S(D\text{-on-}A) = F$**

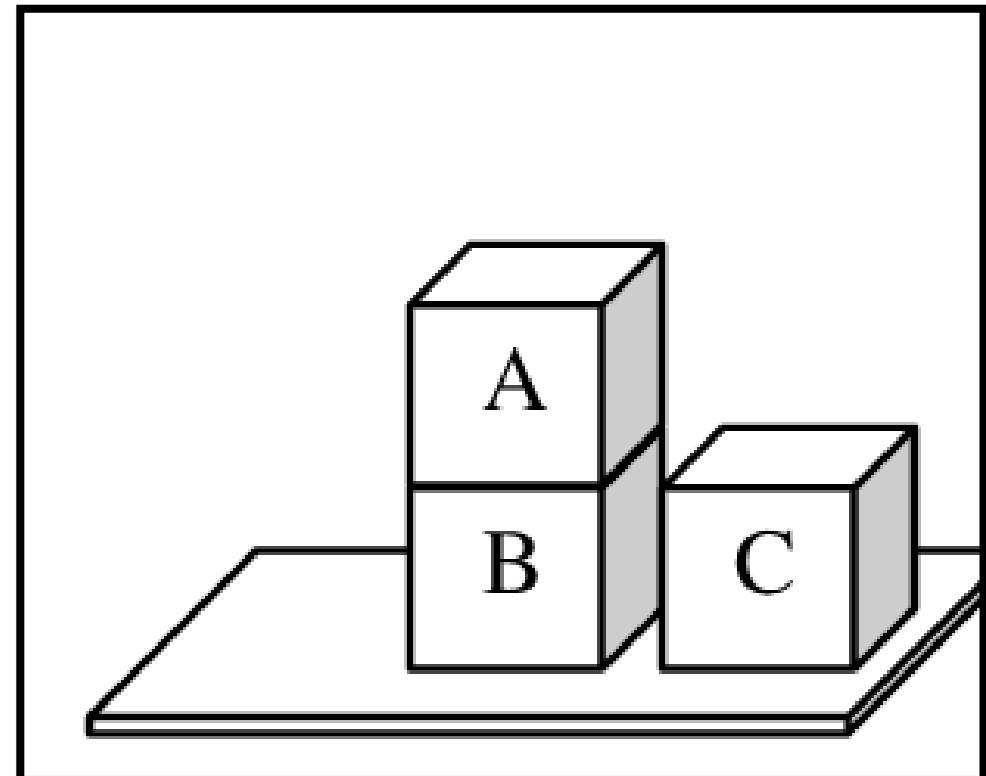
**$S(D\text{-on-}B) = F$**

**$S(D\text{-on-}C) = F$**

**$s(D\text{-on-table}) = T$**

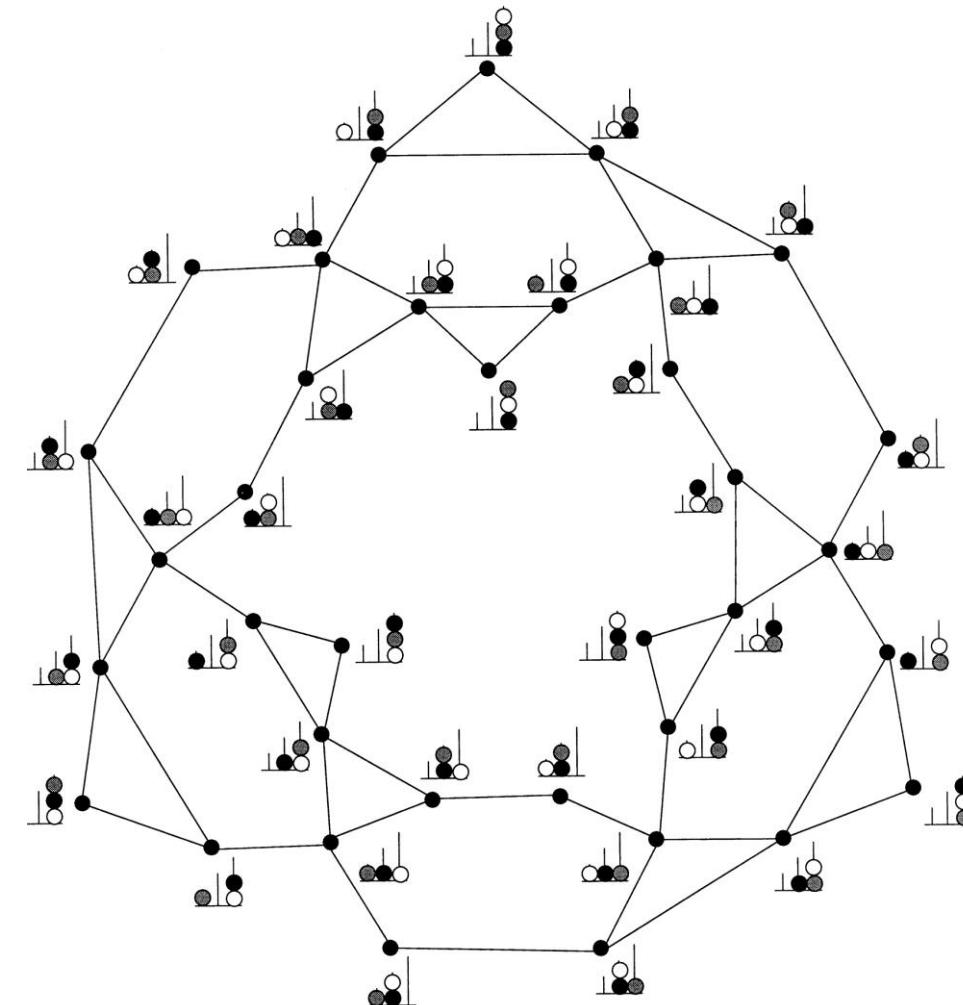
**$s(D\text{-clear}) = T$**

**$2^{20} = 1,048,576$  states  
(not all reachable)**



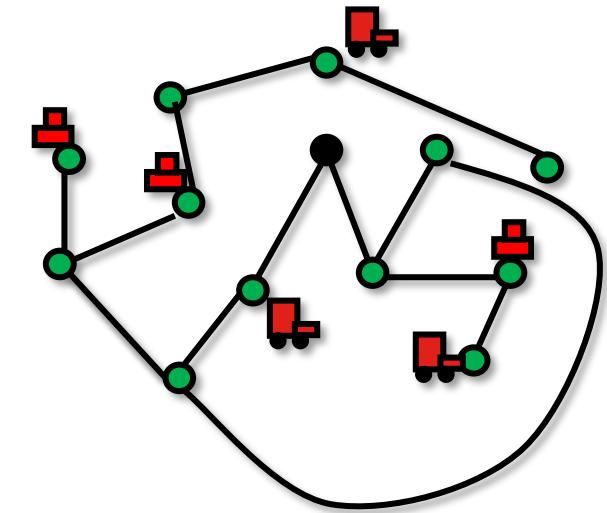
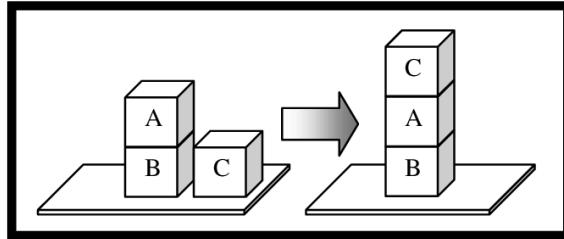
# The Perils of Expressivity

- By having such an expressive formalism, state spaces can begin to explode.
- How do we circumvent this?
  - Use heuristics to guide the search?
  - Find landmarks to give us sub-goals to solve?
  - Use local search (see EHC)?
- Often sacrificing completeness of search in an effort to find a solution quickly.
  - Hence Best-First Search fallback in EHC.
- What if we instead sacrifice expressiveness for efficiency?



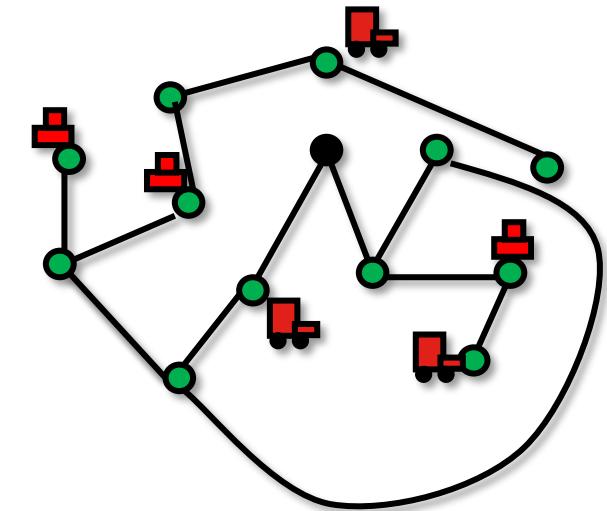
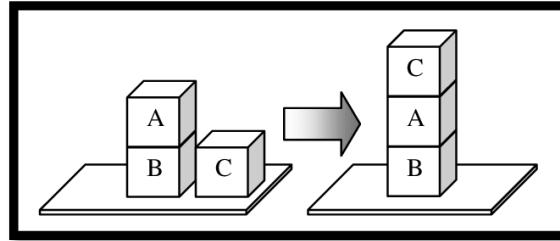
# Invariants

- When humans reason about planning tasks, we implicitly make use of 'obvious' properties and relationships between propositions.
  - E.g. A block can't be atop another block and on the table.
  - E.g. A truck can't be in two places at once.
- Despite this, we still need to ensure PDDL captures this knowledge (add + delete effects).
- Invariants: logical formulas  $\varphi$  that are true in all reachable states.
  - Example:  $\phi = \neg(at\_uni \wedge at\_home)$



# Finding Invariants

- Theoretically, testing whether an invariant is valid is as hard as planning itself.
- But, if we can find invariants and encode it in a way a planner can understand, we could potentially reduce the state space.
- Even finding ‘obvious’ invariants could drastically improve our performance in solving the planning task.



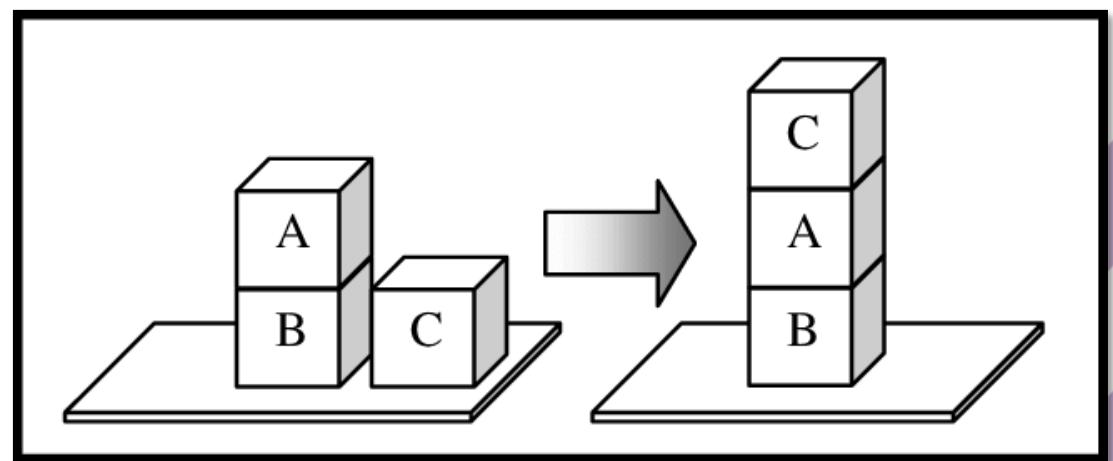
# Mutual Exclusion

- Mutual Exclusion (**mutex**): an invariant over a set of propositions that states only can be true at any point in time.
- $\phi = \neg(at - uni \wedge at - home)$  is a mutex
- This can also extend to sets of propositions, which are mutex if every subset of two is mutex.
- Let's look at some examples...



# Mutual Exclusive: Example

- **BlocksWorld Invariant:**
  - $\neg(ConA) \vee \neg(ConB)$
  - *ConA* and *ConB* are mutex.
- **But also**
  - $\{AonB, ConB, Bclear\}$  is mutex.
  - Because every pair in this set are mutex.



# Mutual Exclusive: Example

- **BlocksWorld Invariant:**

- $\neg(ConA) \vee \neg(ConB)$
- *ConA* and *ConB* are mutex.

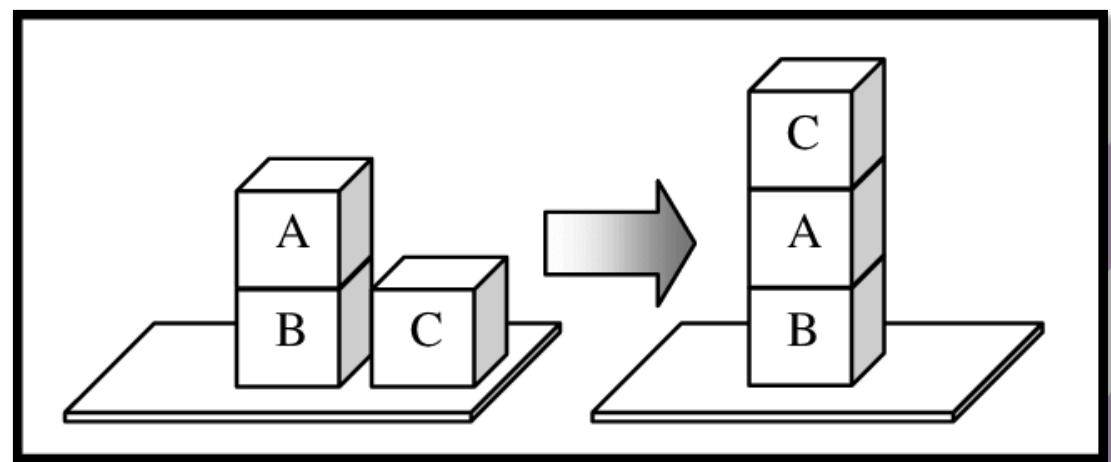
- **But also**

- $\{AonB, ConB, Bclear\}$  is mutex.
- Because every pair in this set are mutex.

- **This makes sense...**

- It's impossible for both A and C to be on B at the same time. And for B to be clear when something is atop it.

- **How does this impact state space size?**



**$\{AonB, ConB\}$**   
 **$\{AonB, Bclear\}$**   
 **$\{ConB, Bclear\}$**

# Finite-Domain State Variables

- We can use these literals to reframe a problem using Finite-Domain State Variables
- We create a variable  $v$  and a domain  $\text{dom}(v)$  that expresses the range of possible values that can be assigned to  $v$ .
- We can then create Finite Domain States that express a state of a planning problem as an assignment to all variables in the state  $V$ .

$$s(v) \in \text{dom}(v) \quad \forall v \in V$$

# Finite-Domain State Variables

- We can use these literals to reframe a problem using Finite-Domain State Variables
- We create a variable  $v$  and a domain  $\text{dom}(v)$  that expresses the range of possible values that can be assigned to  $v$ .
- We can then create Finite Domain States that express a state of a planning problem as an assignment to all variables in the state  $V$ .

$$s(v) \in \text{dom}(v) \quad \forall v \in V$$

- E.g.  $v = \text{below-a}$ ,  $\text{dom}(\text{below-a}) = \{b, c, \text{table}\}$

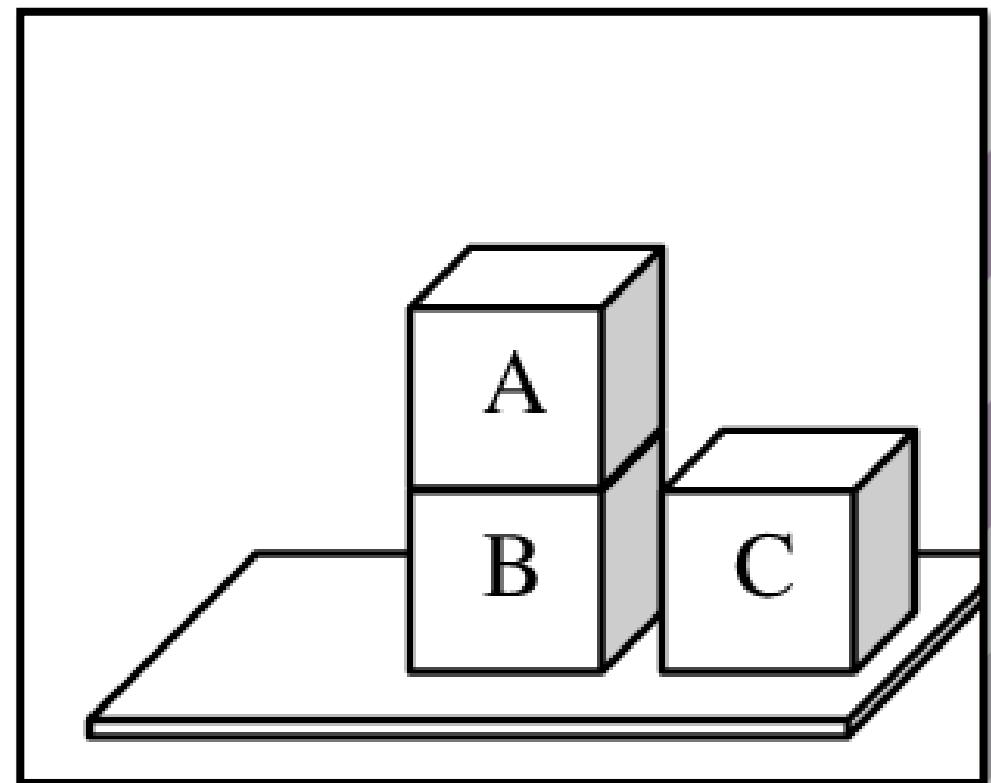
# Blocksworld using Finite-Domain Representation

- Let's revisit BlocksWorld, but this time using propositions and Finite-Domain State Variables.

**below-a: {b, c, table} = b**

**below-b: {a, c, table} = table**

**below-c: {a, b, table} = table**



# Blocksworld using Finite-Domain Representation

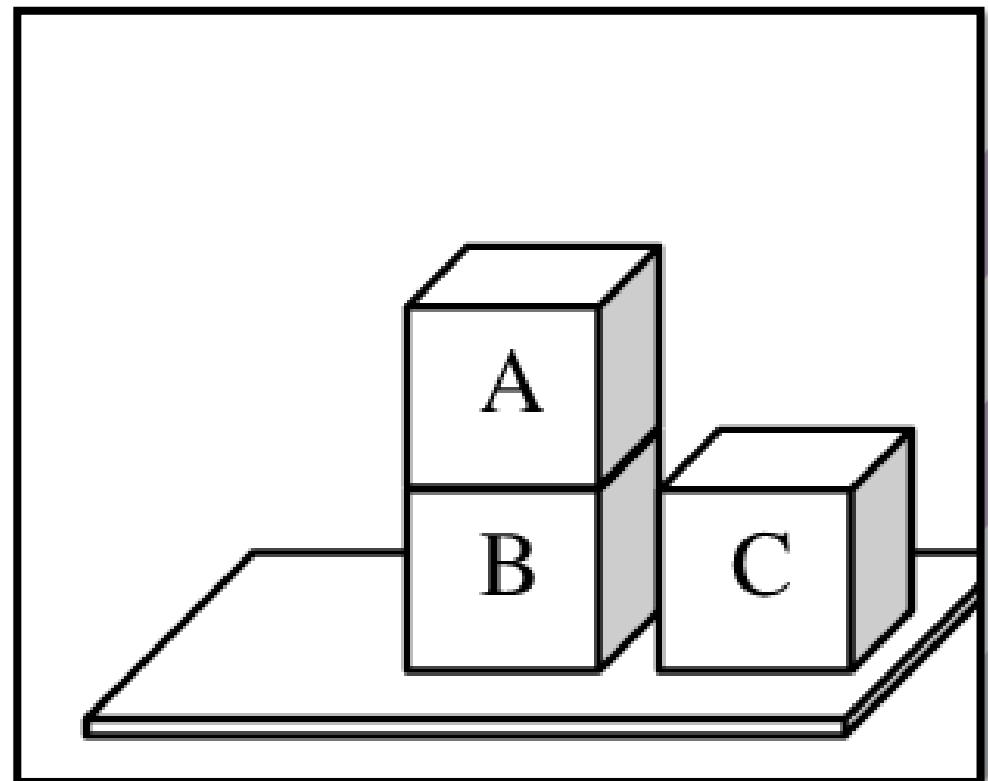
- Let's revisit BlocksWorld, but this time using Finite-Domain State Variables.

**below-a: {b, c, table} = b**

**below-b: {a, c, table} = table**

**below-c: {a, b, table} = table**

$$3^3 = 27 \text{ states}$$



# Blocksworld using Finite-Domain Representation

- Let's revisit BlocksWorld, but this time using Finite-Domain State Variables.

**below-a:** {b, c, table} = b

**below-b:** {a, c, table} = table

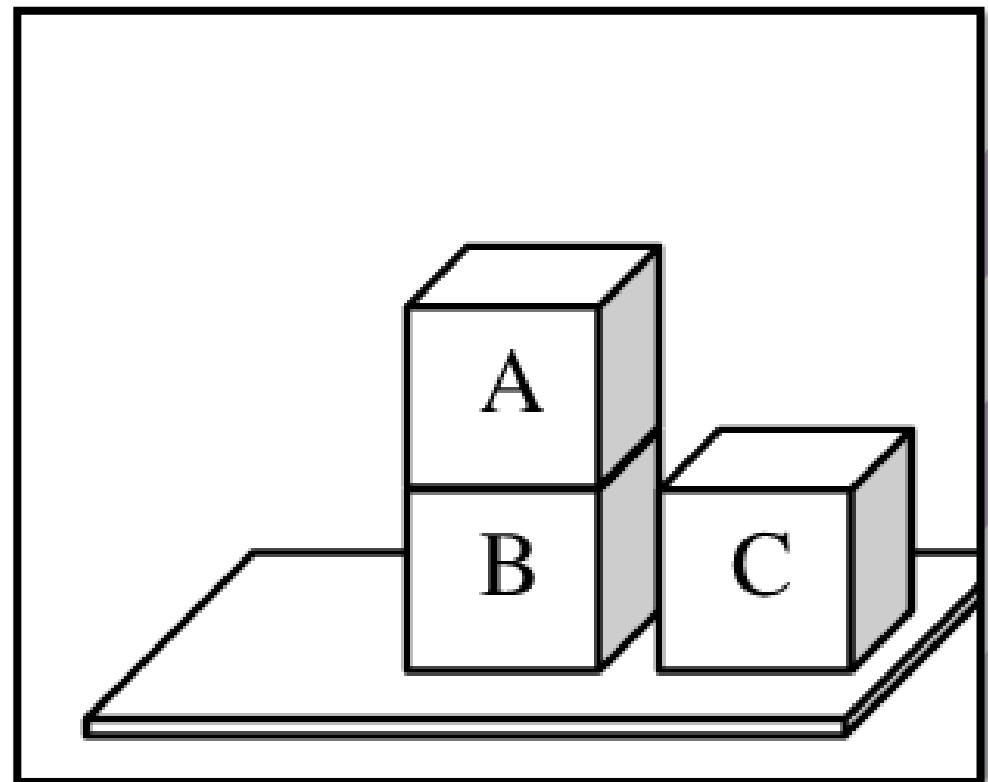
**below-c:** {a, b, table} = table

**above-a:** {b, c, nothing} = nothing

**above-b:** {a, c, nothing} = a

**above-c:** {a, b, nothing} = nothing

**6<sup>3</sup> = 216 states**



# Finite-Domain Formula

- Given the Finite-Domain Representation (FDR) encoding, this is valuable because we can translate it back to the normal propositions, but safely encapsulating the mutex relationships we have discovered.
- Example  
 $(\text{above-}a = \text{ nothing}) \vee \neg (\text{below-}b = c)$
- Corresponds to...  
 $A\text{-clear} \vee \neg B\text{-on-}C$

# Planning as Satisfiability (SAS+)

- Adopt the intuition from finite-state variables into our planning process.
- To do this, we need to encode the planning task as a Boolean Satisfiability problem (SAT), through use of the propositions we've seen earlier.
- We then introduce the FDR encodings transforming the SAT problem into a SAS+ Planning problem.
- This process was popularised by the Fast Downward Planner, which translates PDDL problem encodings into FDR.



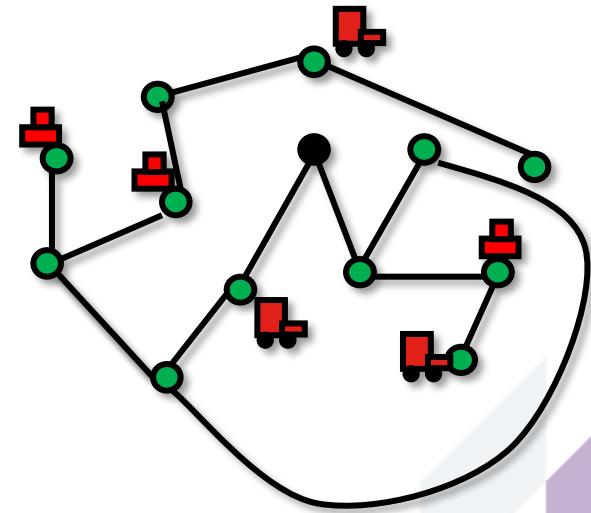
Helmert, M., 2006. **The fast downward planning system.**  
*Journal of Artificial Intelligence Research*, 26, pp.191-246.

# SAS+ More Formally

- An SAS+ planning task is a 4-tuple  $\Pi = \langle V, O, s_0, s^* \rangle$  with the following components:
  - $V = \{v1, \dots, vn\}$   
A set of state variables (or fluents) **V**, each with an associated finite domain  $D_v$ .  
If  $d \in D_v$  we call the pair  $v = d$  an atom.
  - $O$  = a set of operators  
Where an operator is a triple  $\langle name, pre, eff \rangle$ 
    - **Name** of the operator
    - **Pre** and **eff** are partial variable assignments (preconditions and effects).
  - $s_0$  is the initial state, and  $s^*$  is a partial assignment of atoms representing the goal.

# Operators (Actions)

- **Actions look a little different under SAS+:**
  - Two sorts of conditions roll PDDL preconditions and effects together.
- **prevail conditions:**
  - Variable = Value that stays the same;
  - When loading a package **p1** onto a truck **t1** at location **l1**: **<t1,at-l1>**
- **pre\_post conditions:**
  - The value of a variable changes from one value to another:
  - When loading a package **p1** onto a truck **t1** at location **l1**: **<p1,at-l1,in-t1>**



# Domain Transition Graphs (DTGs)

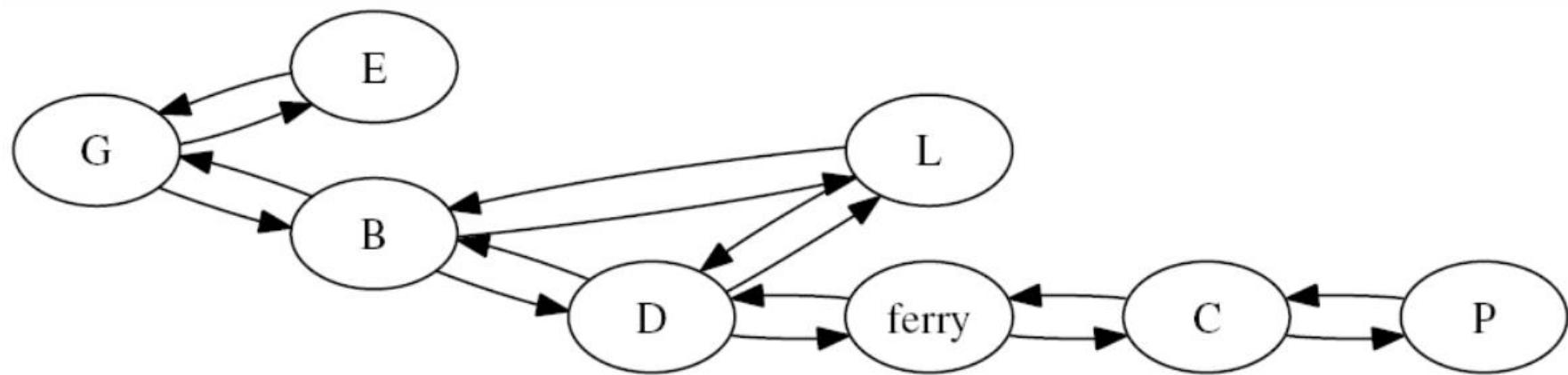
- **Domain:** One for each variable, referring to its domain;
  - **Transition:** pre\_post conditions of operators change the values of a variable;
  - **Graph:** Transitions in the domain form a directed graph.
- 
- **Nodes:**
    - The domain values.
  - **Edges:**
    - An edge exists from A→B iff an operator exists with a **pre\_post** condition  $\langle v, A, B \rangle$



# Interesting DTGs

- **Inter-continental logistics: a truck which can drive itself (on land) or use a ferry.**
- **Actions: Drive the truck?**
  - PDDL action: **drive ?truck ?a ?b ;**
  - Analysis will detect **truck can only be in once place at once.**
- **Action: Load the truck onto the ferry**
  - PDDL action: **board-ferry ?truck ?ferry ?p**
  - Deletes **(at ?truck ?p)** but gives **(on ?ferry ?truck).**
- **Similarly: disembark from ferry.**
- **In both of these cases: we are deleting one thing to give another, so being on ferry is mutually exclusive with being at a location.**

# Interesting DTGs



Result in SAS+: is a single variable for the truck's status corresponding to both at and on in the original PDDL.

# Interesting DTGs

- Shuffle the nodes around and we get...



# Summary

- There are other ways to model planning SAS+ is one of them.
- This doesn't magically solve planning for us, given it can only be used in very specific contexts (conditional effects are not possible).
- But it shows we can optimise planning by approaching the problem in different ways.



# Classical Planning Optimal Planning with SAS+

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



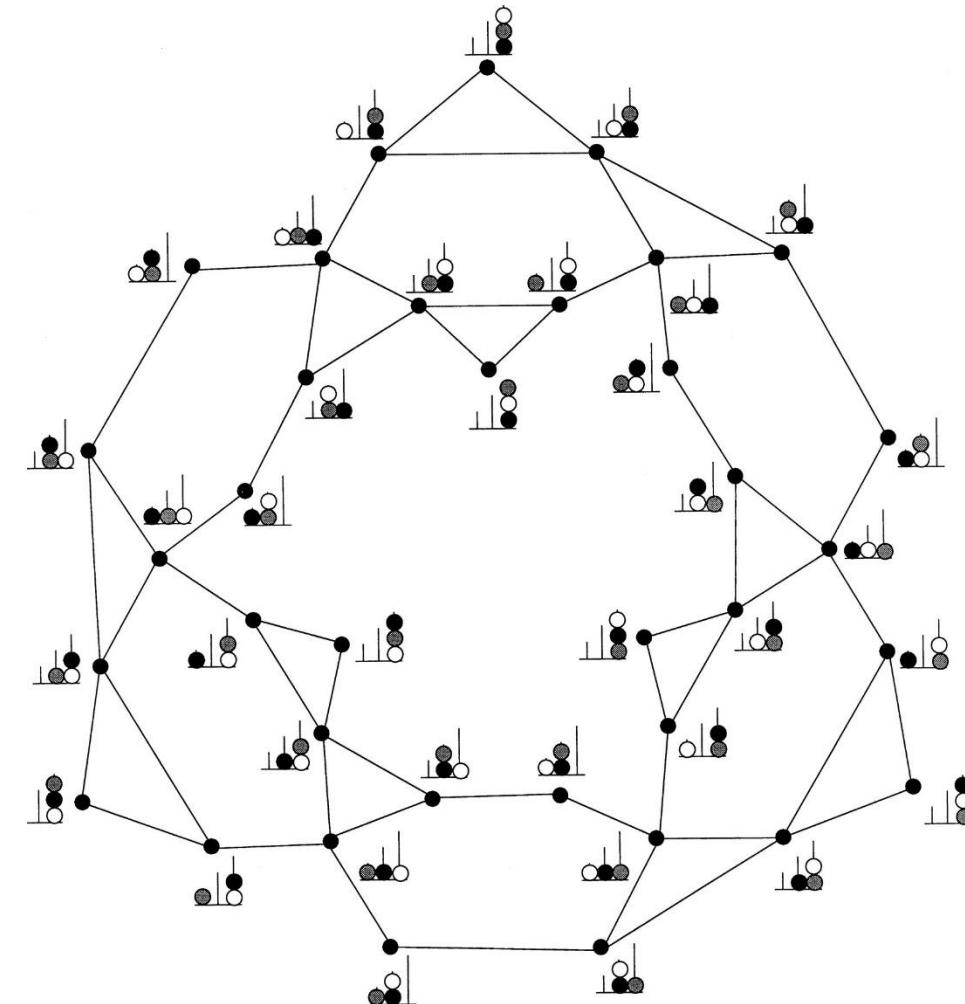
# Classical Planning Pattern Databases

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



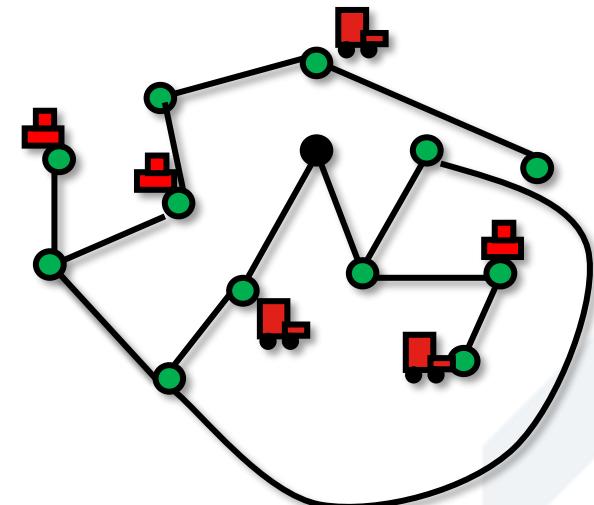
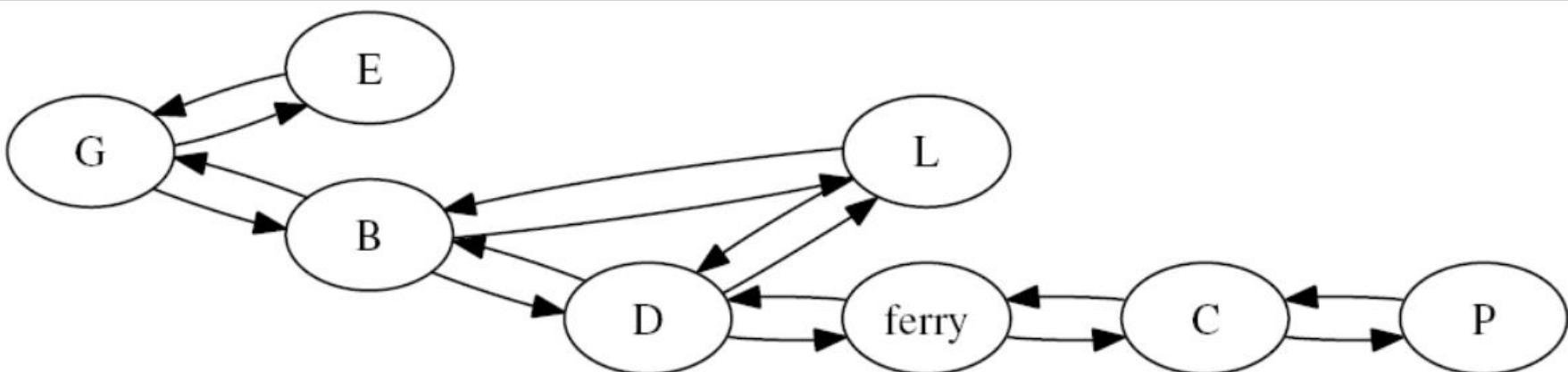
# Recap: The Perils of Expressivity

- By having such an expressive formalism, state spaces can begin to explode.
- How do we circumvent this?
  - Use heuristics to guide the search?
  - Find landmarks to give us sub-goals to solve?
  - Use local search (see EHC)?
- Often sacrificing completeness of search in an effort to find a solution quickly.
  - Hence Best-First Search fallback in EHC.
- What if we instead sacrifice expressiveness for efficiency?



# Recap: SAS+ Planning

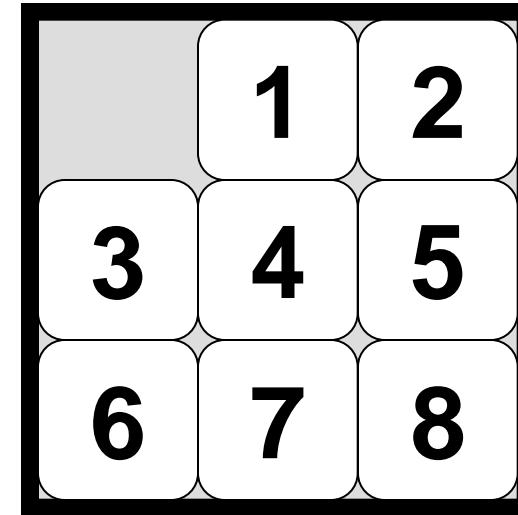
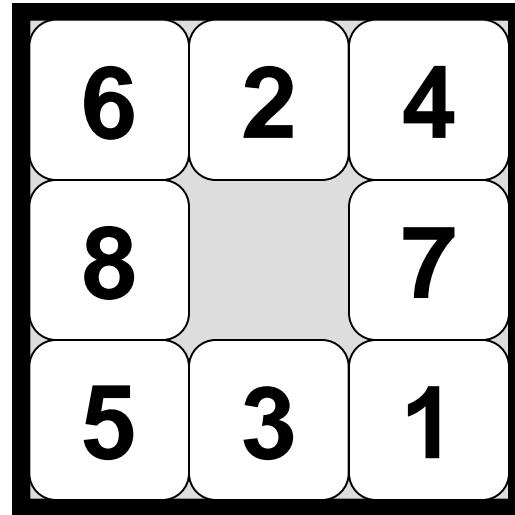
- Optimising planning by analysing the problem.
- SAS+ Planning re-encodes existing problem into new formulation.
- Use of DTG's help us visualise assignment of variables to values during search/execution.



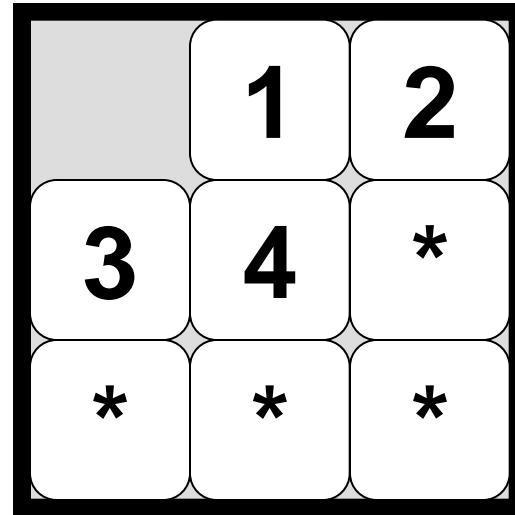
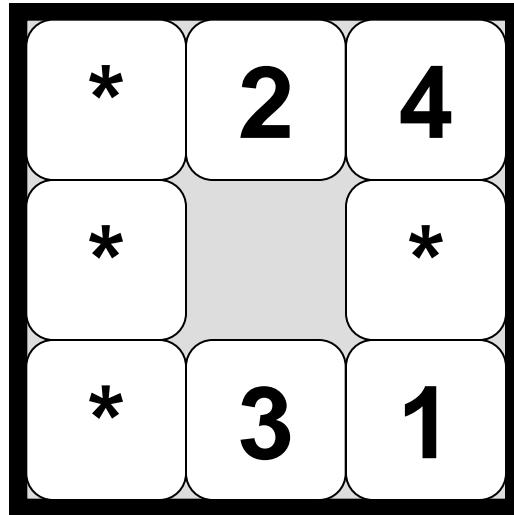
# Pattern Databases

- **Problem: Planning is hard and expensive.**
  - We want to find a way to speed up our **exploration** of the search space.
- **Solution: Store solution costs for sub-problems within the search space.**
- **Why?**
  - **Solution costs to sub-problems** will provide an **admissible heuristic** to solve the actual problem.
  - Heuristic calculation becomes **very fast** (database lookup).
- **How?:**
  - **Searching backwards** from the goal, record **costs of states**.
  - An **expensive** albeit **one-time** calculation.

# Why Heuristics for Sub-Problems?



# Sub-Problems and Heuristics

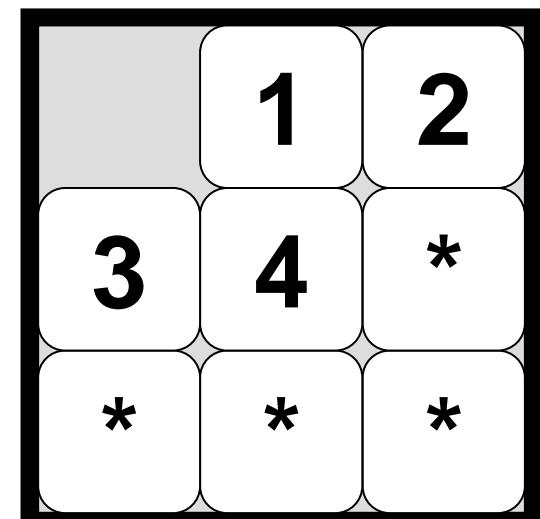
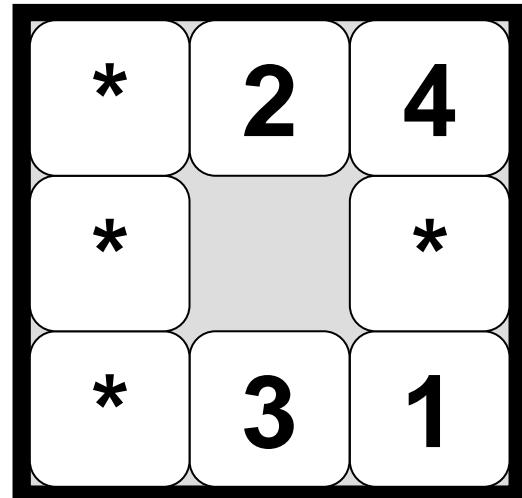


- Sub-Problem: Removing some complexity of the original problem.
- Solution cost will be less than or equal to the optimal solution of the original.

# Patterns

- **Partial specification of a given state.**
  - Information is incomplete, abstraction of original state.
- **Target Pattern: partial specification of the goal state.**
- **Establish a Pattern Database (PDB)**
  - Set of all permutations of the target pattern.
- **Calculate pattern cost.**
  - Compute distance to target pattern.
  - Minimum number of actions to achieve target pattern.

Pattern



Target Pattern

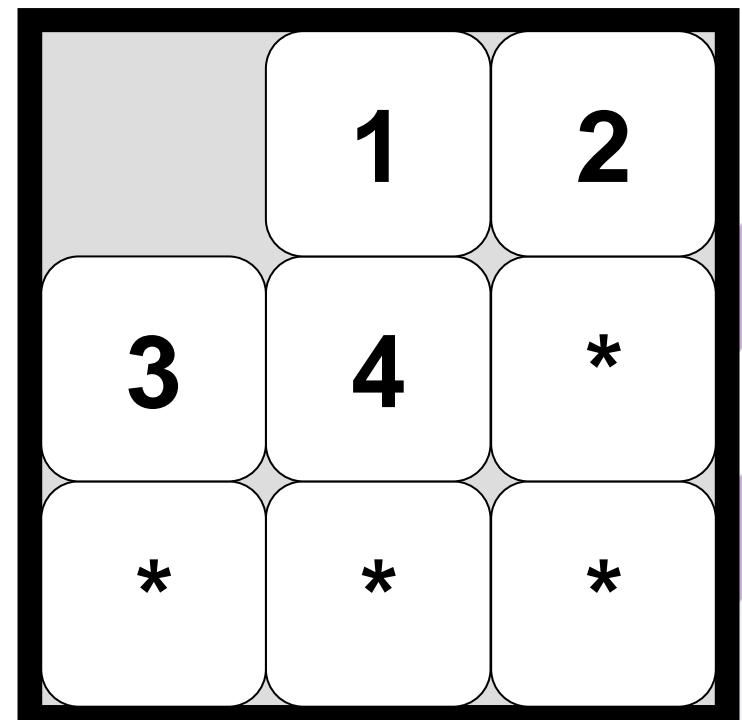
# Pattern Discovery

- How are patterns discovered in a given problem?
- We search the ‘abstraction’ of the state space.
- Search is achieved using breadth-first-search, moving backwards from the goal state.
  - Actions applied are the inverse of the original.
- Pattern cost = search-tree depth from  $s_{goal}$  to  $s_{init}$



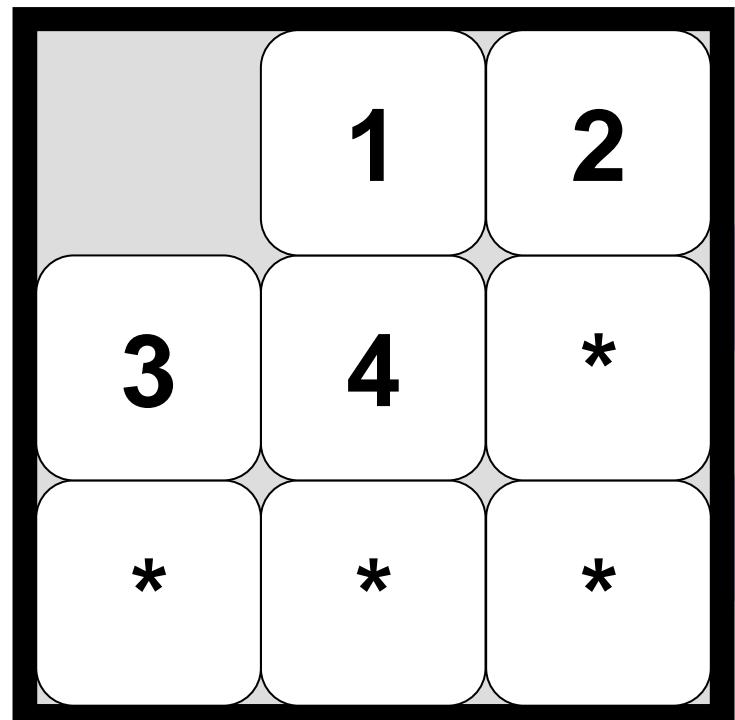
# Abstractions

- **Pattern Discovery explores the ‘abstract’ state space.**
- **Our target patterns exist within the abstract state space.**
- **We only care about specific values in any given state, so the number of unique states is now smaller.**
  - We have already abstracted the state space, to only the values visible in the target pattern.
- **But how does this work and what does the abstract state space look like?**

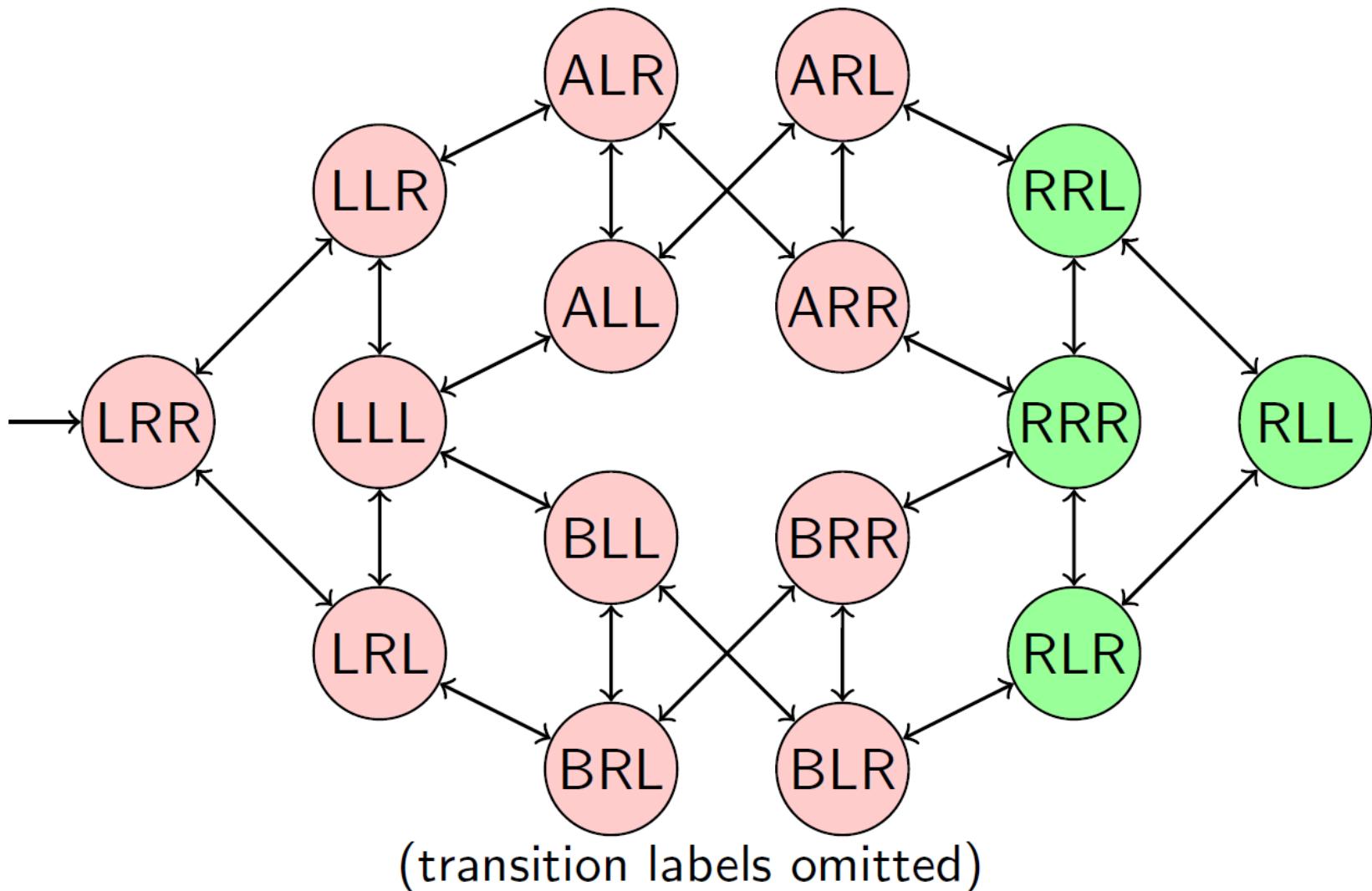


# Abstractions

- **Function  $\alpha(S) = S'$  applied on a state  $S$ , transforming it into abstract state  $S'$** 
  - Function (equivalence relation) helps to remove distinctions between states in abstract space.
  - Hence this may result in  $\alpha(S_1) = \alpha(S_2)$
- **Maps state space into a smaller state space.**
  - In **abstract state transition system**, similar states are now indistinguishable.
- **Abstraction Heuristic: Heuristic estimate is cost-to-goal in the abstract transition system.**

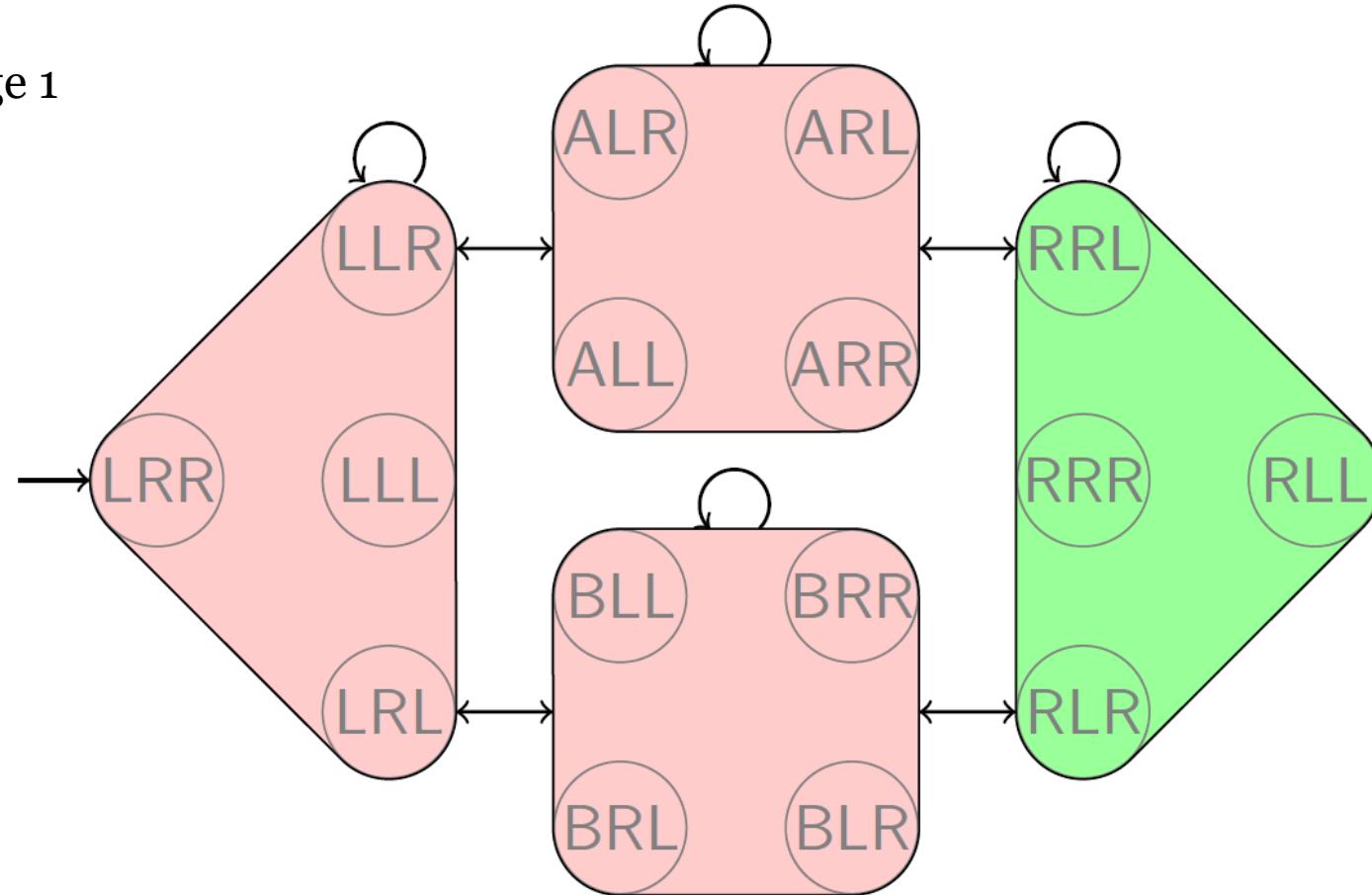


# Abstractions (Example)



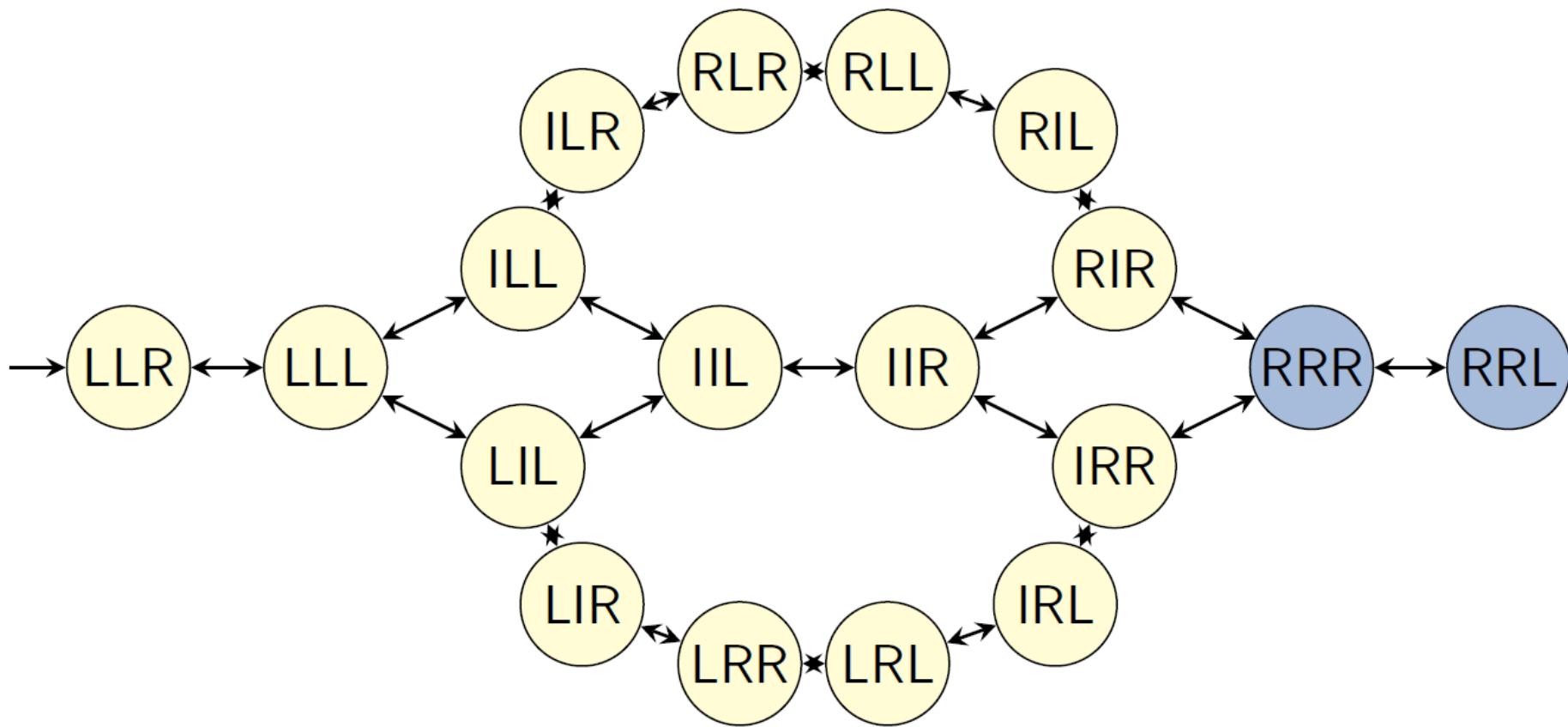
# Abstractions (Example)

$\alpha$  = location of package 1

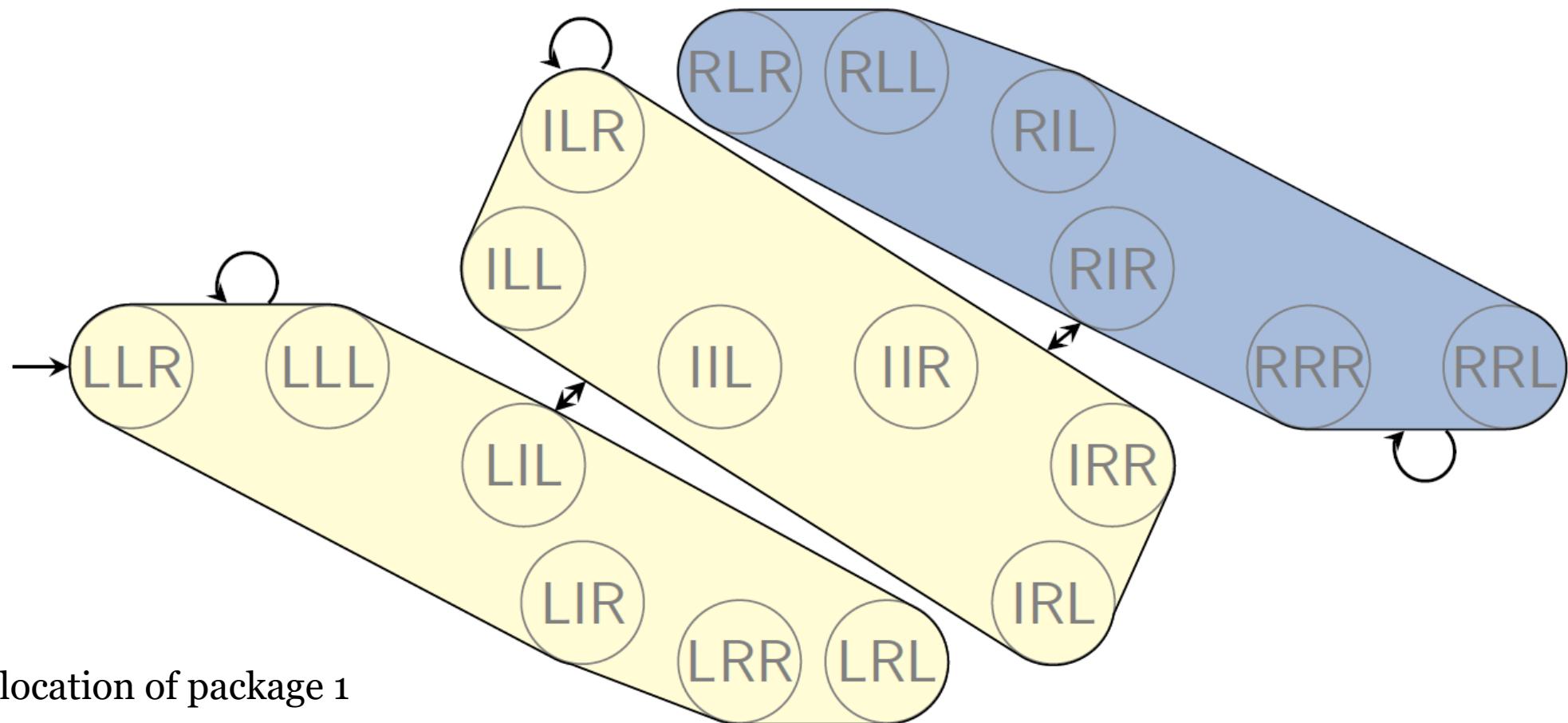


$$\rightsquigarrow h^\alpha(s_0) = 2$$

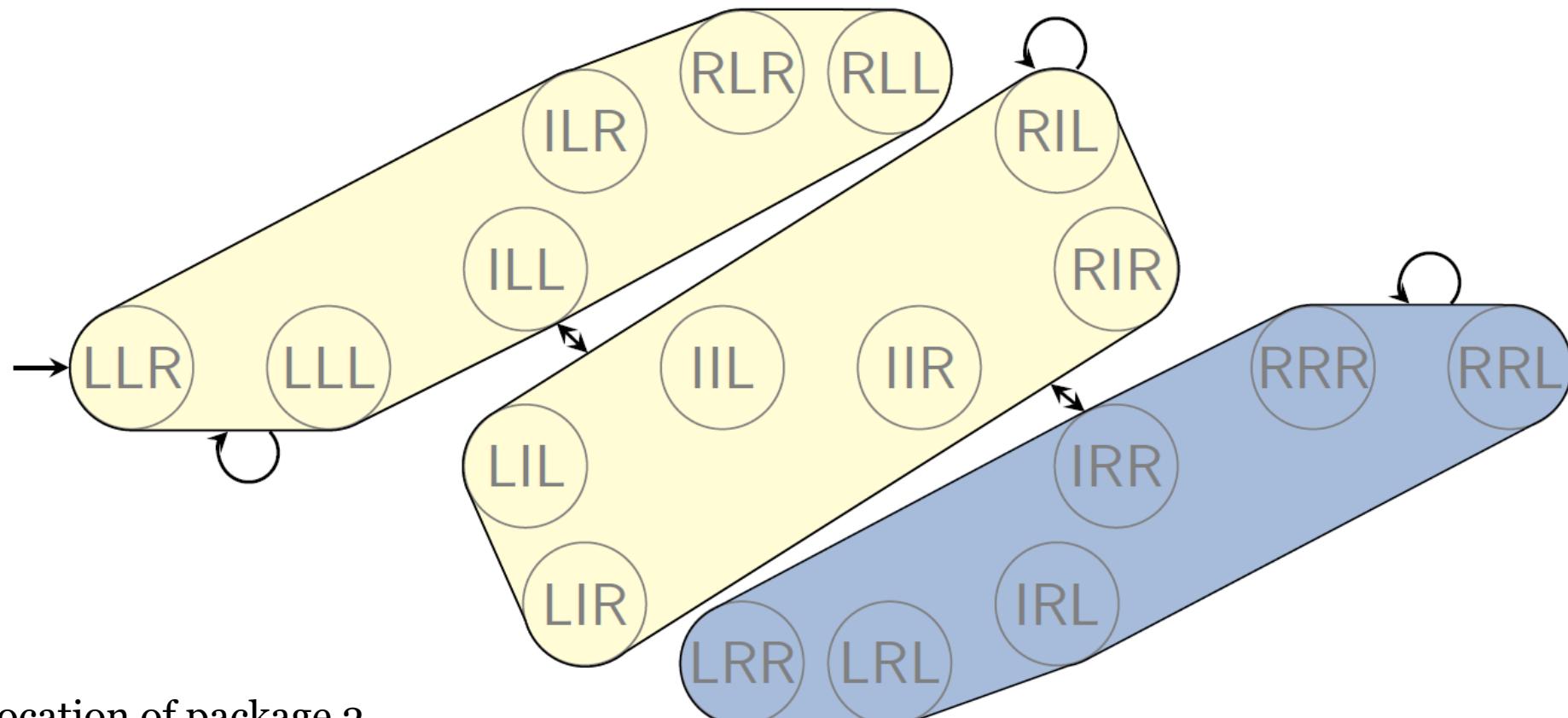
# Abstractions (Example #2)



# Abstractions (Example #2)



# Abstractions (Example #2)



# Planning with Pattern Databases

- **Create a set of all state propositions in mutex groups**
  - These are the variables we're interested in.
- **Construct abstract problem space(s).**
- **Construct Pattern Database**
- **Run planning process from initial state.**

# Pattern Database Planning for BlocksWorld

- Initial State

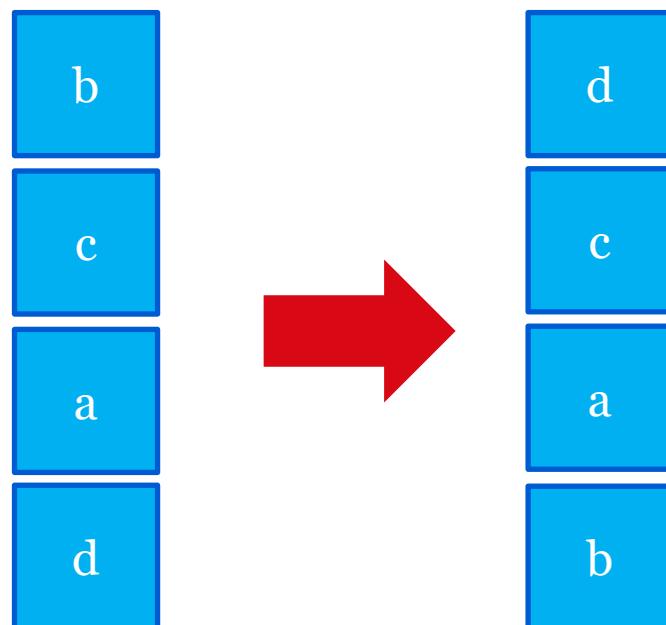
(clear b)  
(ontable d)  
(on b c)  
(on c a)  
(on a d)

(pick-up a)

P = (clear a), (ontable a), (handempty);  
A = (holding a); and  
D = (ontable a), (clear a), (handempty)

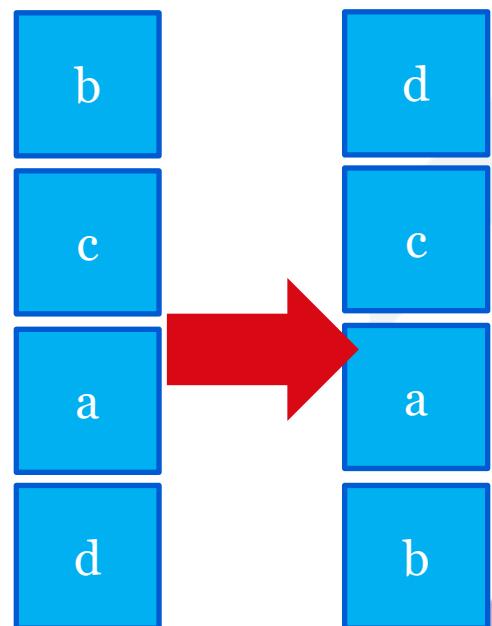
- Goal State

(on d c)  
(on c a)  
(on a b)



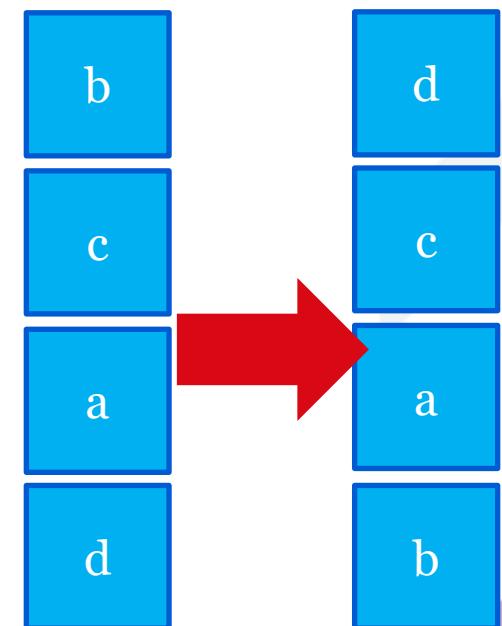
# SAS+ Encoding for BlocksWorld

- $G_1 = \{(on\ c\ a), (on\ d\ a), (on\ b\ a), (clear\ a), (holding\ a)\},$
- $G_2 = \{(on\ a\ c), (on\ d\ c), (on\ b\ c), (clear\ c), (holding\ c)\},$
- $G_3 = \{(on\ a\ d), (on\ c\ d), (on\ b\ d), (clear\ d), (holding\ d)\},$
- $G_4 = \{(on\ a\ b), (on\ c\ b), (on\ d\ b), (clear\ b), (holding\ b)\},$
- $G_5 = \{(ontable\ a), true\},$
- $G_6 = \{(ontable\ c), true\},$
- $G_7 = \{(ontable\ d), true\},$
- $G_8 = \{(ontable\ b), true\},$  and
- $G_9 = \{(handempty), true\},$



# SAS+ Encoding for BlocksWorld

- $G_1 = \{(on\ c\ a), (on\ d\ a), (on\ b\ a), (clear\ a), (holding\ a)\}$ ,
- $G_2 = \{(on\ a\ c), (on\ d\ c), (on\ b\ c), (clear\ c), (holding\ c)\}$ ,
- $G_3 = \{(on\ a\ d), (on\ c\ d), (on\ b\ d), (clear\ d), (holding\ d)\}$ ,
- $G_4 = \{(on\ a\ b), (on\ c\ b), (on\ d\ b), (clear\ b), (holding\ b)\}$ ,
- $G_5 = \{(ontable\ a), true\}$ ,
- $G_6 = \{(ontable\ c), true\}$ ,
- $G_7 = \{(ontable\ d), true\}$ ,
- $G_8 = \{(ontable\ b), true\}$ , and
- $G_9 = \{(handempty), true\}$ ,



# PDB(EVENOUT) PDB(ODDOOUT)

EVENOUT(GOAL) = (on c a)

- ((clear a),1)
- ((holding a),2)
- ((on b a),2)
- ((on d a),2)

ODDOOUT(GOAL) = (on a b), (on a c)

- ((on d c) (clear b),1)      • ((on b c) (clear b),4)
- ((on a b) (clear c),1)      • ((on a c) (holding b),4)
- ((on d c) (holding b),2)      • ((on c b) (clear c),4)
- ((clear c) (clear b),2)      • ((on d b) (holding c),4)
- ((on d c) (on d b),2)      • ((on a c) (on d b),4)
- ((on a b) (holding c),2)      • ((on b c) (holding b),5)
- ((on a c) (on a b) ,2)      • ((on a b) (on b c),5)
- ((clear c) (holding b),3)      • ((on d b) (on b c),5)
- ((clear b) (holding c),3)      • ((on c b) (holding c),5)
- ((on a c) (clear b),3)      • ((on a c) (on c b),5)
- ((on d b) (clear c),3)      • ((on c b) (on d c),5)
- ((holding c) (holding  
b),4)

# Pattern Databases

- **Benefits:**

- Heuristic calculations are now constant time (it's a lookup function).
- Reusable when solving the same problem.

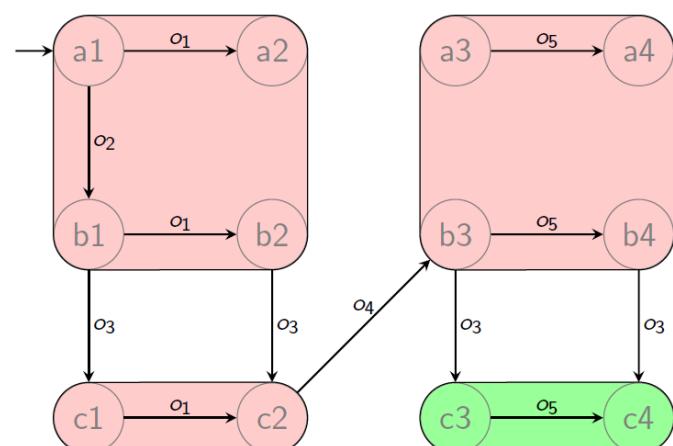
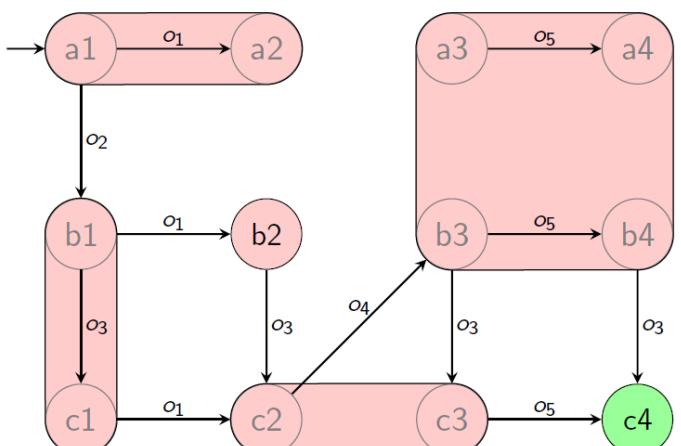
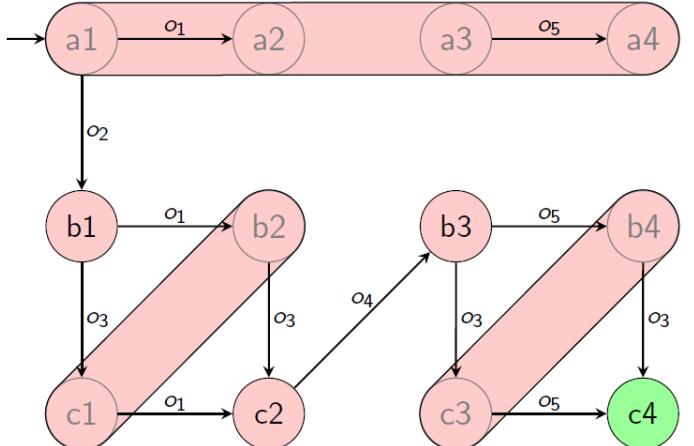
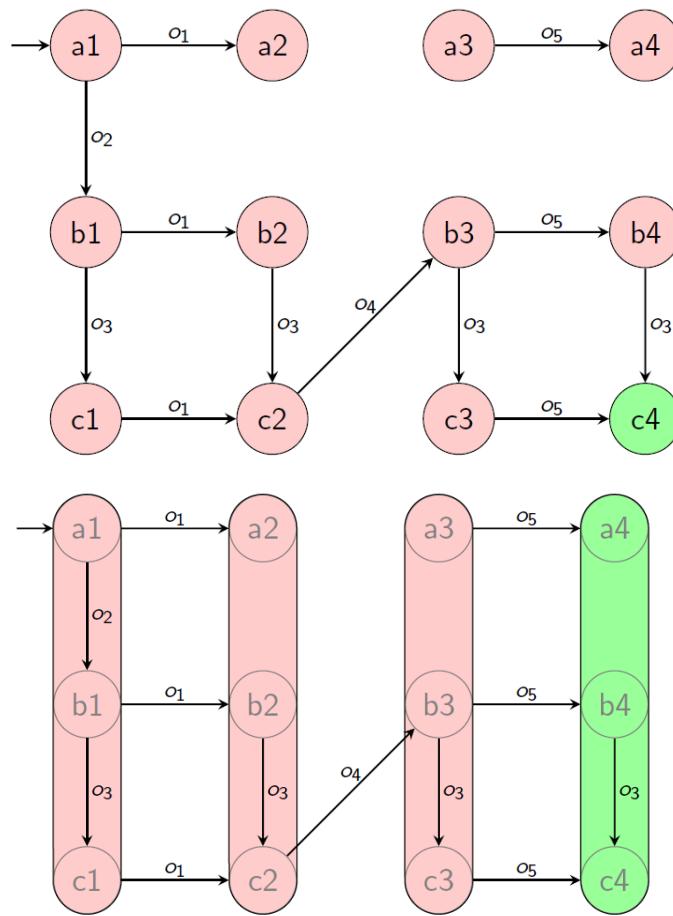
- **Drawbacks:**

- Slow computation time to build the pattern database.
- Reusability is quite limited (changing goal or increasing variables etc.)



# Multiple Forms of Abstraction

- This session – sneakily – introduces you to abstractions for planning and heuristic search.
- Projections (pattern databases)
- Domain abstractions
- Cartesian abstractions
- Merge-and-Shrink abstractions



# Classical Planning Pattern Databases

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



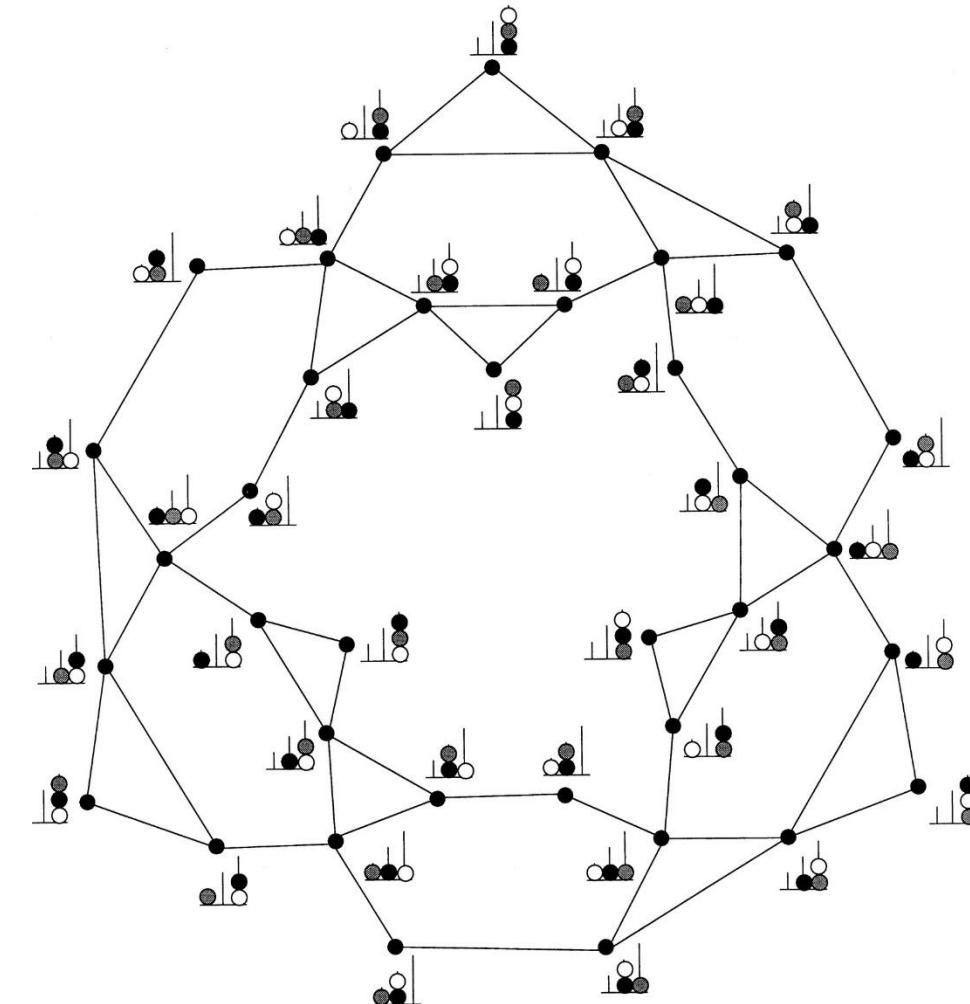
# Classical Planning Optimal Planning Cost Partitioning

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



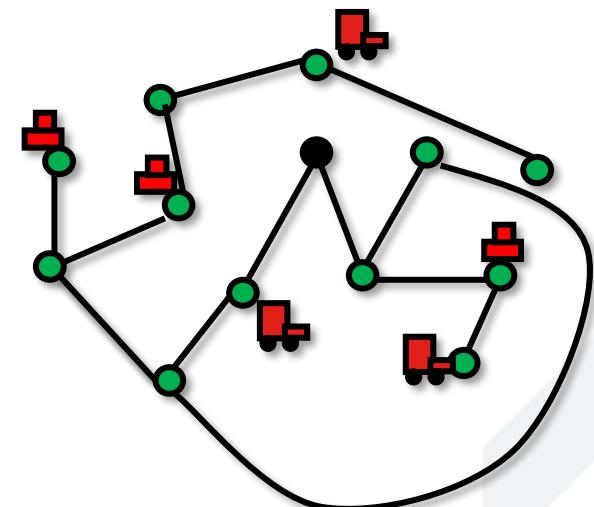
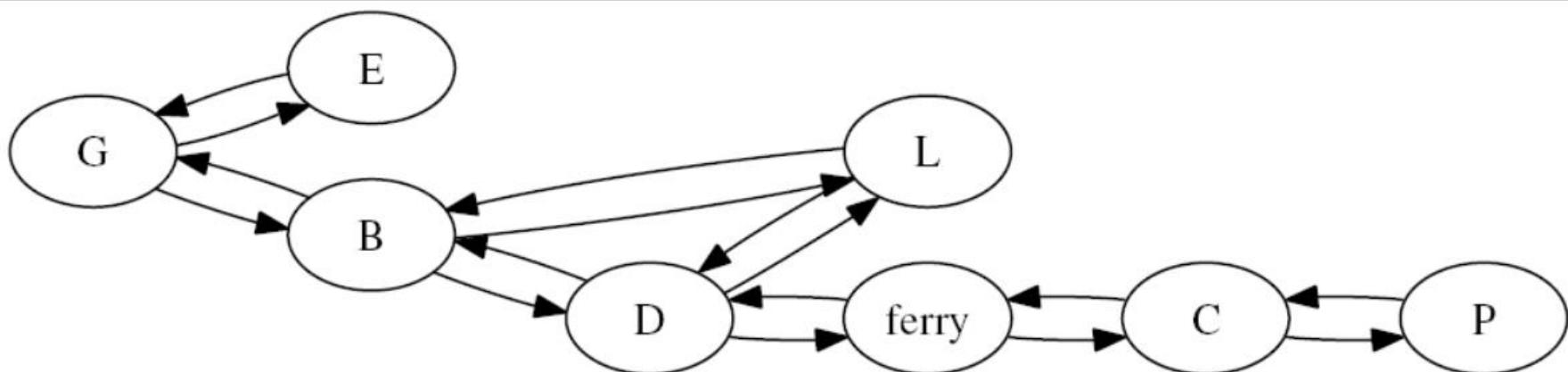
# Recap: The Perils of Expressivity

- By having such an expressive formalism, state spaces can begin to explode.
- How do we circumvent this?
  - Use heuristics to guide the search?
  - Find landmarks to give us sub-goals to solve?
  - Use local search (see EHC)?
- Often sacrificing completeness of search in an effort to find a solution quickly.
  - Hence Best-First Search fallback in EHC.
- What if we instead sacrifice expressiveness for efficiency?



# Recap: SAS+ Planning

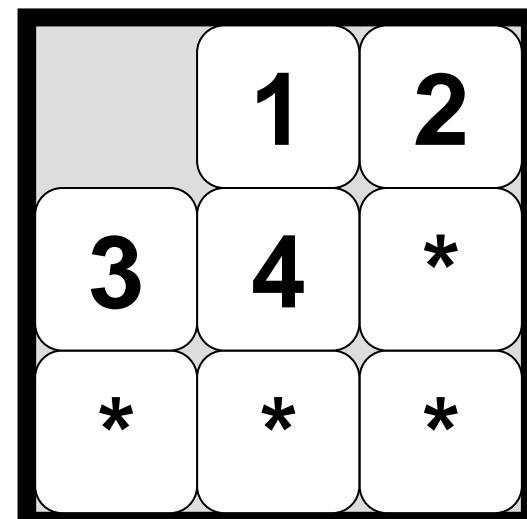
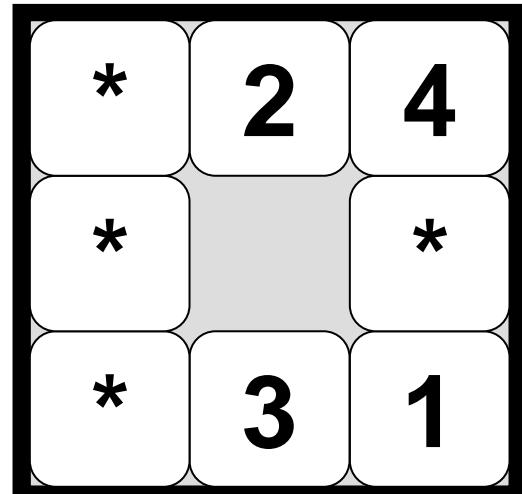
- Optimising planning by analysing the problem.
- SAS+ Planning re-encodes existing problem into new formulation.
- Use of DTG's help us visualise assignment of variables to values during search/execution.



# Recap: Pattern Databases

- Generate heuristics of sub-problems.
- Target Patterns: partial specification of the goal state.
- Establish a Pattern Database (PDB)
  - Set of all permutations of the target pattern.
- Calculate pattern cost.
  - Compute distance to target pattern.
  - Minimum number of actions to achieve target pattern.

Pattern



Target Pattern

# Combining Heuristic Information

- If we want to achieve optimal planning, we need to ensure admissible heuristics are provided.
- But given planning is hard (NP-Hard in even the simplest case), can't expect any one heuristic to always work well in a given task.
- Also, no one heuristic can really grasp the complexity of larger domains.
- So can we find a way to use combine information from different heuristics?



# Example

- If we have multiple abstraction heuristics for the state space, how do we know which is best for a given problem?
- Difficult to determine, especially when using domain independent heuristics.
- Even more difficult to determine on a per-domain and per-problem basis.

$$h_1(s_1) = 5$$

$$h_2(s_1) = 5$$

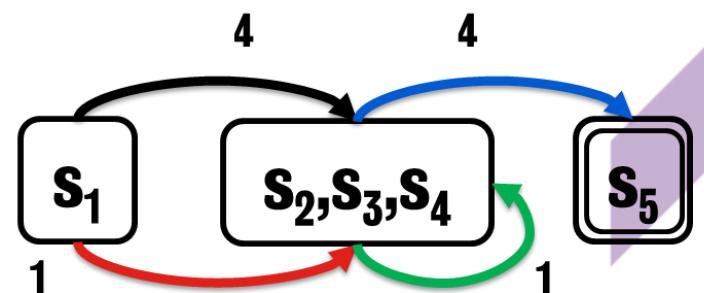
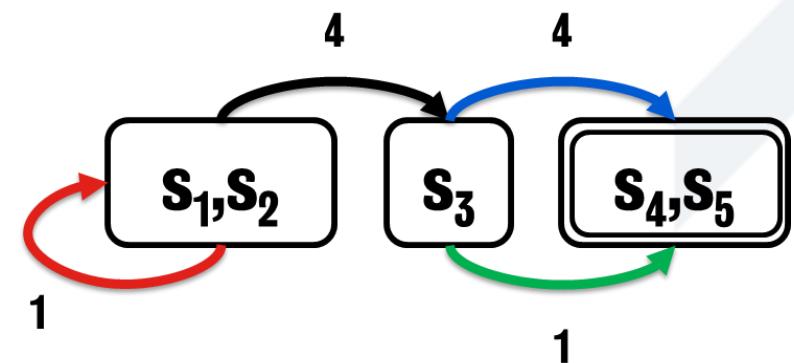
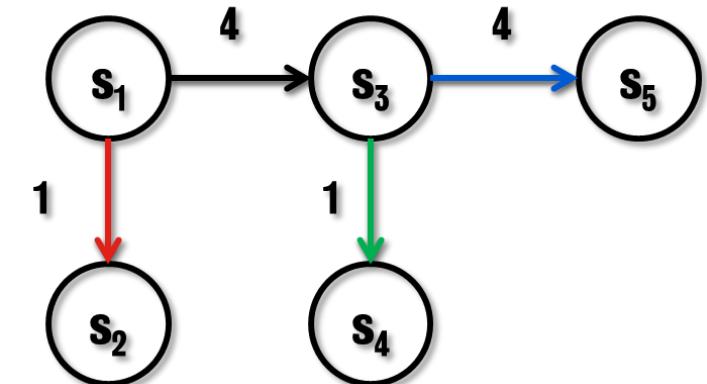


Image Credit:

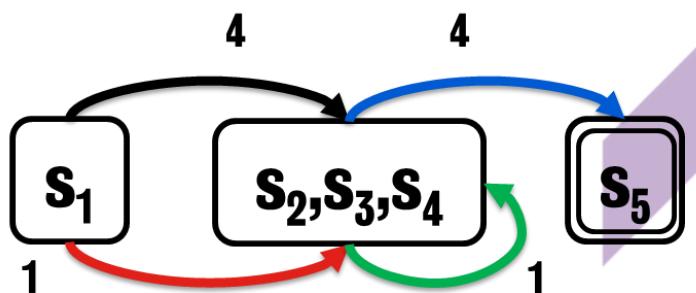
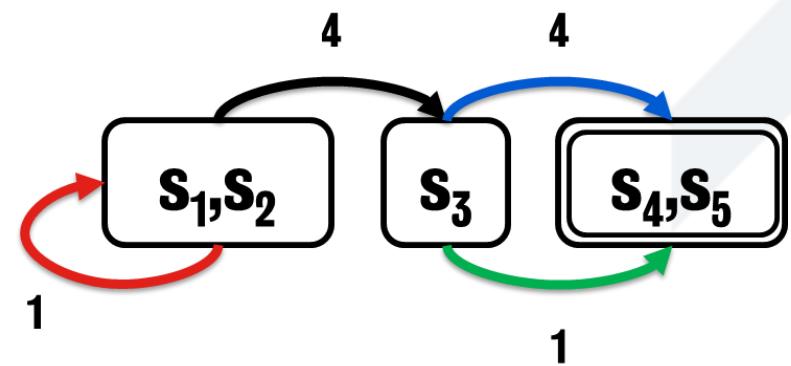
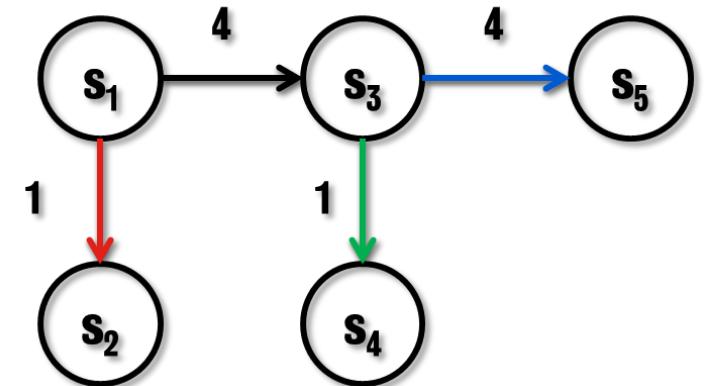
A Comparison of Cost Partitioning Algorithms for Optimal Classical Planning  
J. Seipp, T. Keller and M. Helmert, ICAPS 2017

# Example

- What if we tried to combine or alternate between these two heuristics?
  - $h^{\text{add}}$  or  $h^{\text{max}}$ ?
- $h^{\text{max}}(s_1) = 5$
- $h^{\text{add}}(s_1) = 10$
- $h^{\text{max}}(s_3) = 1$
- $h^{\text{add}}(s_3) = 5$

$$h_1(s_3) = 1$$

$$h_2(s_3) = 4$$



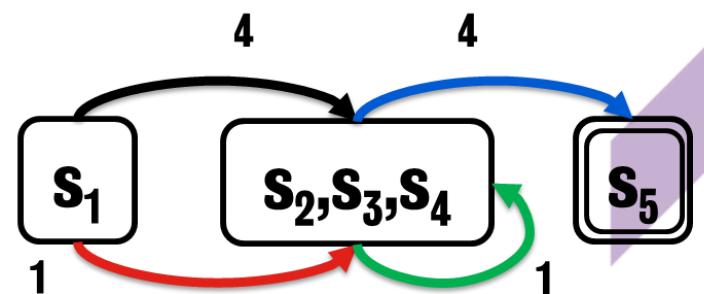
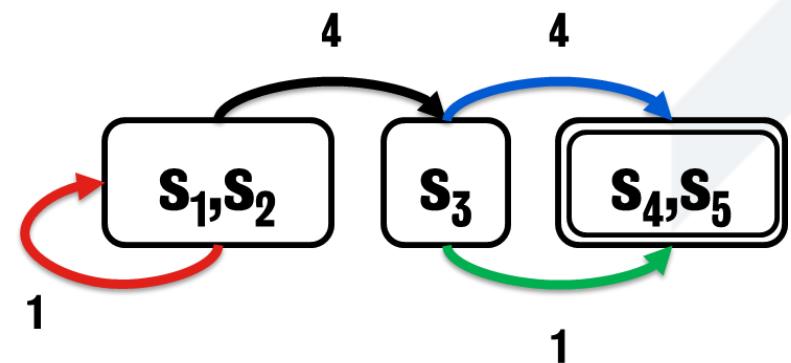
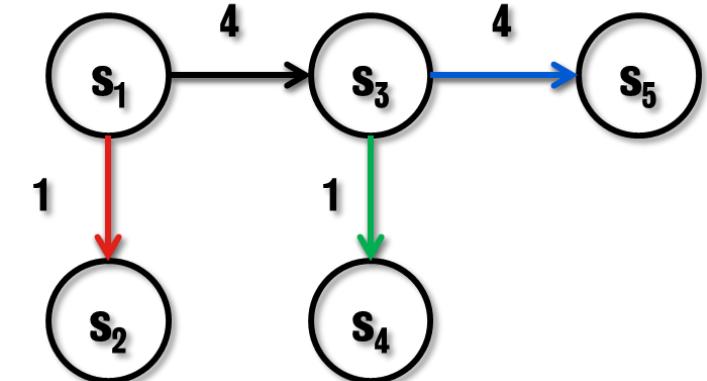
# Cost Partitioning

- Distribute the cost of each action across identical planning tasks.
- Combine arbitrary admissible heuristics into one single heuristic.
- However, the partitioning is designed such that sum of heuristics is now additive (and therefore admissible).

$$h(s_1) = h_1(s_1) + h_2(s_1) \leq g(sg)$$

$$h_2(s_1) = 5$$

$$h_1(s_1) = 5$$

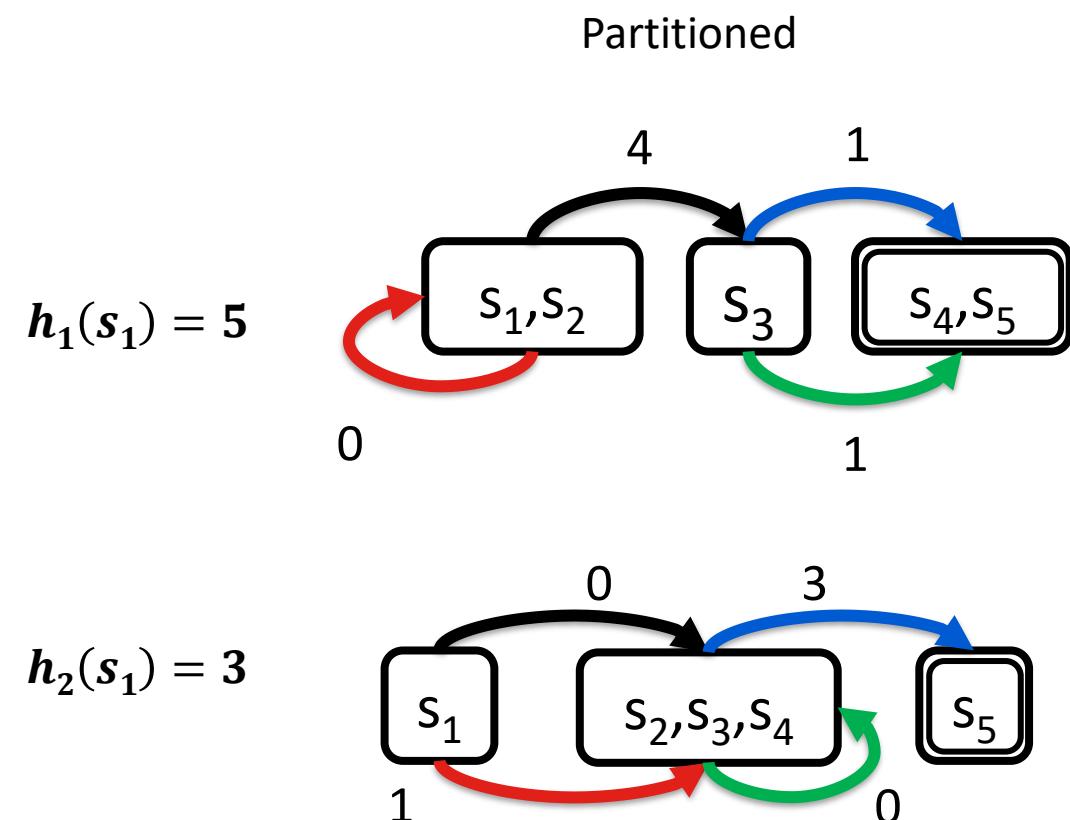
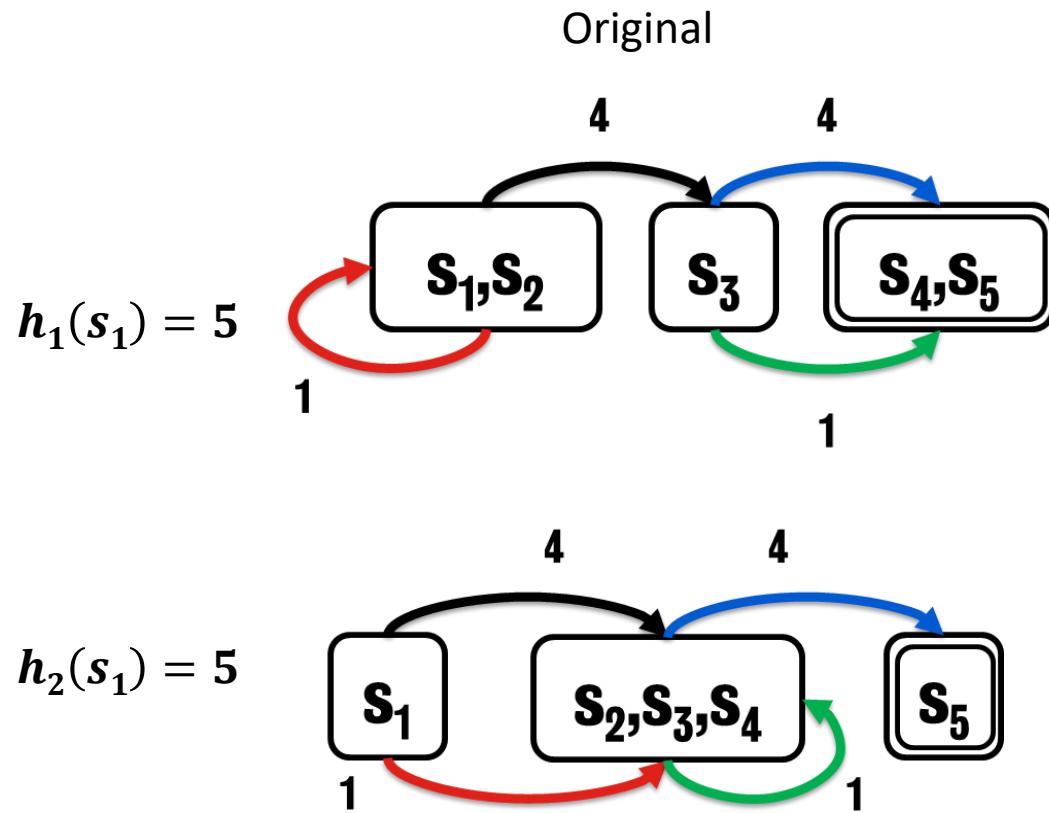


# Optimal Cost Partitioning

$$h(s_1) = 5 + 3 = 8$$

- The highest heuristic value for a given state amongst all cost partitionings.

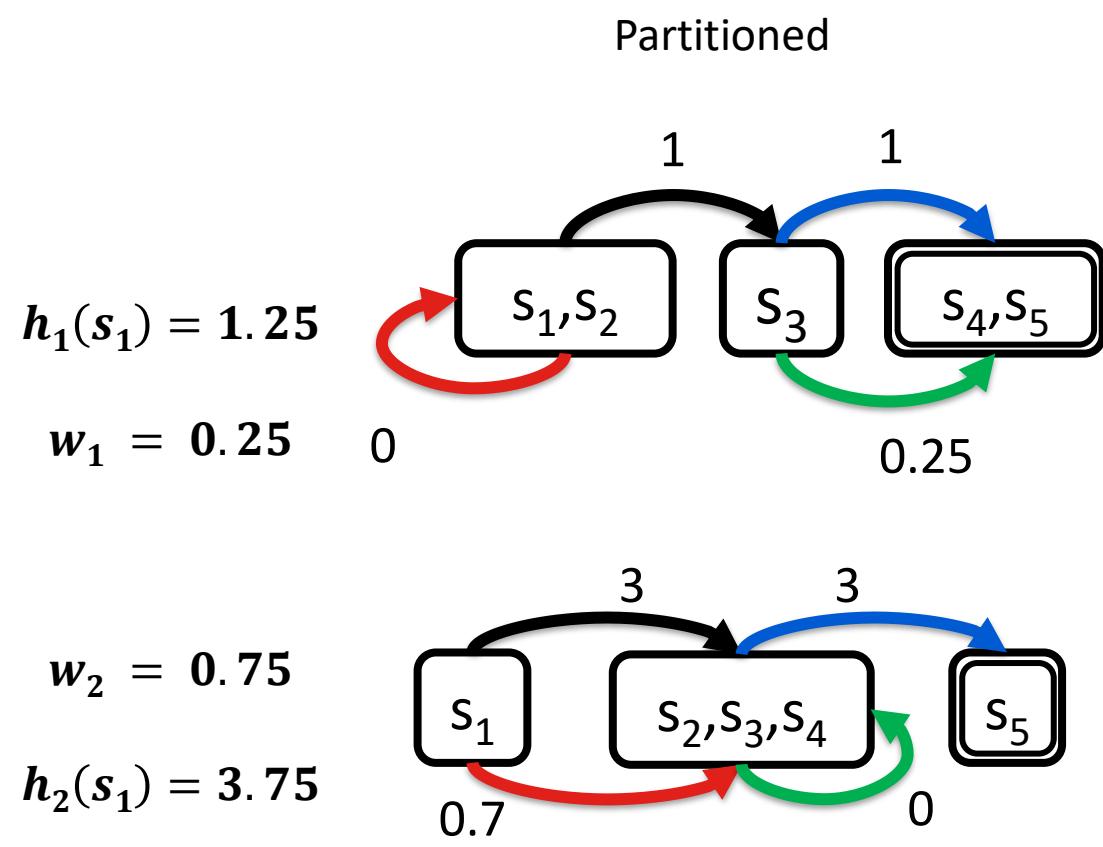
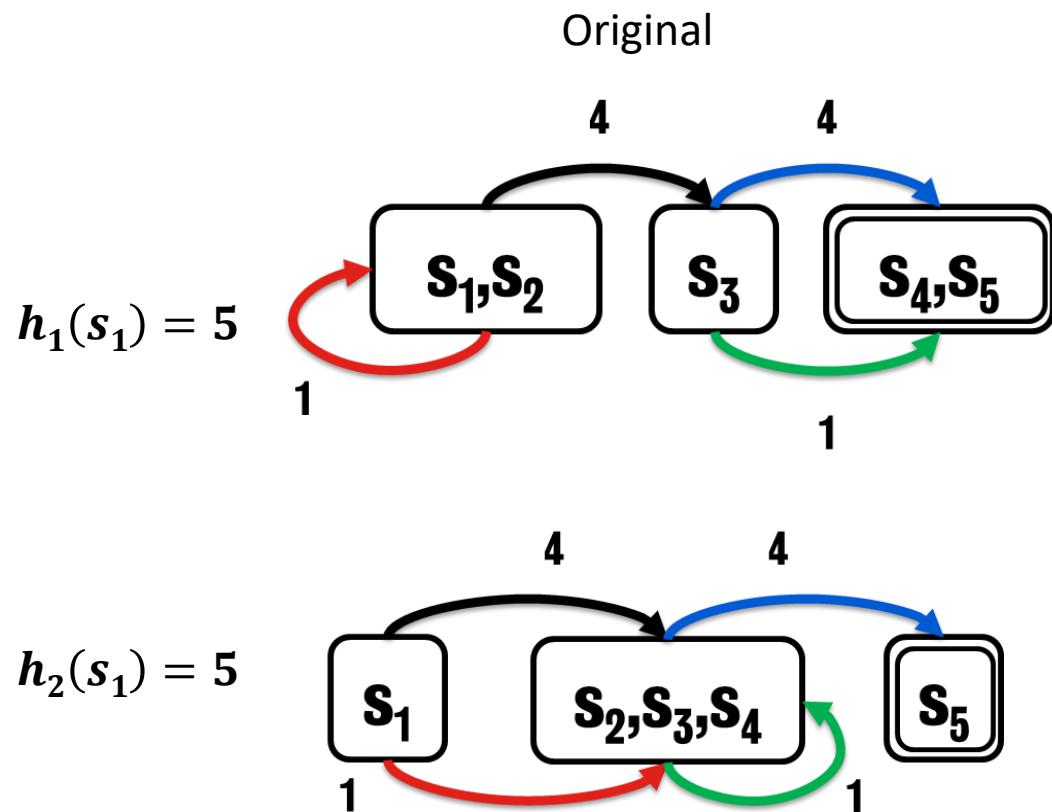
- Pro: Fast to compute (polynomial time)
- Cons: Still too expensive in practice



$$h(s_1) = 1.25 + 3.75 = 5$$

# Post Hoc Optimisation

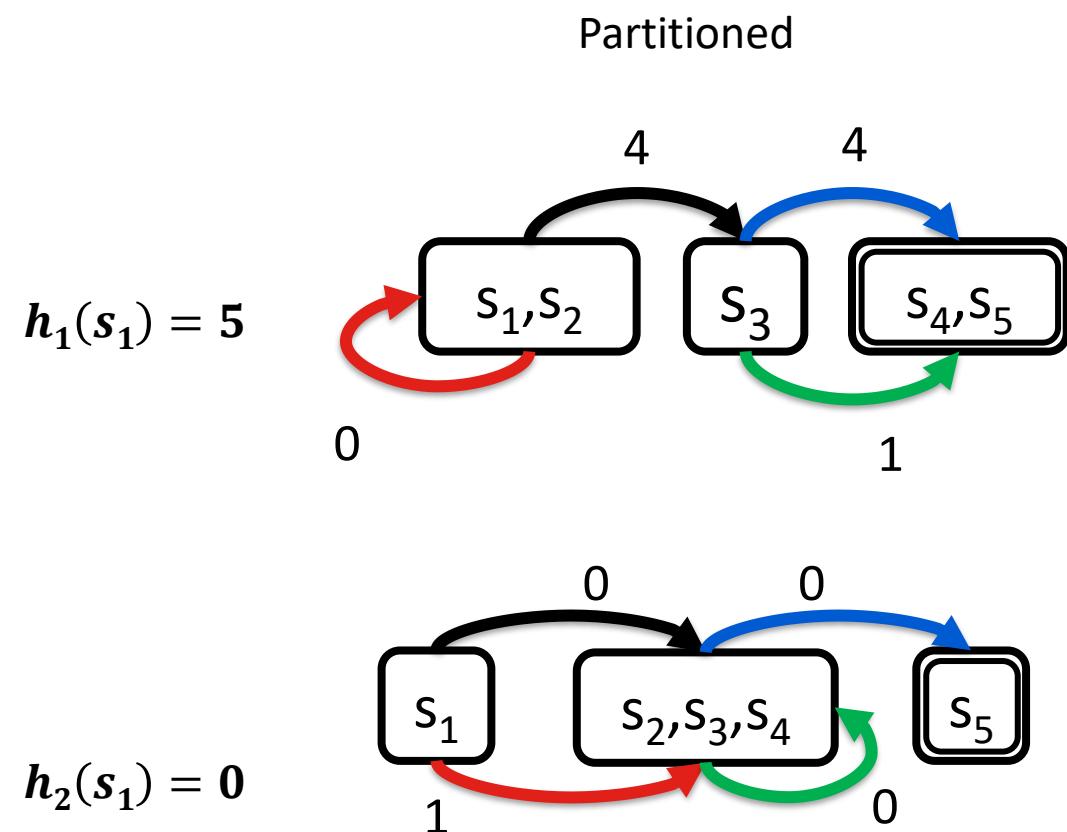
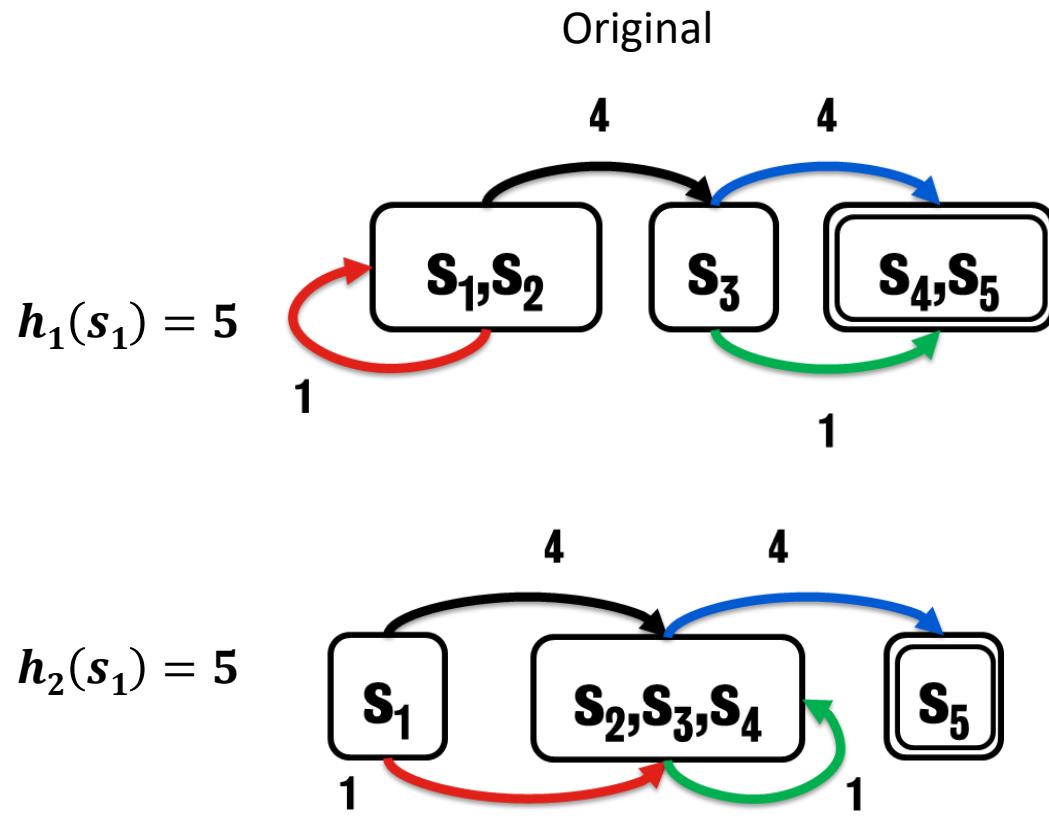
- Compute weight factors  $w$  to be applied to costs relevant to  $h$ .
- If not relevant to  $h$ , set cost to 0.



$$h(s_1) = 5 + 0 = 5$$

# Greedy Zero-One Cost

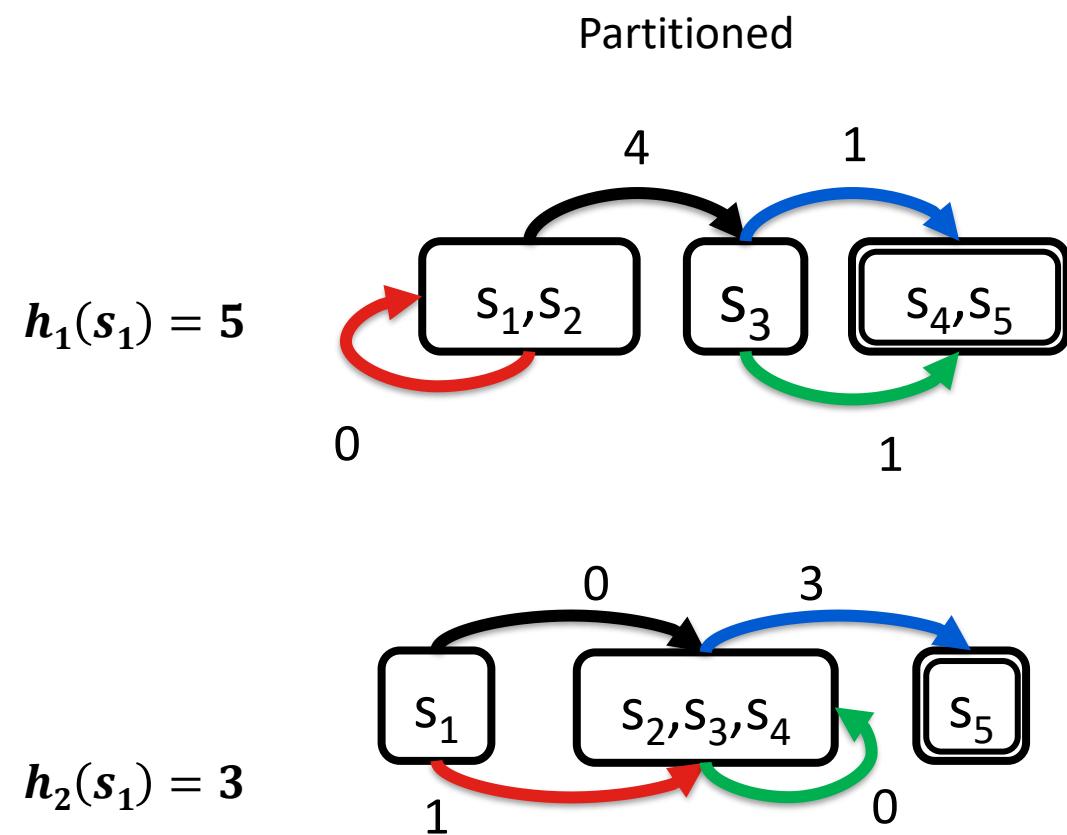
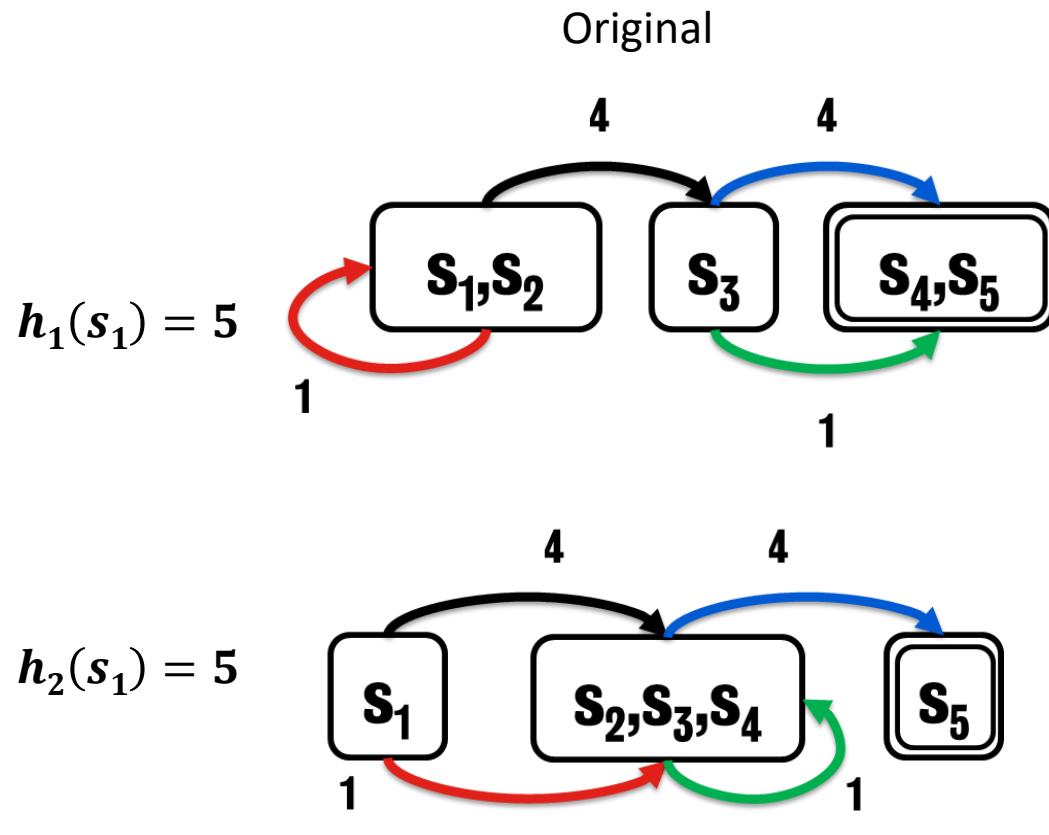
- Orders heuristics in a particular sequence.
- We then use the full costs for the first relevant heuristic.



$$h(s_1) = 5 + 3 = 8$$

# Saturated Cost Partitioning

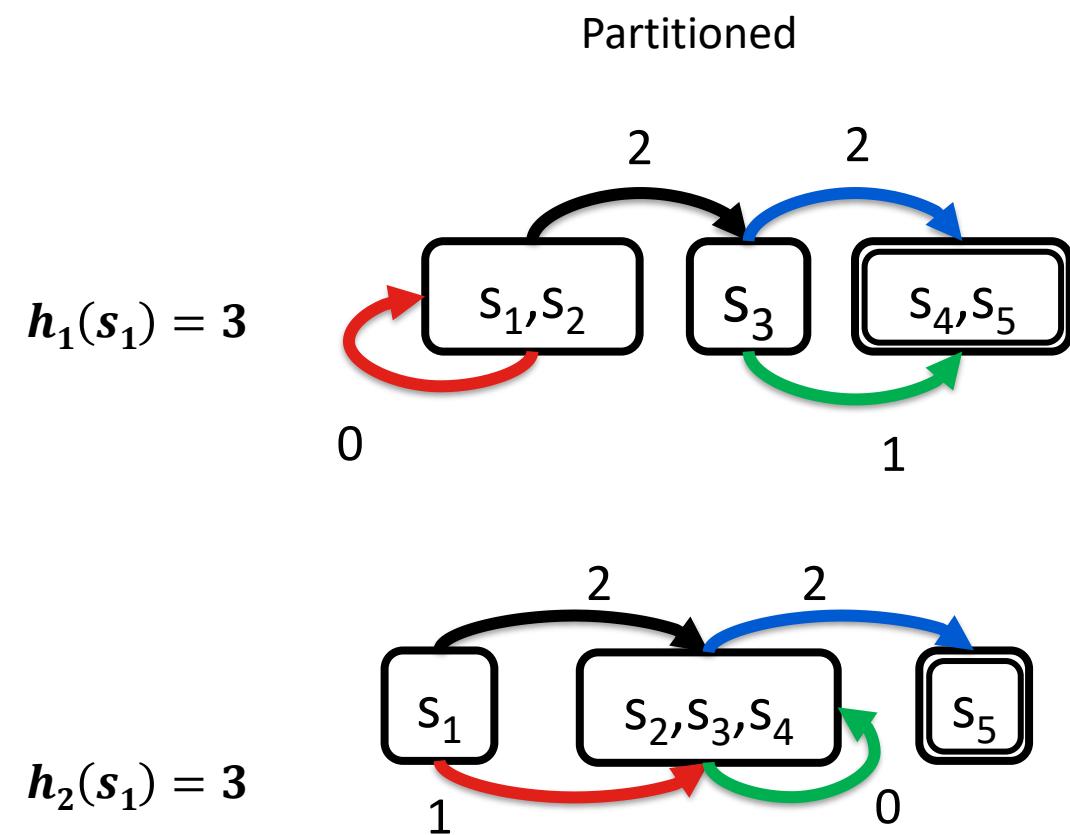
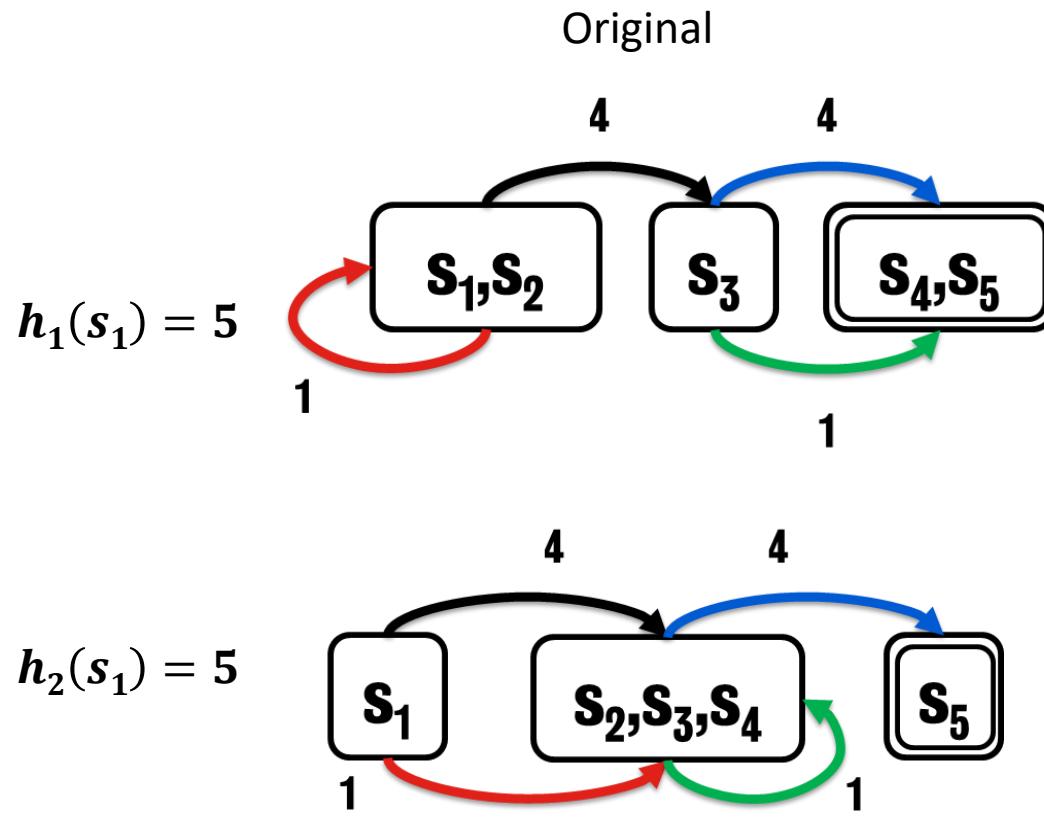
- Orders heuristics in a particular sequence.
- Use minimum costs to preserve estimate.
- Use remaining cost for other heuristics.



$$h(s_1) = 3 + 3 = 6$$

# Uniform Cost Partitioning

- Distribute costs uniformly to all relevant actions.
- Distribution across all heuristics.



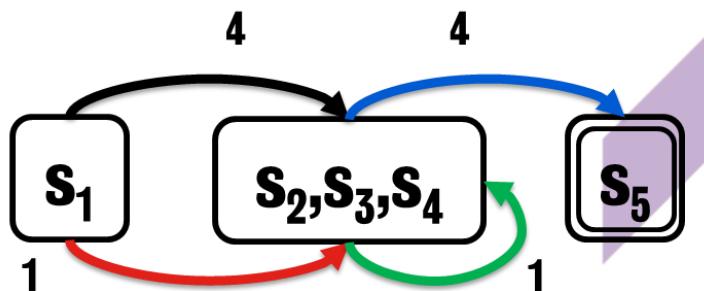
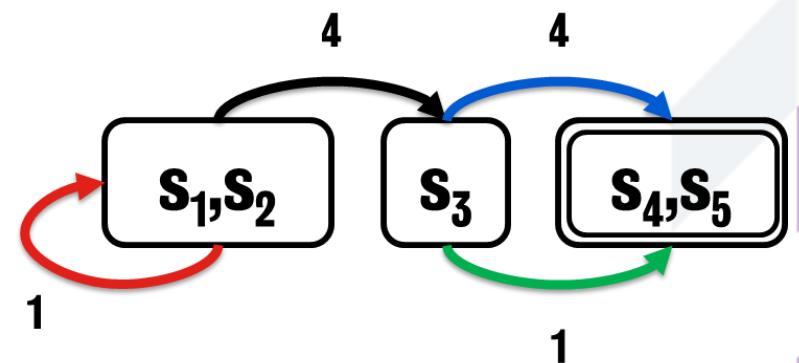
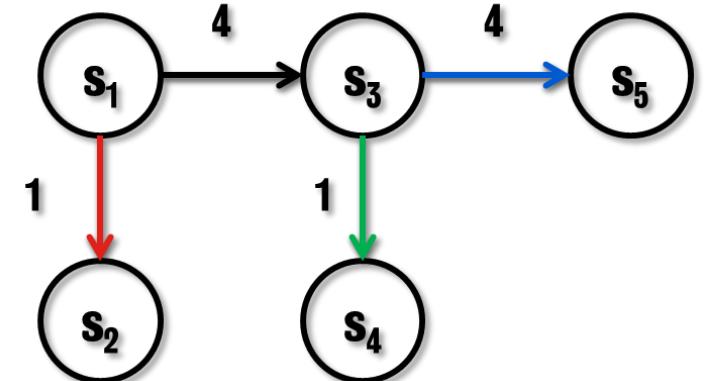
# In Summary

- Cost partitioning is one method to allow for us to combine multiple admissible heuristics.
- While useful in optimal planning, not something we will then pursue in the rest of the module.

$$h(s_1) = h_1(s_1) + h_2(s_1) \leq g(sg)$$

$$h_2(s_1) = 5$$

$$h_1(s_1) = 5$$



# Classical Planning Optimal Planning Cost Partitioning

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# A.I. PLANNING

Amanda Coles



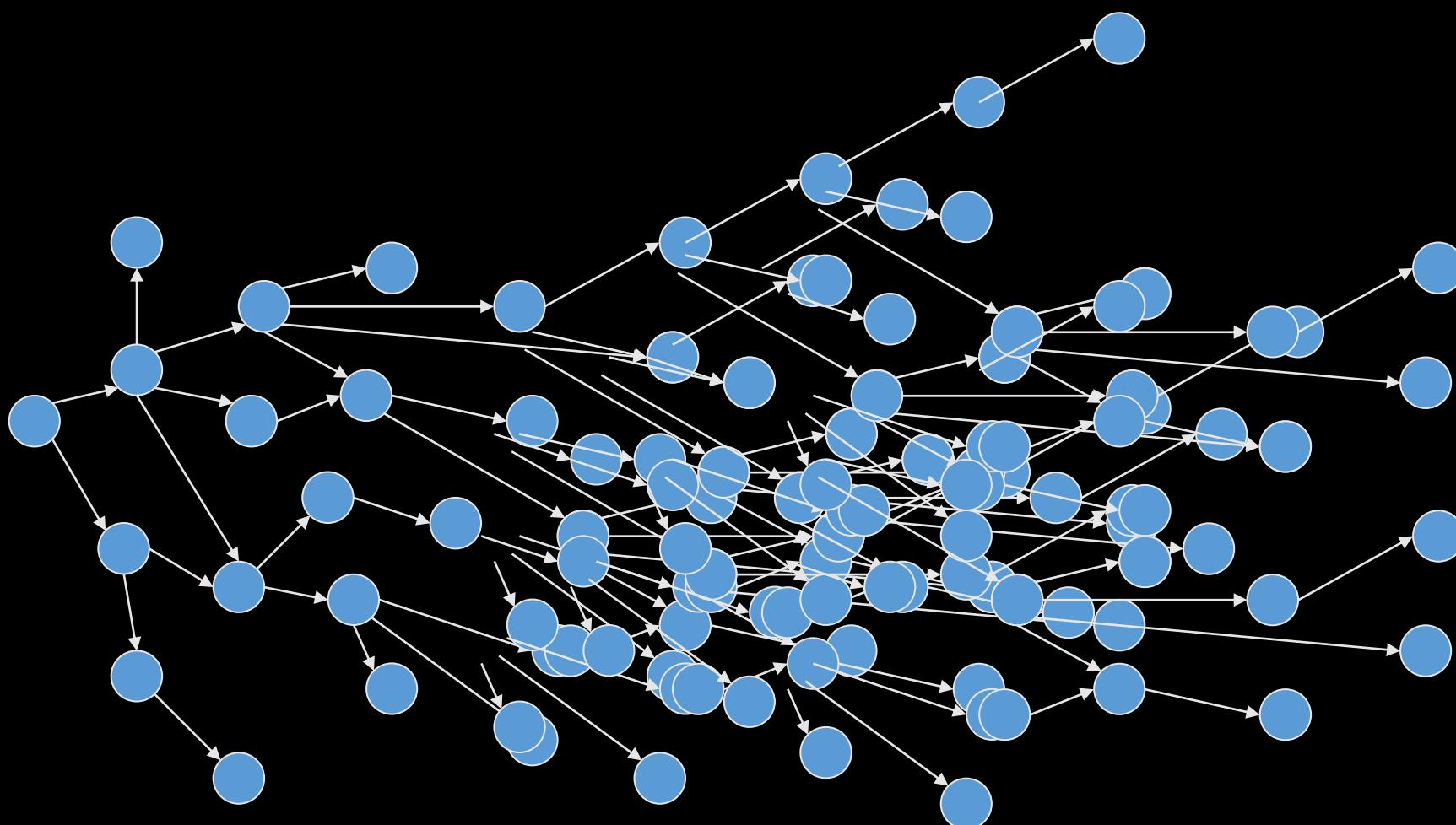
“You know what the difference is between a dream and a goal?... A plan.” – Jodi Picoult



# PLANNING

Still a good thing to do first

“In any non-trivial situation, there is a **choice** about what to do next...”



# Planning – The Problem

- In planning problems we have:
  - an initial state, I

(at mydvd amazon)

(at truck amazon)

(at driver home)

(path home amazon)

(link amazon london)

(link london myhouse)

# Planning – The Problem

- In planning problems we have:
  - an initial state, I
  - a goal, G

(at mydvd myhouse)

# Planning – The Problem

- In planning problems we have:
  - an initial state, I
  - a goal, G
  - some actions, A, defined according to a domain

(load mydvd truck depot)

(walk driver home amazon)

(board-truck driver truck amazon)

(drive-truck driver amazon london)

...

# Planning – The Problem

- In planning problems we have:
  - an initial state, I
  - a goal, G
  - some actions, A, defined according to a domain
    - these have **preconditions** and **effects**

```
(:action load
  (:parameters (?d - item ?t - truck ?p - place)
    (:precondition (and (at ?t ?p) (at ?d ?p)))
    (:effect (and (not (at ?d ?p))
      (in ?d ?t)))
  )
)
```

# Planning – The Problem

- In planning problems we have:
  - an initial state, I
  - a goal, G
  - some actions, A, defined in a domain
- A planner **takes these** and gives us **a plan**.
  - An ordered sequence of actions from A that will turn I into G
  - **what** to do; and **when** to do it.
  - Performs **search** to consider the different possible plans available, until a plan from I to G is found.

# Planning More Formally

- A classical (STRIPS) planning problem is a tuple  $\langle P, A, I, G \rangle$  where:
  - $P$  is a finite set of propositions;
  - $A$  is a set of actions each of which has:
    - A precondition, a set of propositions that must be true in order for  $A$  to be executed;
    - Add effects: a set of propositions added that become true upon execution of  $A$ ;
    - Delete effects: A set of propositions that are deleted upon execution of  $A$ .
  - $I$  a finite set of propositions representing the initial state
  - $G$  the goal, a set of propositions that must be true for a state to be considered a goal state.

(at mydvd amazon)

(at truck amazon)

(at driver home)

(path home amazon)

(link amazon london)

(link london myhouse)

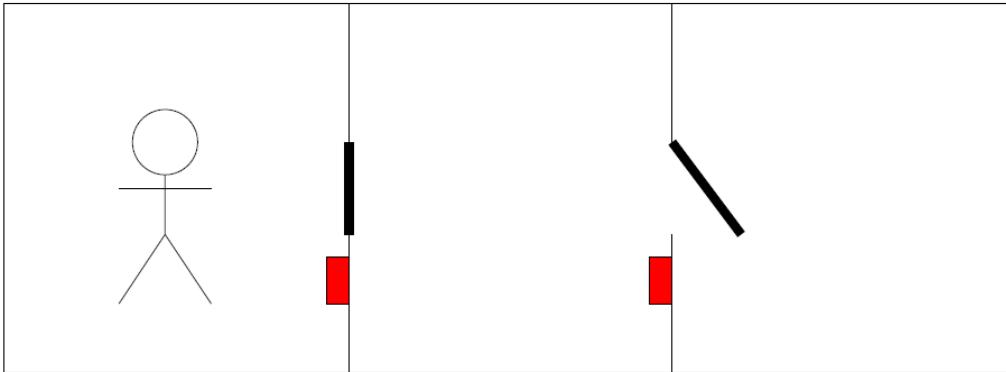


(at mydvd myhouse)

- Initial: (nostranger tollove), (knows rules)
- Goal: (so do I)

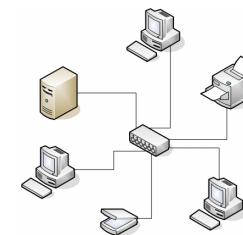
Action	Preconditions	Effects
Never	(full commitment) (knows rules)	(not (full commitment)) (so do I)
Gonna	(knows rules)	(not (knows rules)) (any other guy)
Give	(any other guy)	(not (any other guy)) (knows rules)
YouUp	(nostranger tollove) (any other guy)	(not (nostranger tollove)) (full commitment)

# Back to the Easier Problems...



- How might one encode this in PDDL? Initial state: right door is open; standing in 1st room. Goal: standing in third room.
  - Right door open (knows rules)
  - In First room (nostranger tollove)
  - Goal of standing in right room (so do I)
  - Actions for buttons: Gonna and Give
  - To move, Never and YouUp
- Intuition is a powerful tool.

# Why should I care?



We know: **the current state of the world**: Rover at X, energy supply adequate, package at Y etc.

**The desired state of the world**: Science data, continued adequate supply, package at Z, clean house, sell goods etc.

**What actions we can carry out** (under which conditions): Sample soil, switch on power station, drive truck, clean floor, print page.

**We want to know what to do and in what order to do it.**

\$775m USD

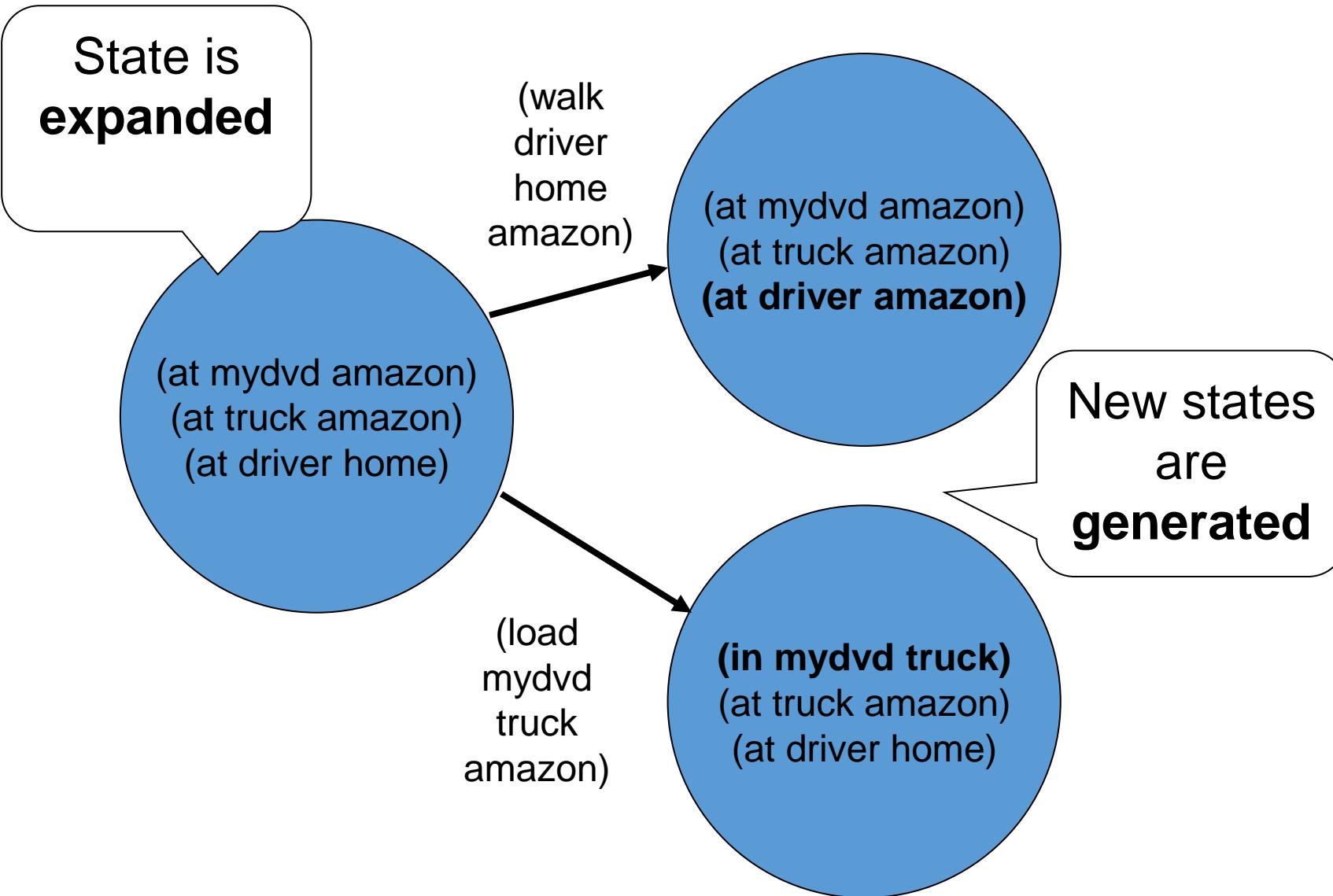


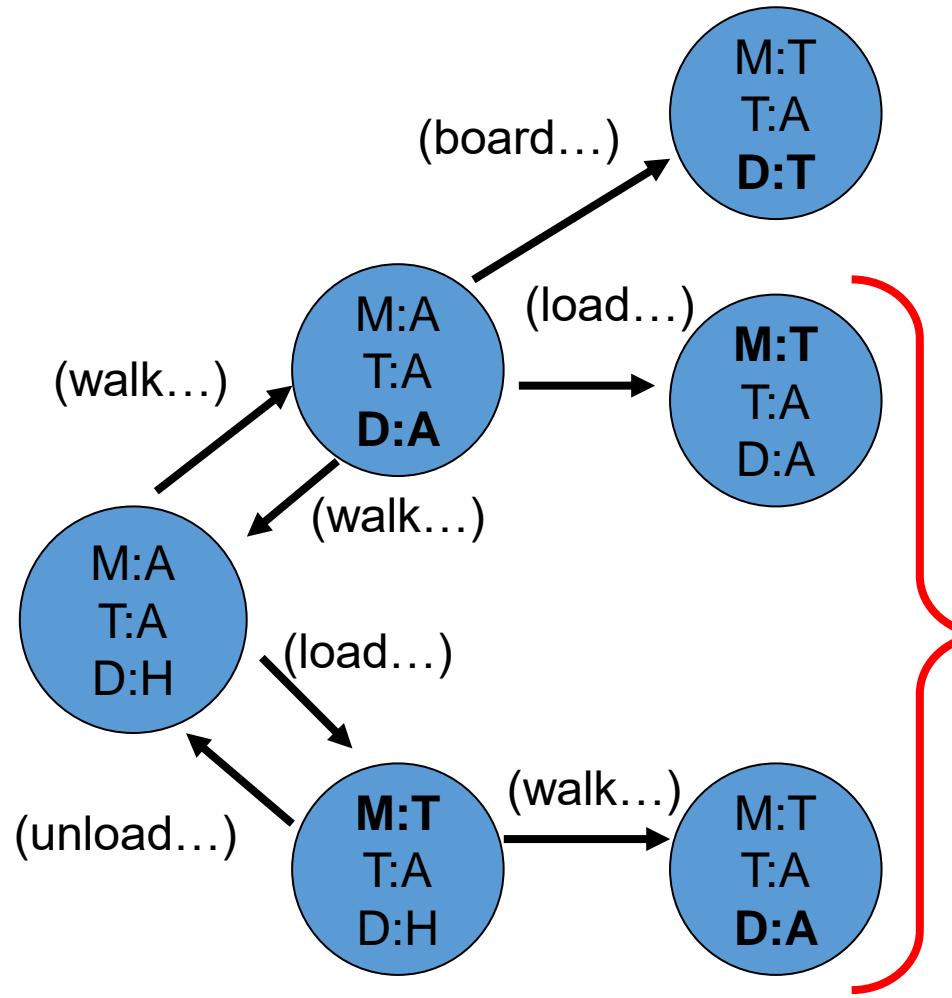
<https://www.youtube.com/watch?v=YSrnV0wZywU>

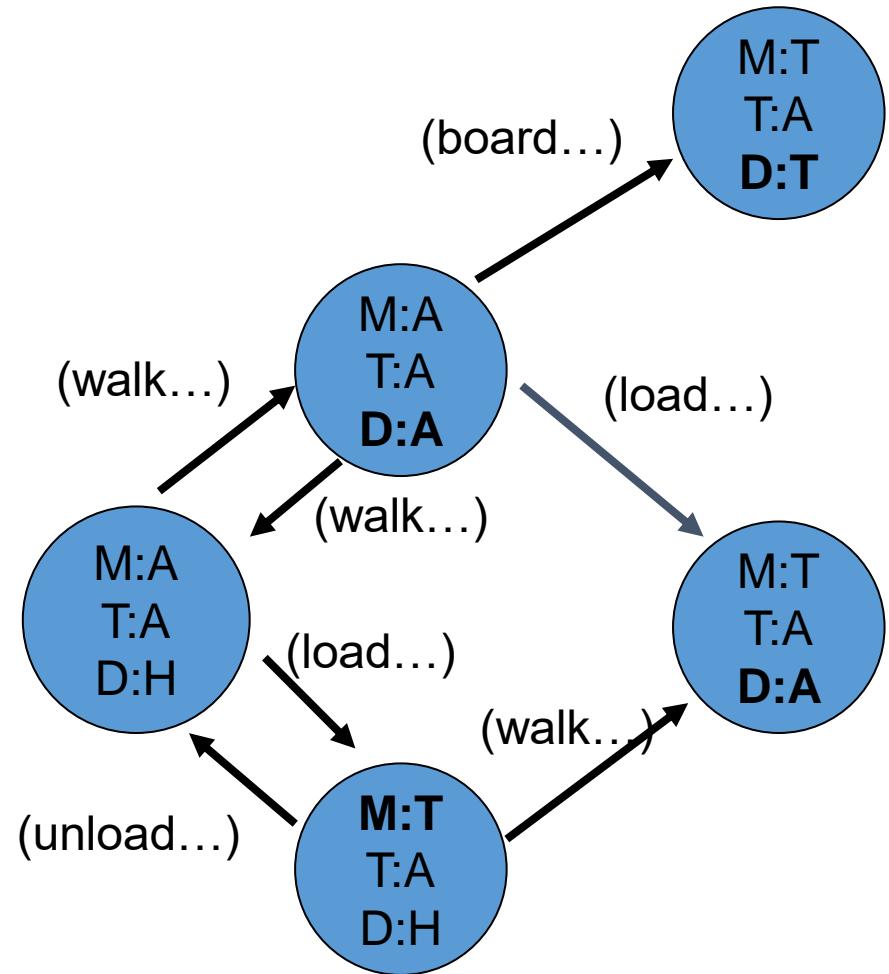
- Warehouse Video 1:32
- Animation @ 12:00
- Example plan @ 26:17

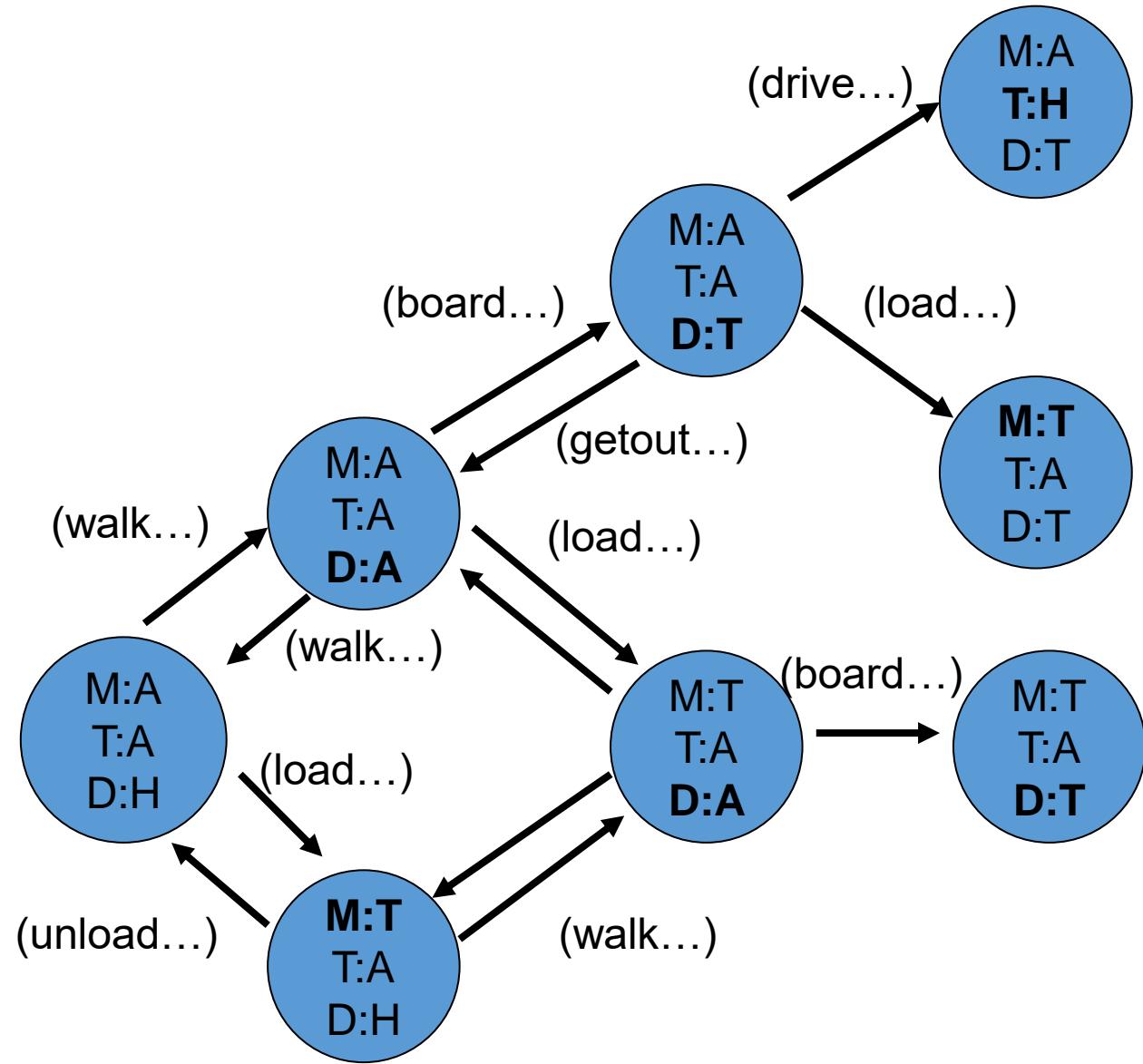
# Planning as Forward Search

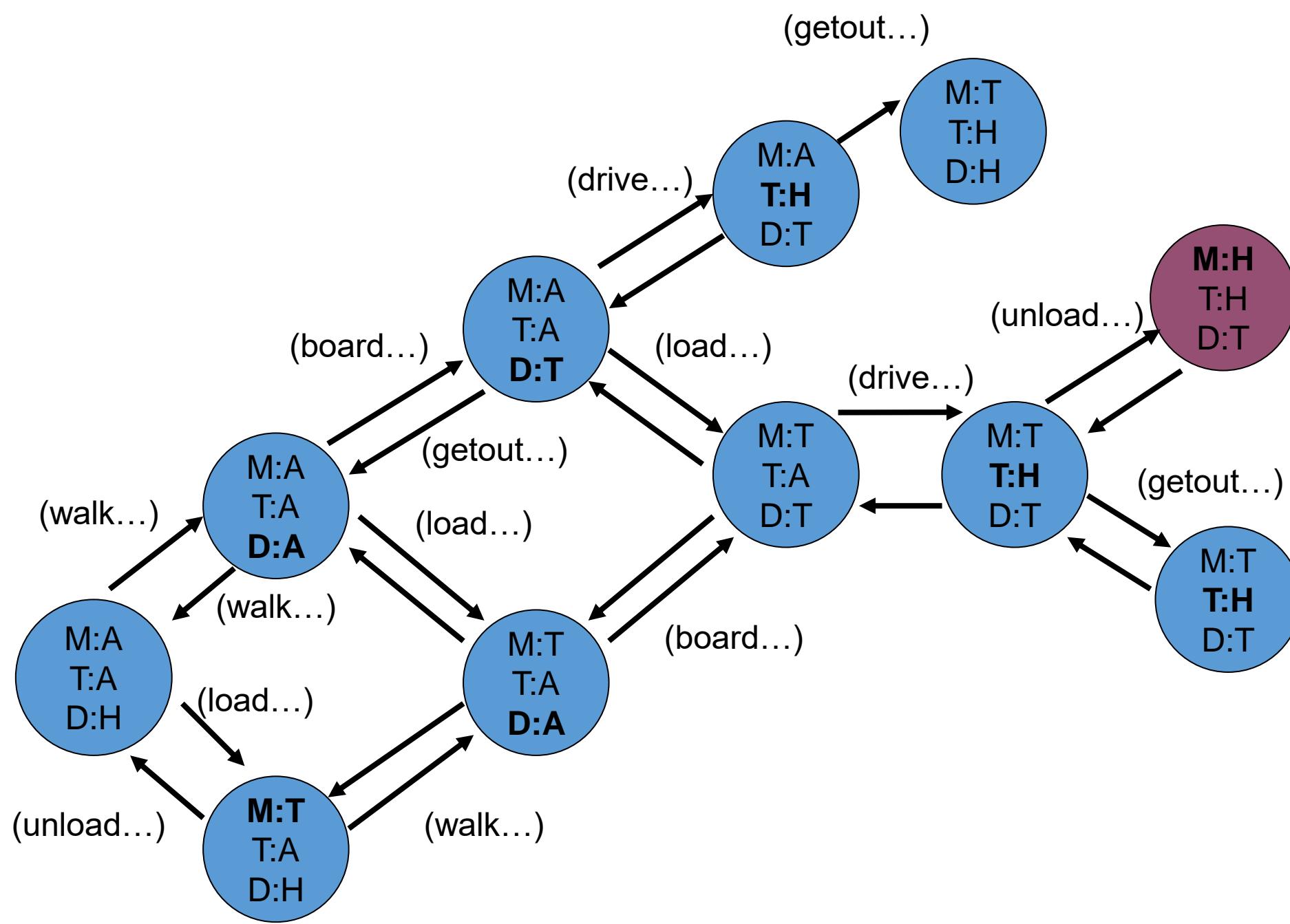
# Search spaces



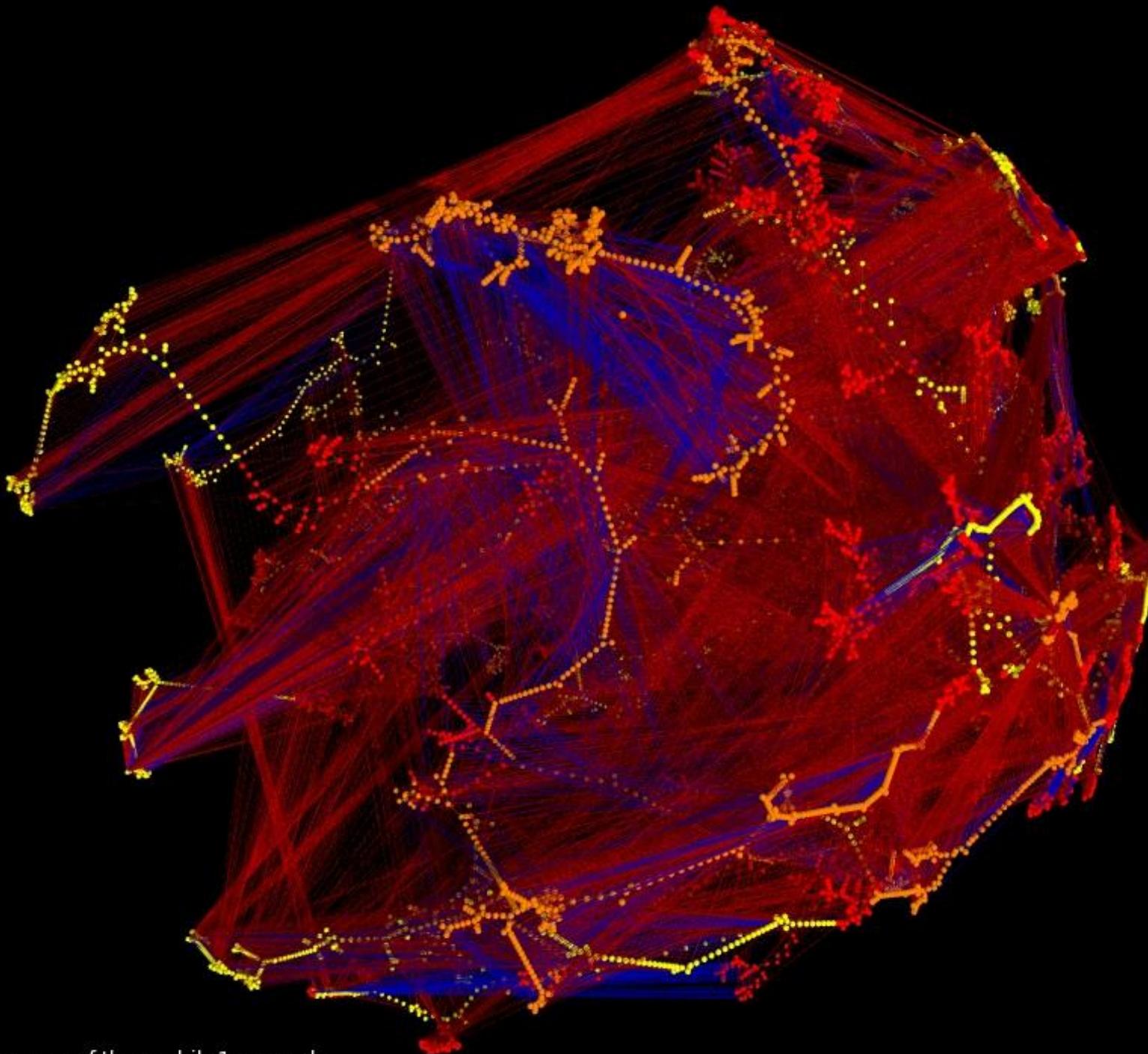






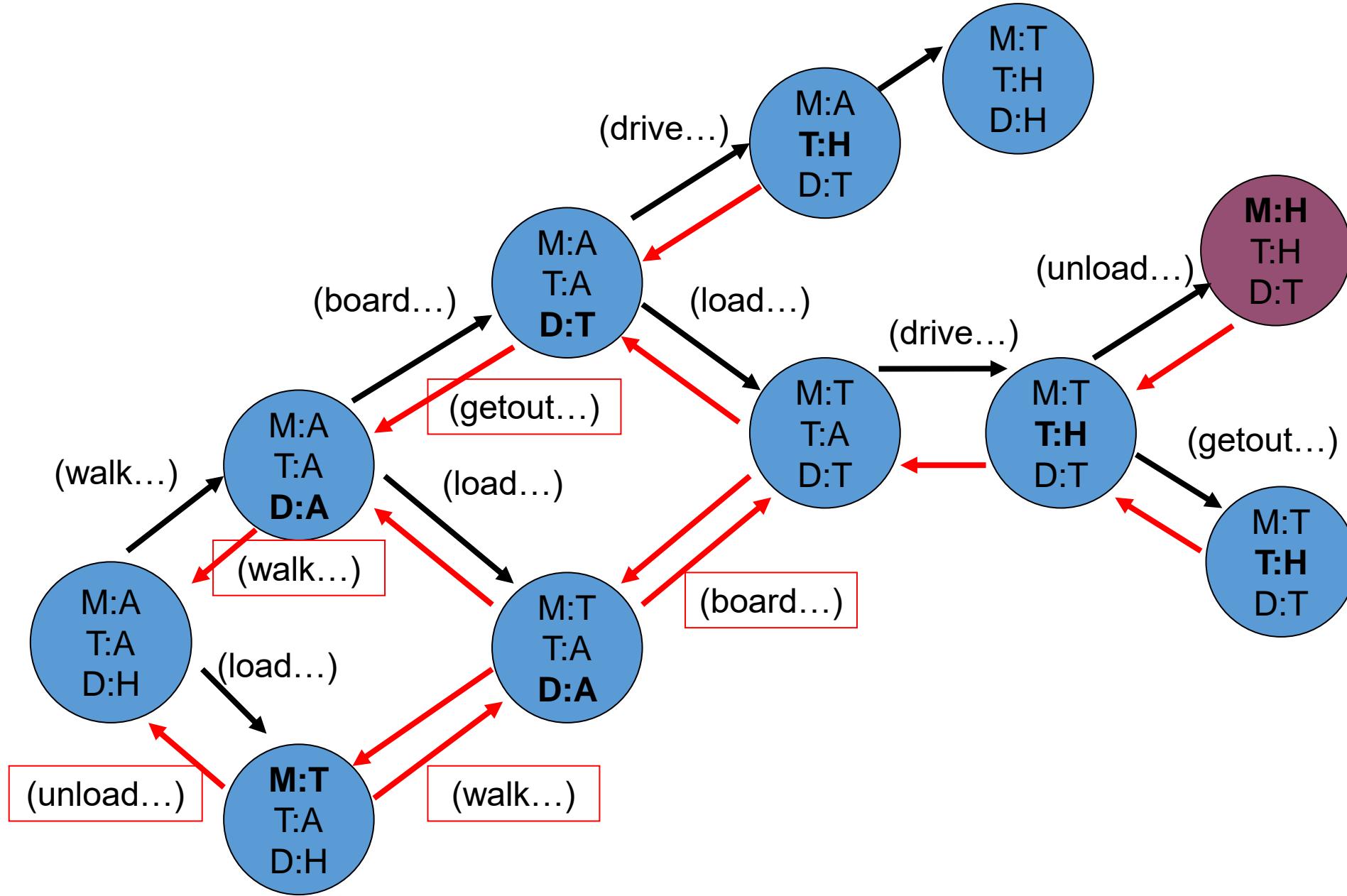


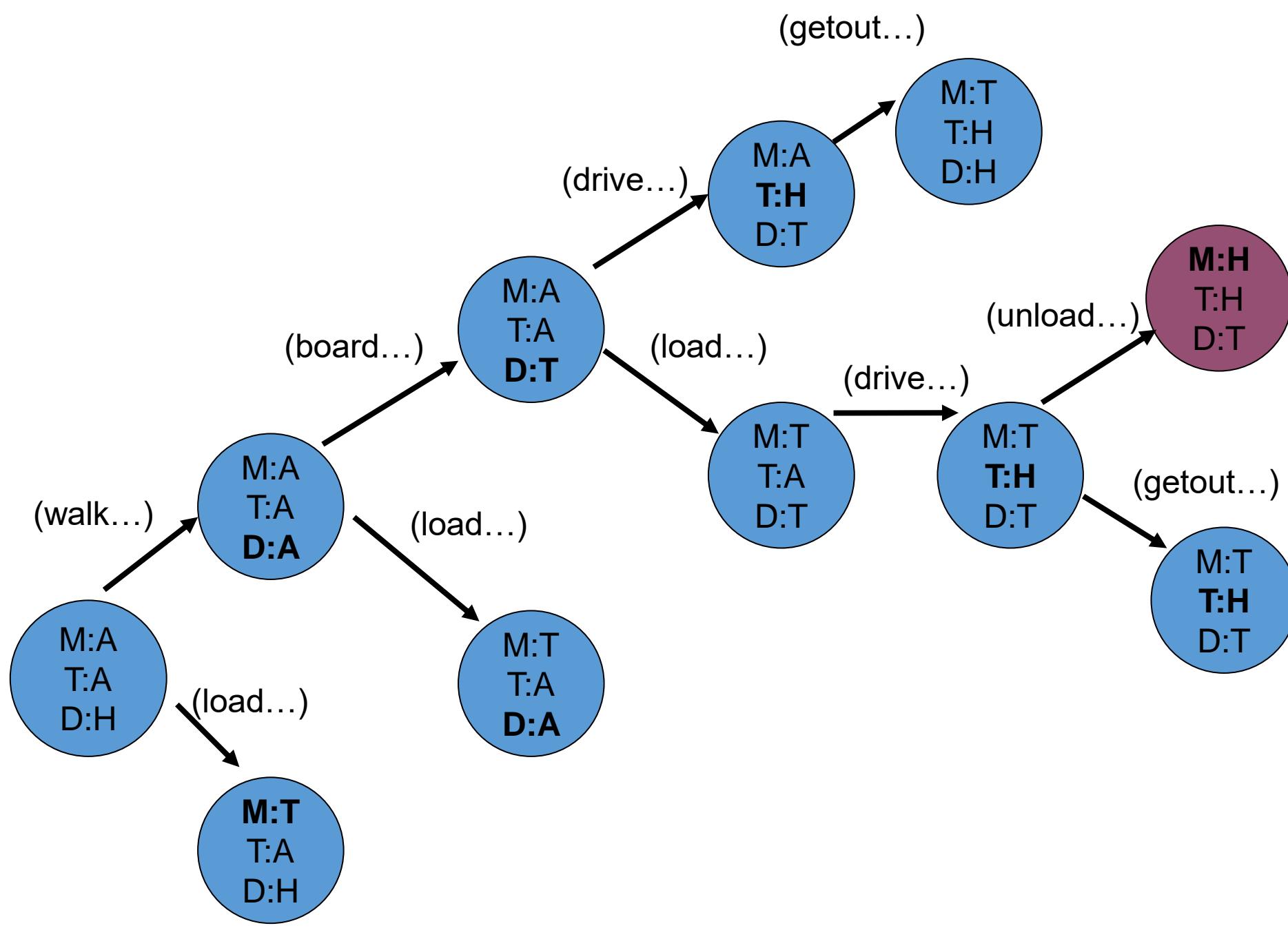
State space of the mobile1 example



# Don't go round in circles

- Keep a closed list:
  - All states that have been seen before
- When expanding a state, **ignore successors on the closed list**





# Forward Search, Graphs & Trees

- The search space is a **Directed Graph**
  - Lots of different paths, can undo all actions, go around in circles, ...
- **Forwards search** from the initial state, with a closed list, builds a **tree**
  - Only one path into each node is kept
  - No backwards edges

# Forward search

- **What you have been doing so far**
- Closed list – states already dealt with
- Also, **Open list – states to deal with**
- Initially: closed = [], open = [I]

while open not empty:

S = remove the next state from open

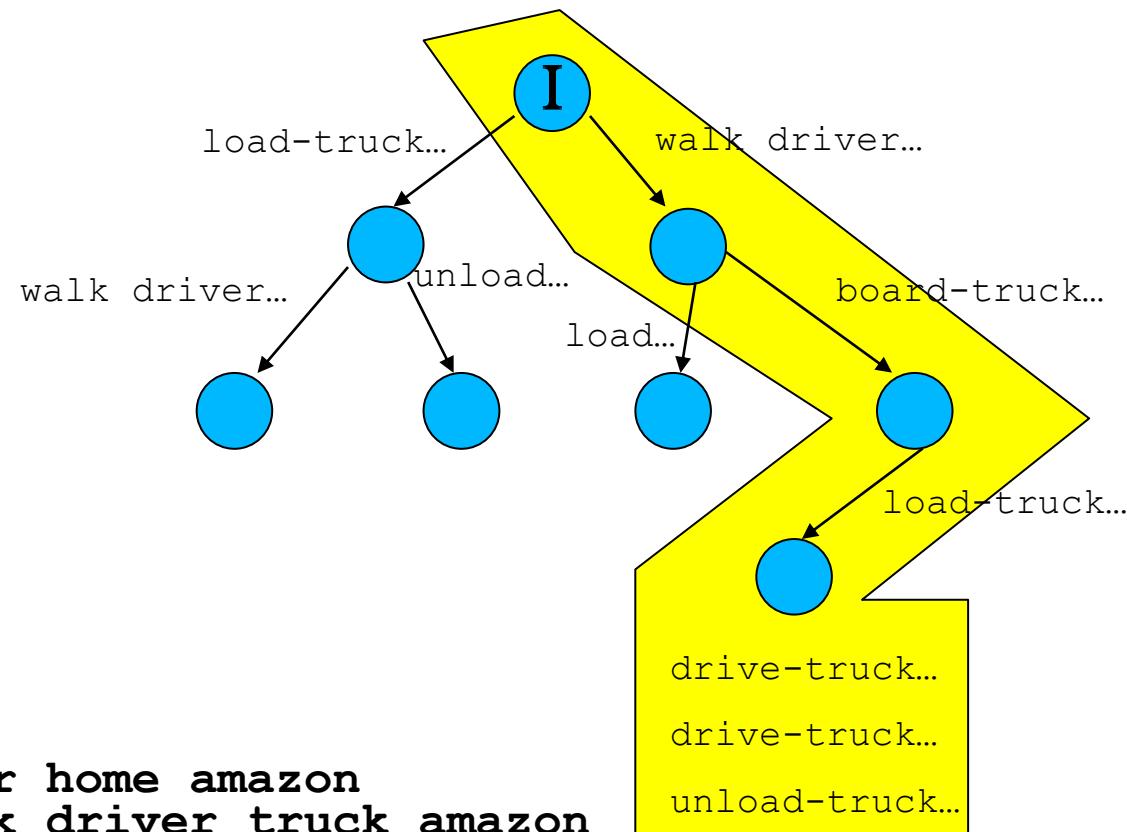
If S is not in **closed**

expand S and

add S to **closed**;

add successors to **open**

# Planning as Forward Search

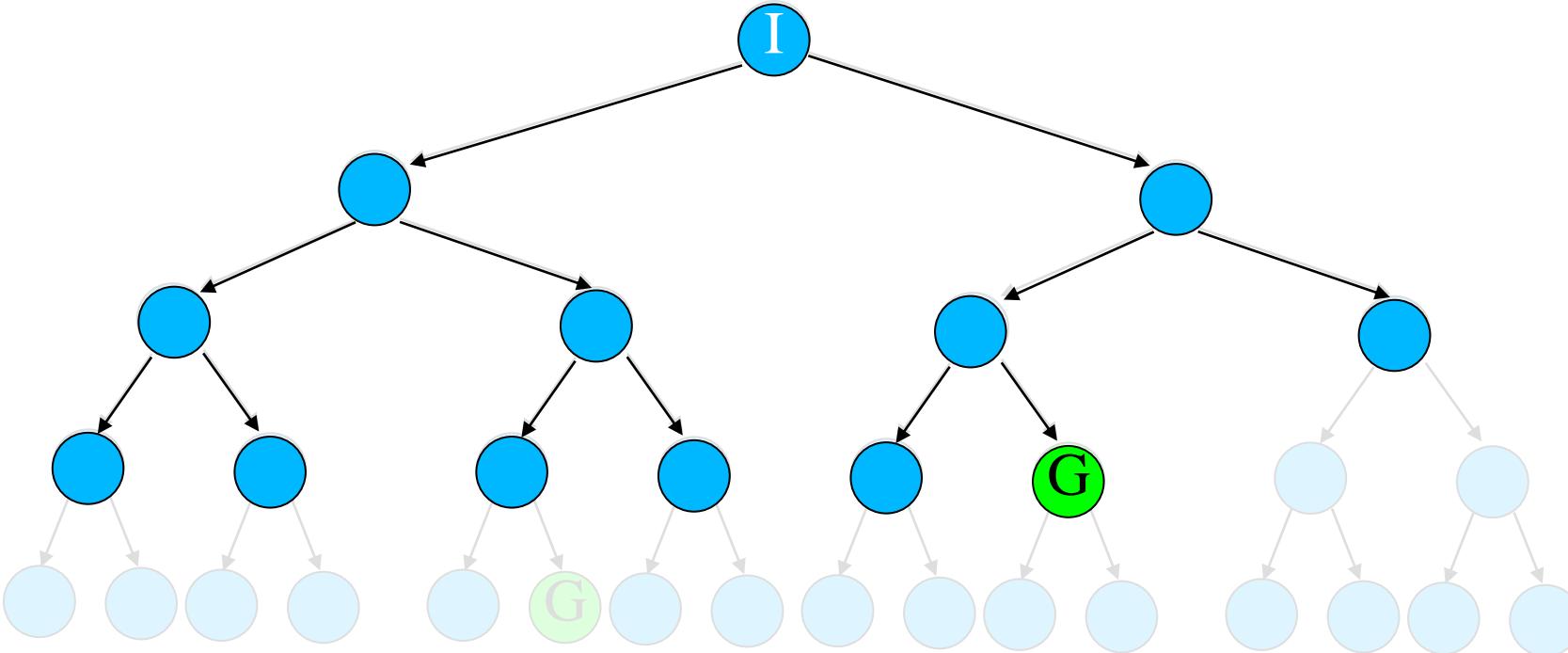


```
walk driver home amazon
board-truck driver truck amazon
load-truck mydvd truck amazon
drive-truck driver amazon london
drive-truck driver london myhouse
unload-truck mydvd myhouse
```

# Open list options

- What if open was a **queue**?
  - Can **push** states onto the back
  - Can **pop** states from the front
  - > **Breadth-first search**

# Breadth-First Search

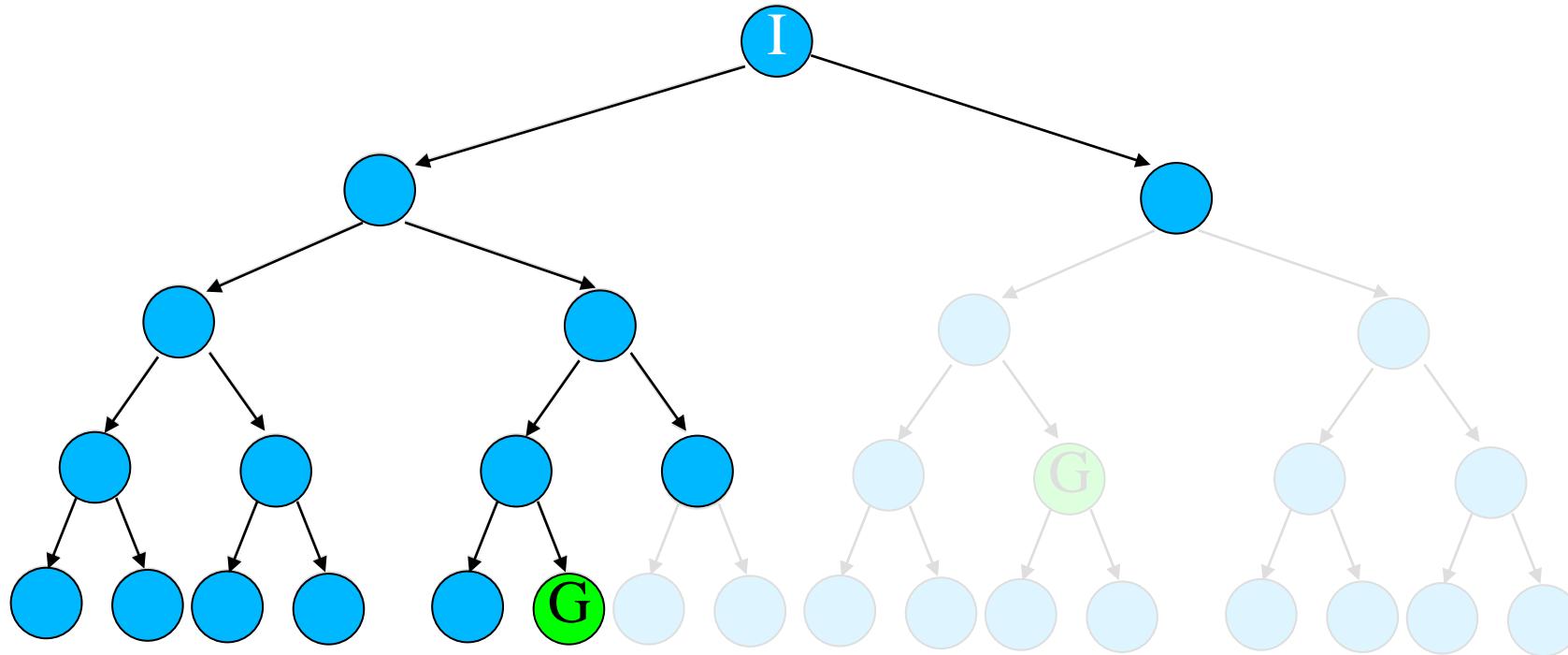


- Generate all the children of the current node.
  - Expand nodes in the order that they were generated.

# Open list options

- What if open was a **queue**?
  - Can **push** states onto the back
  - Can **pop** states from the front
  - > **Breadth-first search**
- What if open was a stack?
  - Can push/pop states from the front
  - > **Depth-first search**

# Depth-First Search



- Generate all children of the current node.
- If it has children select and expand its first child.
- If not, backtrack to the most recent node with unexpanded children.

# Open list options

- What if open was a **queue**?
  - Can **push** states onto the back
  - Can **pop** states from the front
  - > **Breadth-first search**
- What if open was a stack?
  - Can push/pop states from the front
  - > **Depth-first search**
  - (not generally used in planning)

# How long will this take?

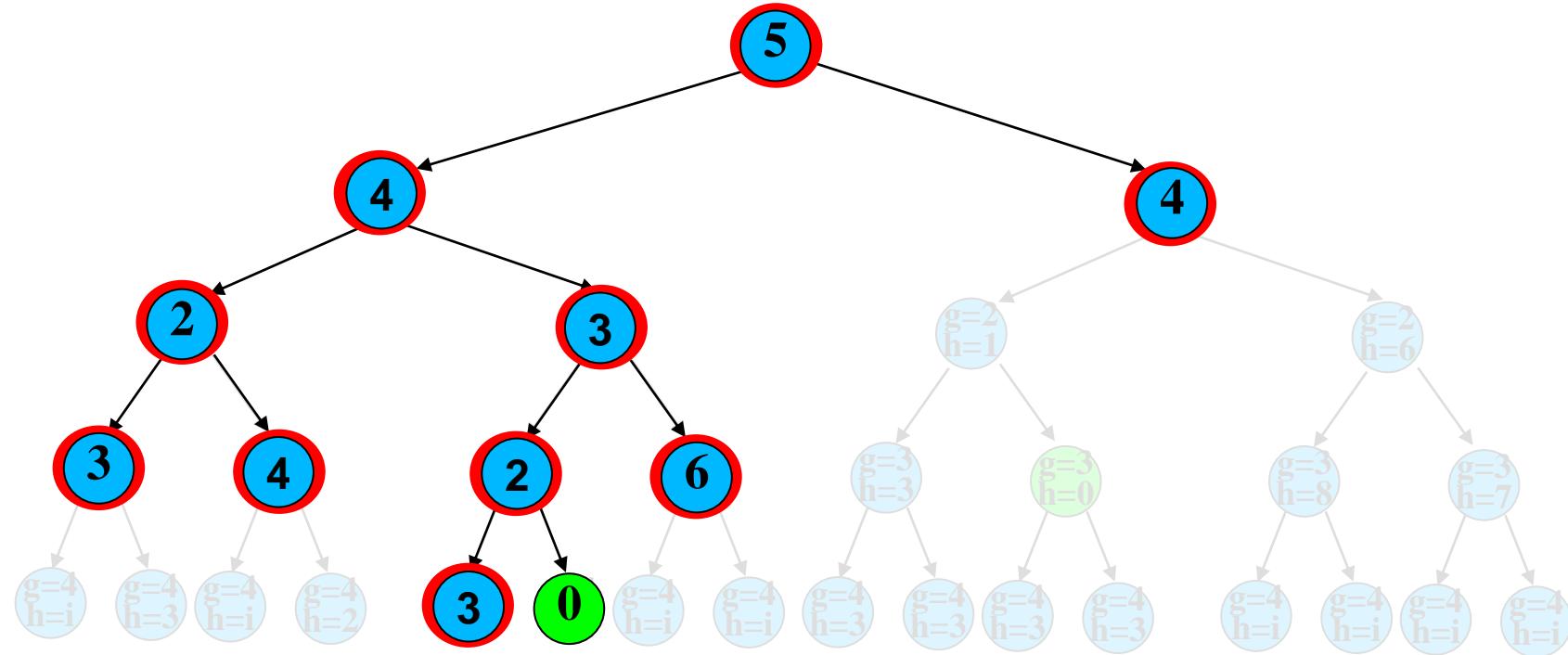
- Suppose:
  - 20 actions to choose from,
  - Solution plan is 20 steps long.
- Worst case: need to search through all  $20^{20}$  possibilities.
  - ( $> 1$  with 26 zeros after,  $10^{26}$ ).
  - $4.35 \times 10^{17}$  is the estimated age of the universe in seconds....
- Planning is PSPACE hard.
- Need to find a way to search through only the **most promising looking plans**.

# Planning as Heuristic Search

# Heuristic Search

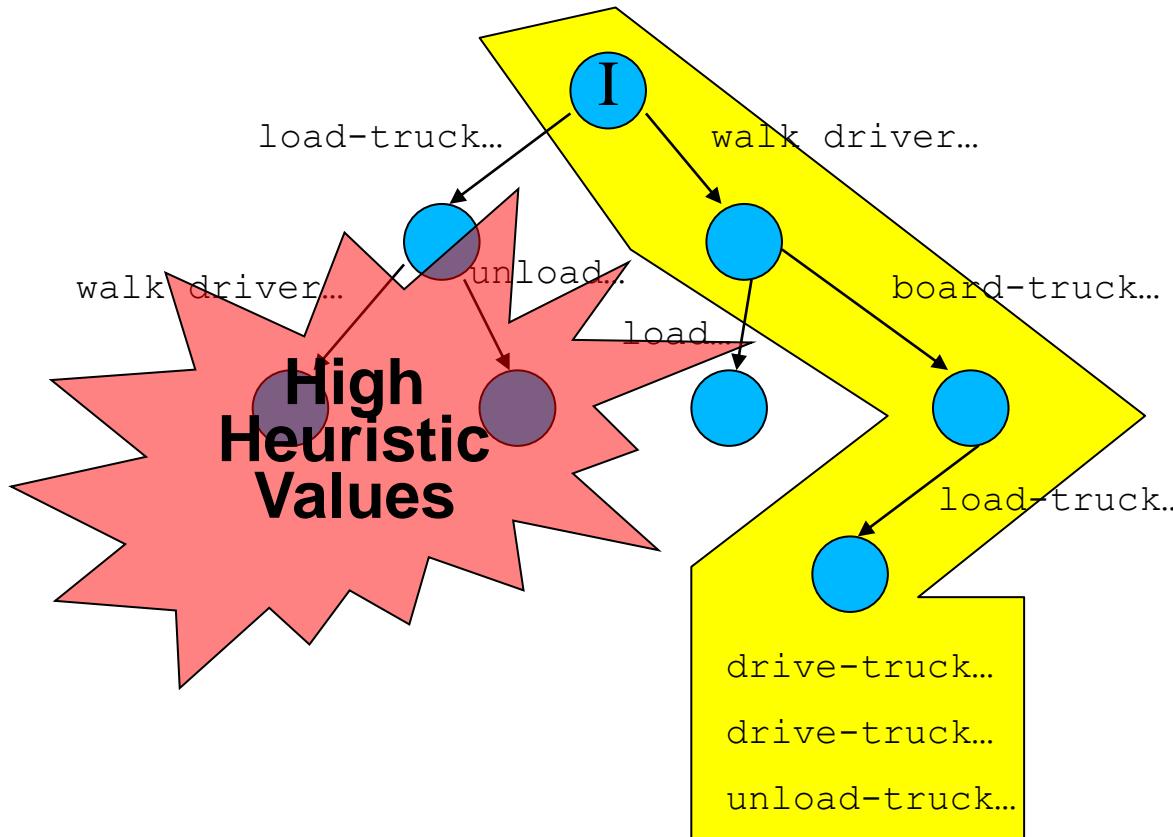
- A **Heuristic** is a function that takes a state and returns an **estimate of its distance from the goal**.
  - Smaller = better, as nearer to a goal state
- Heuristics are (usually) **not perfect**, or there is no search
- For now, let's assume we have some function  $h(S)$  that gives us a 'distance to go' from the state  $S$  to the goal:
  - For goal states,  $h(S) = 0$
  - For other states,  $h(S) > 0$

# Best-First Search



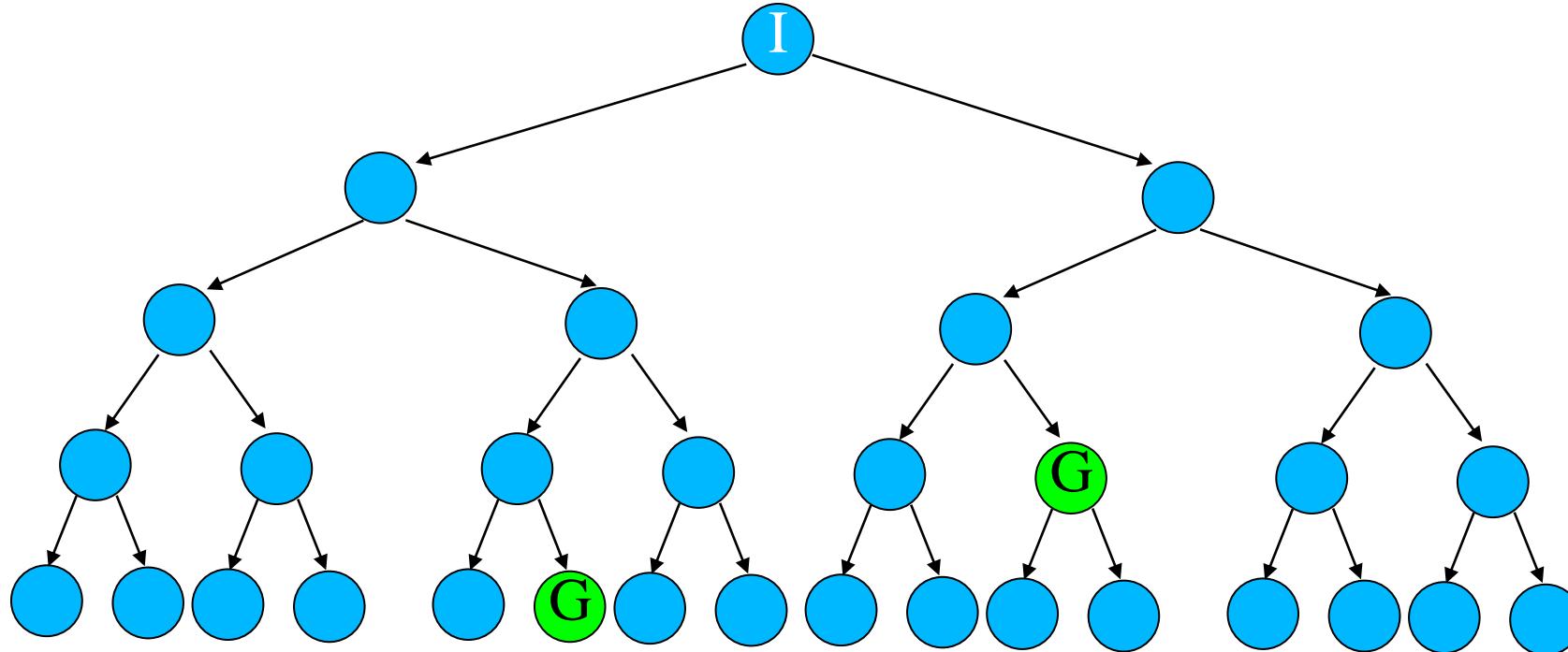
- Open list: priority queue with **stable insertion**

# An ideal world



**Decreasing  
Heuristic  
Values**

# What about plan length?



- The heuristic took us to the **left goal state** (4 plan steps)
- The other goal state has just 3 plan steps ☹

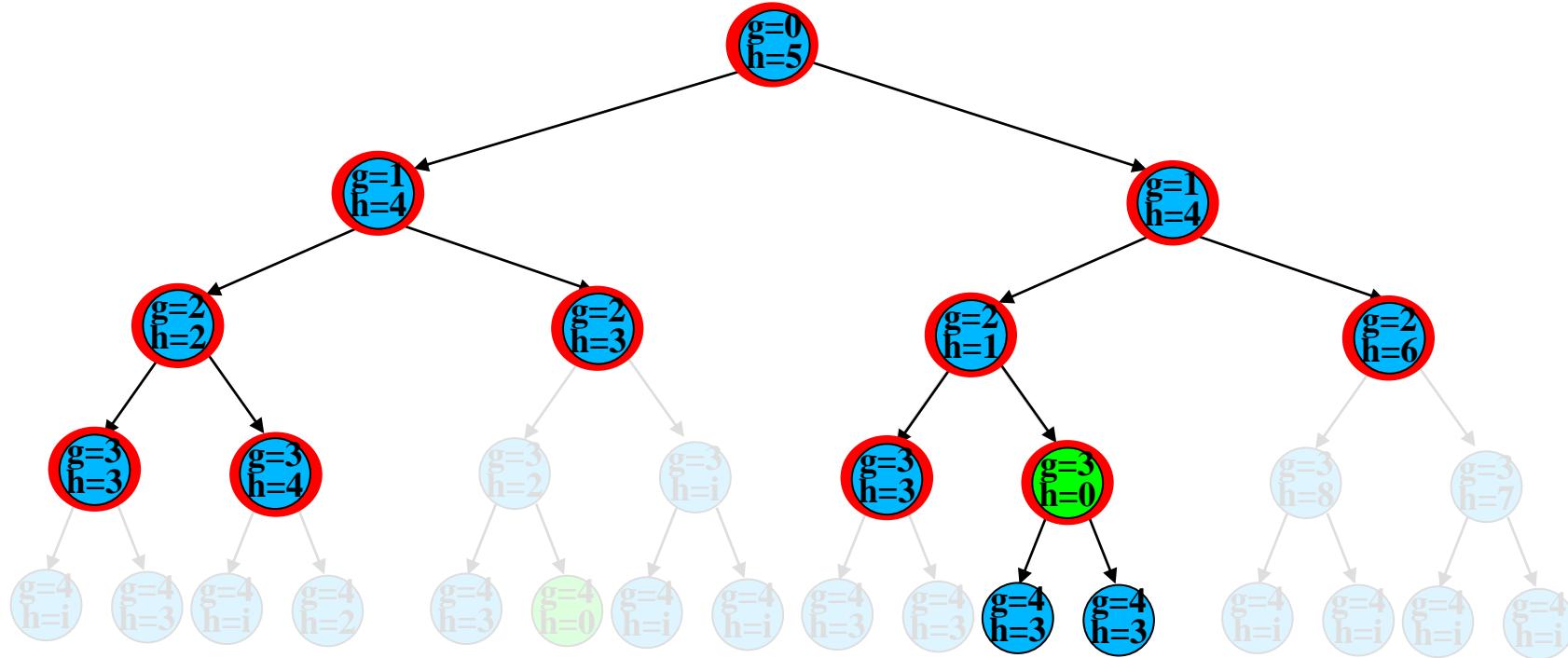
# What is ‘best’?

- Priority = smallest  $h(S)$  = ignore **plan so far**
- How about priority =  $f(S)$ , where

$$f(S) = g(S) + h(S)$$

- Intuition: distance so far + distance to goal estimates **eventual distance** for that option

# A\* Search



- Open list: priority queue on  $f(S) = g(S) + h(S)$

# Awesome proof (i)

- Q: What if  $h(S)$  is admissible and consistent?
  - **admissible**: never overestimates the distance to a goal state.
  - **consistent (monotonic)**:  $h(S) \leq h(S') + d(S, S')$
- A: at any point, the smallest  $f(S)$  on the open list under estimates ( $\leq$ )  
**the shortest possible plan** that can solve the problem
  - Proof:  $g(S)$  is perfect, so doesn't overestimate distance from  $I$  to  $(S)$ ;  $h(S)$  doesn't overestimate; so adding them together will never produce an overestimate.

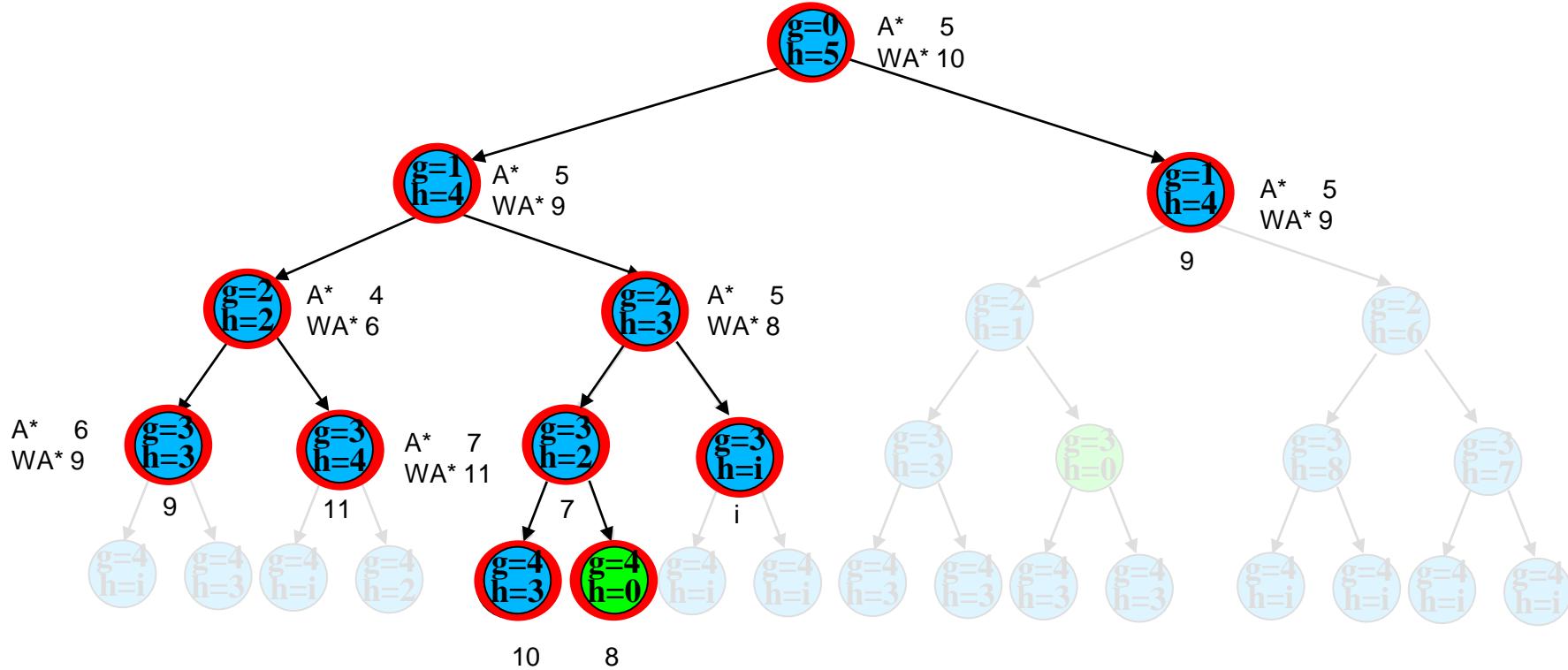
## Awesome proof (ii)

- Q: What if we find a goal state, but put it on the open list, and wait until it is **expanded**?
- A: we have found the **shortest possible plan**
  - Proof by contradiction: search expands a state with smallest  $f(S)$ ; if there was a shorter path to a goal state  $S'$ ,  $f(S')$  would be less than  $f(S)$ , and it would have been expanded first.
  - Consistency guarantees that there is no node that has not yet been put on the openlist that might have a shorter path than one that is there.

# Weighted A\* Search (WA\*)

- Q: What if we want a plan that is ‘good’ quality but not necessarily optimal?
- A: we can use **weighted A\***
  - $f(s) = g(s) + W * h(s);$
  - W is a constant that gives an optimality bound.
  - Now the solution we find will be within a factor W of optimal.
  - Find a solution faster by biasing search towards nodes closer to the goal.
  - Advantages:
    - Don’t have to explore as much of the search space to find a solution;
    - Still have some guarantee of the solution being a reasonable one.

# WA\* Search (W=2)



- Open list: priority queue on  $f(S) = g(S) + W^*h(S)$

# What is a good plan?

- Which plan is better?

(fly-helicopter strand barbican)

or...

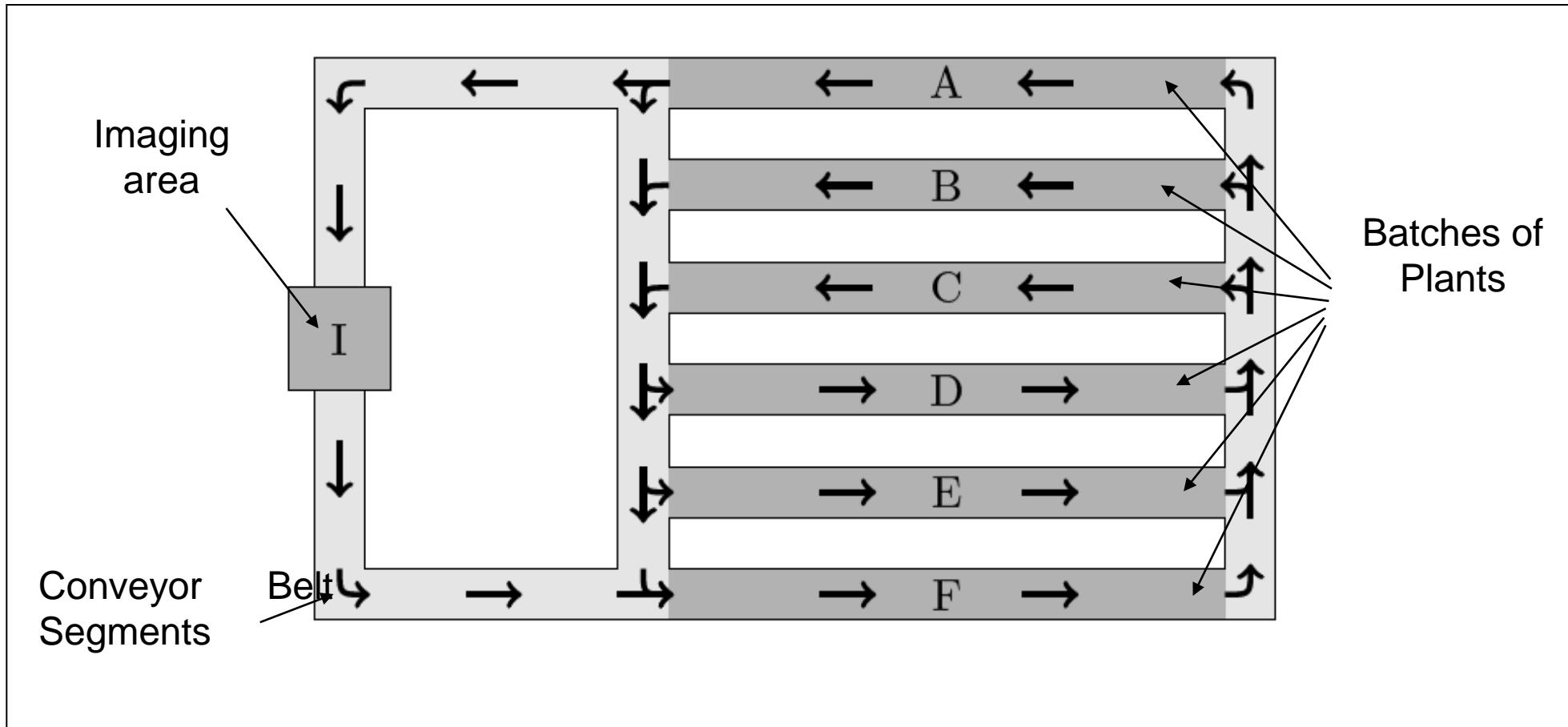
(walk strand temple)

(tube temple barbican)

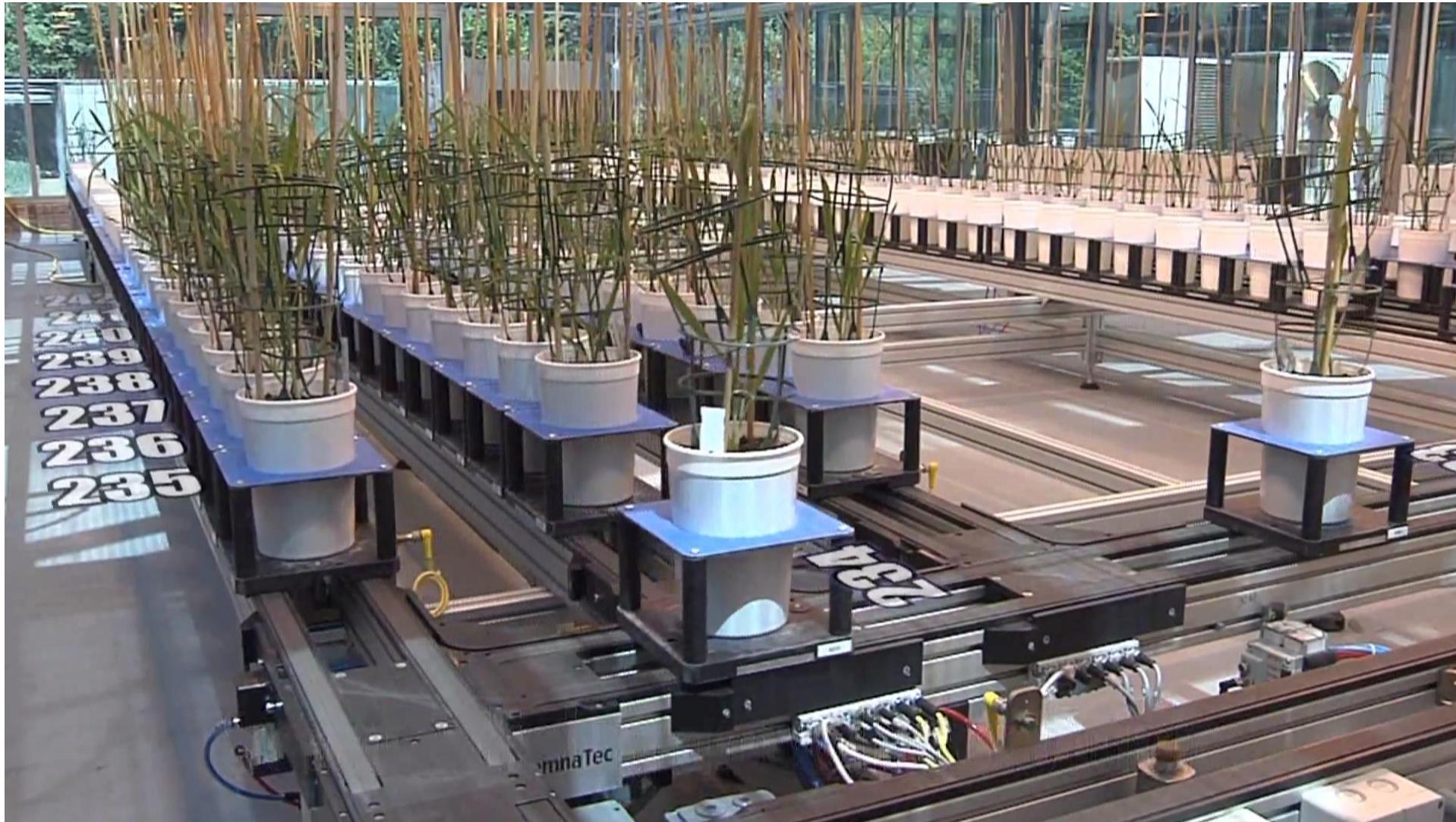
# Planning Application: The Scalyzer Domain

# Example: The Scalyzer Domain

- “The Scalyzer Domain: Greenhouse Logistics as a Planning Problem”, Helmert and Lasinger, ICAPS 2010



# Video



[http://www.youtube.com/watch?v=ovnwzzt\\_Xbs](http://www.youtube.com/watch?v=ovnwzzt_Xbs)

# Scanalyzer Types, Predicates

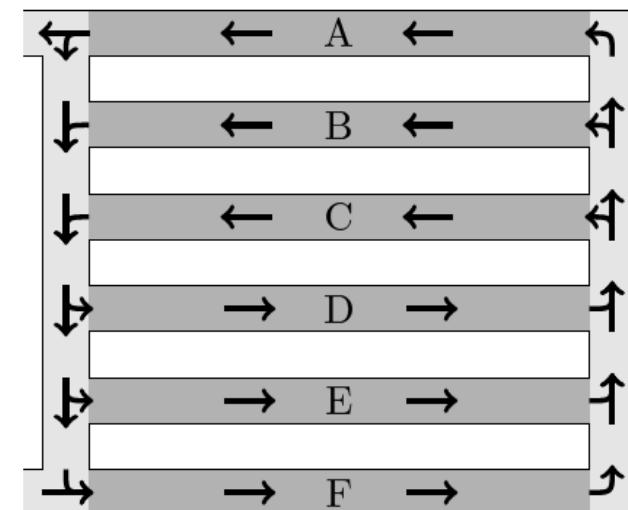
- Type of the conveyor belts: **segments**
- Type of the plants themselves: **batches**
  - **Operational constraint** – plants are moved one batch at a time, not individually.
- Predicates:
  - A batch of plants is **on** a given segment
  - A batch of plants has been **analyzed**
  - Segments are linked together in **cycles**

# Scanalyzer: Initial State

```
(:init (= (total-cost) 0)
(CYCLE-2 A D) (CYCLE-2 A E) (CYCLE-2 A F)
(CYCLE-2 B D) (CYCLE-2 B E) (CYCLE-2 B F)
(CYCLE-2 C D) (CYCLE-2 C E) (CYCLE-2 C F)
(CYCLE-2-WITH-ANALYSIS A F))
```

Static  
Facts

```
(on b1 A) (on b2 B) (on b3 C)
(on b4 D) (on b5 E) (on b6 F)
)
```



# Scanalyzer Actions in PDDL (i)

```
(:action rotate-2
:parameters (?s1 ?s2 - segment
             ?b1 ?b2 - batch)
:precondition (and (CYCLE-2 ?s1 ?s2)
                   (on ?b1 ?s1) (on ?b2 ?s2))
:effect (and
          (not (on ?b1 ?s1)) (on ?b1 ?s2)
          (not (on ?b2 ?s2)) (on ?b2 ?s1)
          (increase (total-cost) 1)))
```

Typed Parameters

Restricts pairs of segments that can be swapped

Swap two batches

Action has cost 1

# Scalyzer Actions in PDDL (ii)

```
(:action rotate-and-analyze-2
:parameters (?s1 ?s2 - segment
             ?b1 ?b2 - batch)
:precondition (and
               ((CYCLE-2-WITH-ANALYSIS ?s1 ?s2))
               (on ?b1 ?s1) (on ?b2 ?s2))
:effect (and (not (on ?b1 ?s1)) (on ?b1 ?s2)
            (not (on ?b2 ?s2)) (on ?b2 ?s1)
            (analyzed ?b1)
            (increase (total-cost) 3))))
```

In example: only the A—F conveyor goes through the imaging area

Mark batch as being  
analyzed

# Scanalyzer: The Goal

```
(:goal (and  
       (analyzed b1) (analyzed b2) (analyzed b3)  
       (analyzed b4) (analyzed b5) (analyzed b6)  
       (on b1 A) (on b2 B) (on b3 C)  
       (on b4 D) (on b5 E) (on b6 F)) )
```

Analyse  
Batches

Return to  
Initial Positions

```
(:metric minimize (total-cost))
```

Prefer plans with lower costs:  
rotate-2 costs 1, rotate-and-analyze costs 3.

# Scalyzer: Demo

```
0.000: (analyze-2 seg-in-1 seg-out-1 car-in-1 car-out-1) [0.001]
0.001: (analyze-2 seg-in-1 seg-out-1 car-out-1 car-in-1) [0.001]
0.002: (analyze-2 seg-in-3 seg-out-1 car-in-3 car-out-1) [0.001]
0.002: (rotate-2 seg-in-1 seg-out-2 car-in-1 car-out-2) [0.001]
0.003: (rotate-2 seg-in-3 seg-out-1 car-out-1 car-in-3) [0.001]
0.004: (analyze-2 seg-in-2 seg-out-1 car-in-2 car-out-1) [0.001]
0.005: (rotate-2 seg-in-2 seg-out-1 car-out-1 car-in-2) [0.001]
0.006: (analyze-2 seg-in-1 seg-out-1 car-out-2 car-out-1) [0.001]
0.007: (rotate-2 seg-in-1 seg-out-1 car-out-1 car-out-2) [0.001]
0.008: (rotate-2 seg-in-1 seg-out-2 car-out-2 car-in-1) [0.001]
0.009: (rotate-2 seg-in-1 seg-out-3 car-in-1 car-out-3) [0.001]
0.010: (analyze-2 seg-in-1 seg-out-1 car-out-3 car-out-1) [0.001]
0.011: (rotate-2 seg-in-1 seg-out-1 car-out-1 car-out-3) [0.001]
0.012: (rotate-2 seg-in-1 seg-out-3 car-out-3 car-in-1) [0.001]
```

# Scalyzer: Planner Performance

DSS: A\* search + heuristic **just for Scalyzer**

IPC: **domain independent** planning competition planners

Layout 1			Layout 2			Layout 3			Layout 4		
Size	DSS	IPC									
6	<b>18*</b>	<b>18*</b>	6	<b>22*</b>	<b>22*</b>	6	<b>26*</b>	<b>26*</b>	6	<b>22*</b>	<b>22*</b>
8	<b>24*</b>	<b>24*</b>	8	<b>30*</b>	<b>30*</b>	8	<b>36*</b>	<b>36*</b>	8	32	<b>30*</b>
10	<b>30*</b>	<b>30*</b>	10	<b>38*</b>	44	10	<b>46*</b>	<b>46*</b>	10	<b>40</b>	44
12	<b>36*</b>	<b>36*</b>	12	<b>46*</b>	54	12	<b>56*</b>	60	12	—	<b>56</b>
14	<b>42*</b>	<b>42*</b>	14	<b>54*</b>	64	14	<b>66*</b>	72	14	—	<b>66</b>
16	<b>48*</b>	<b>48*</b>	16	<b>62*</b>	74	16	—	<b>86</b>	16	—	<b>84</b>
18	<b>54*</b>	<b>54*</b>	18	—	<b>84</b>	18	—	<b>94</b>	18	—	<b>94</b>
20	<b>60*</b>	<b>60*</b>	20	—	<b>94</b>	20	—	<b>108</b>	20	—	<b>106</b>
22	<b>66*</b>	<b>66*</b>	22	—	<b>104</b>	22	—	<b>114</b>	22	—	<b>116</b>
24	<b>72*</b>	<b>72*</b>	24	—	<b>114</b>	24	—	<b>134</b>	24	—	<b>128</b>

# Classical Planning Planning Under Uncertainty Markov Decision Process

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Classical Planning: Material Overview

- **Classical Planning: The fundamentals.**

- **Non-Forward Search**

- **Improving Search**

- Heuristic Design (RPG/LAMA)

- Dual Openlist Search

- Graphplan

- SAT Planning

- POP Planning

- HTN Planning

- **Optimal Planning**

- SAS+ Planning

- Pattern Databases

- **Planning Under Uncertainty**



# Fundamentals: Forward Search

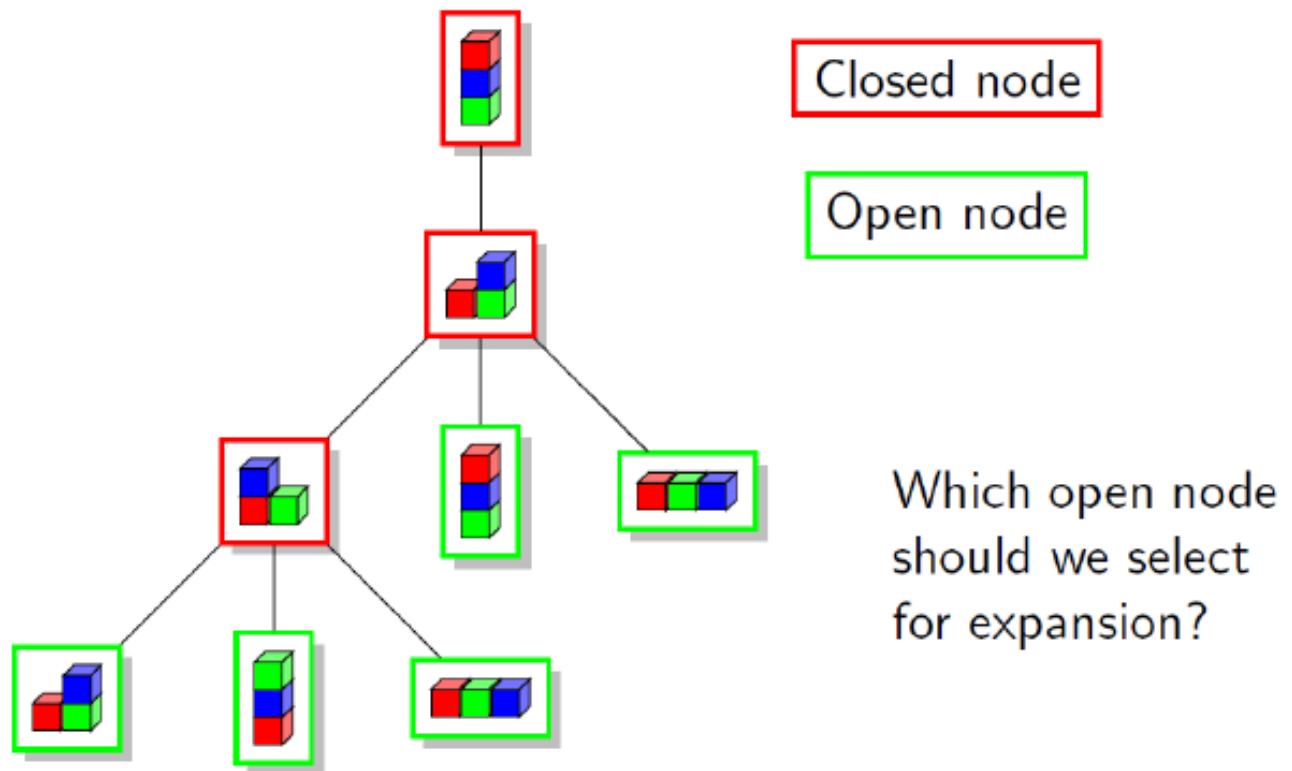
- **Definition of the problem – PDDL**

- **Forward Search**

- Breadth/Depth First

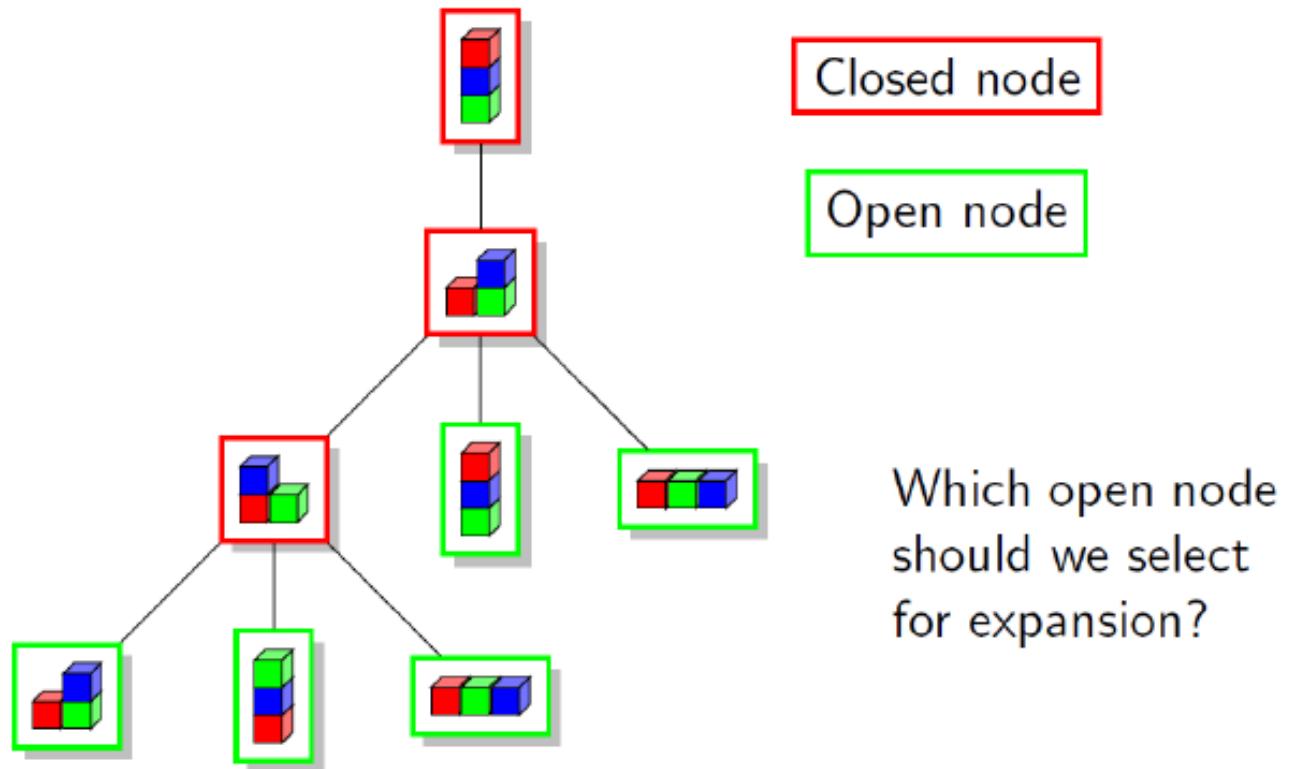
- **Introducing Heuristics**

- Best First Search/A\*



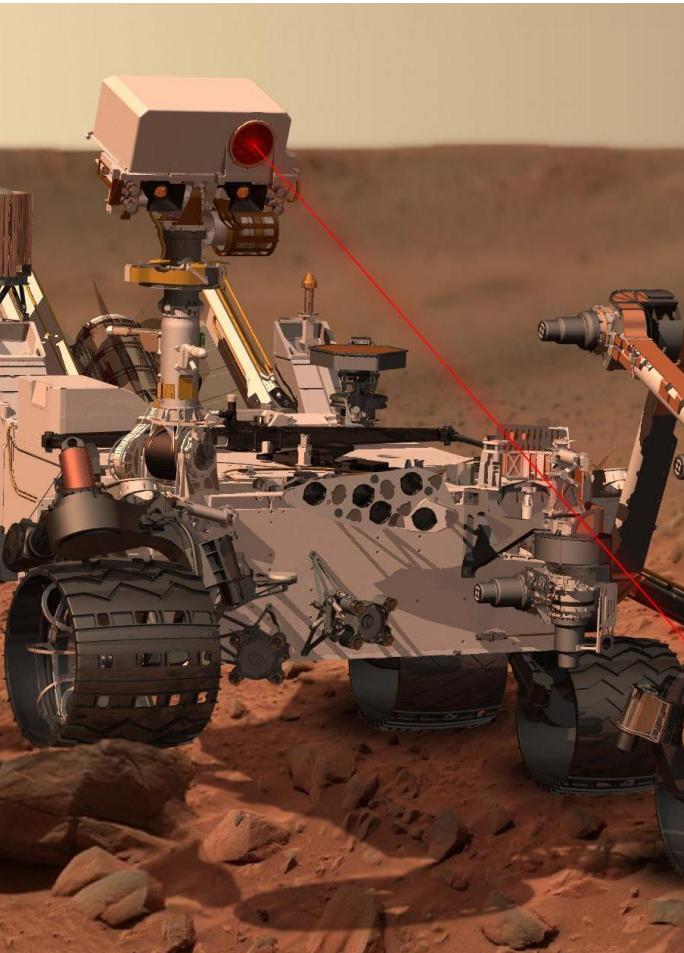
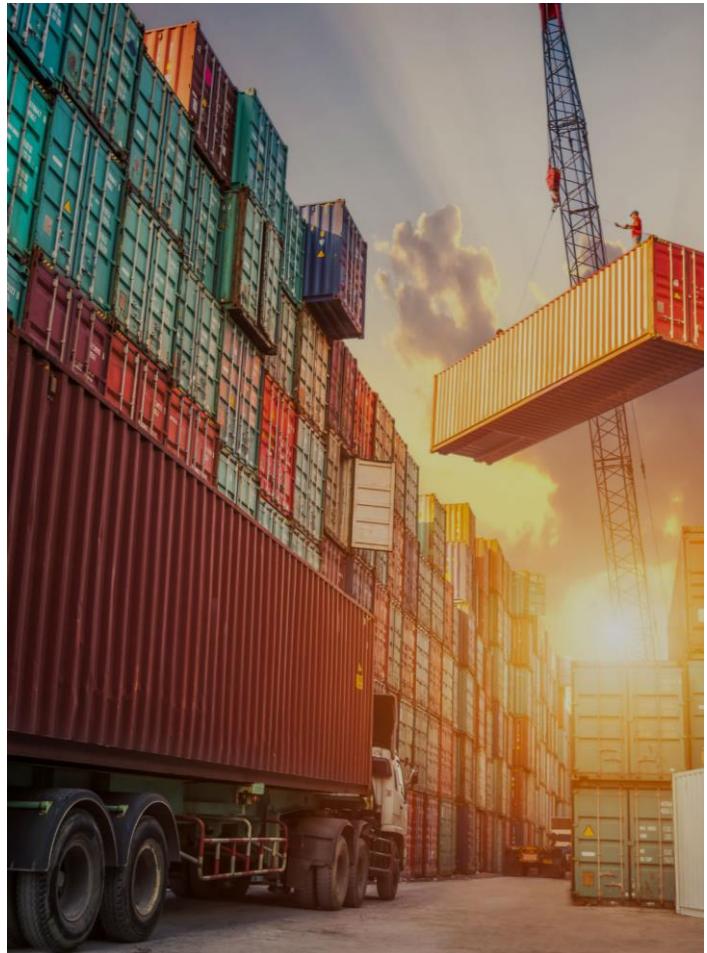
# Planning Under Uncertainty

- Issues that can arise include...
- We might not know which state we are in.
  - i.e. **Partial observability**
- We don't know if actions will succeed as intended.
  - i.e. **Non-Deterministic Actions**
- Actions might not execute as intended, resulting in possible new effects.
  - i.e. **Non-Deterministic Effects**



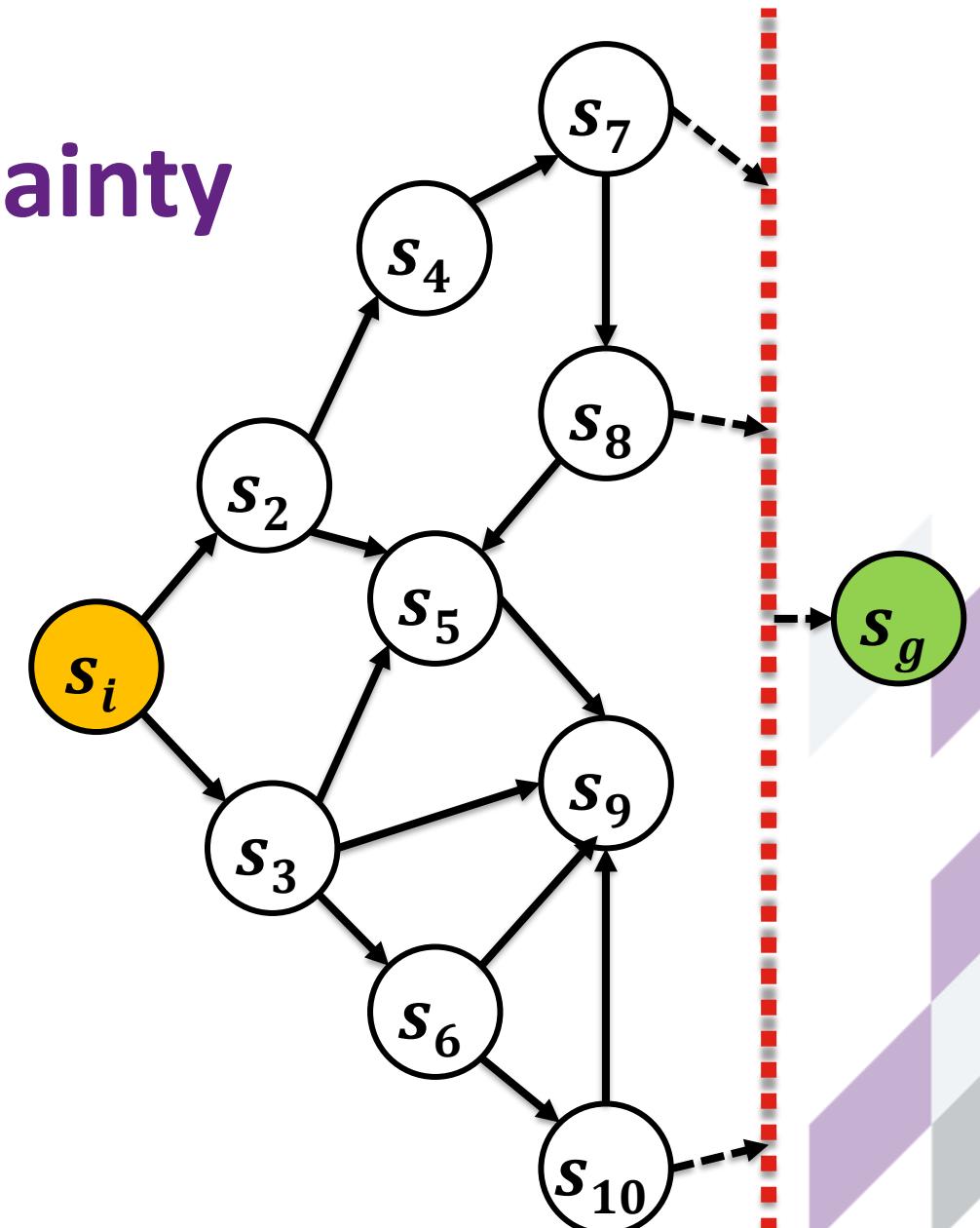
“The best laid schemes o' mice an' men / Gang aft a-gley.”  
“To a Mouse”, Robert Burns, 1785

# Why is this important?



# Classical Planning = No Uncertainty

- We assume the state is known.
- We assume all states in the problem are known.
- We assume that we know all actions from the current state.
- We assume that a given action will always execute as intended.
- We assume that the successor states of action execution are also known.



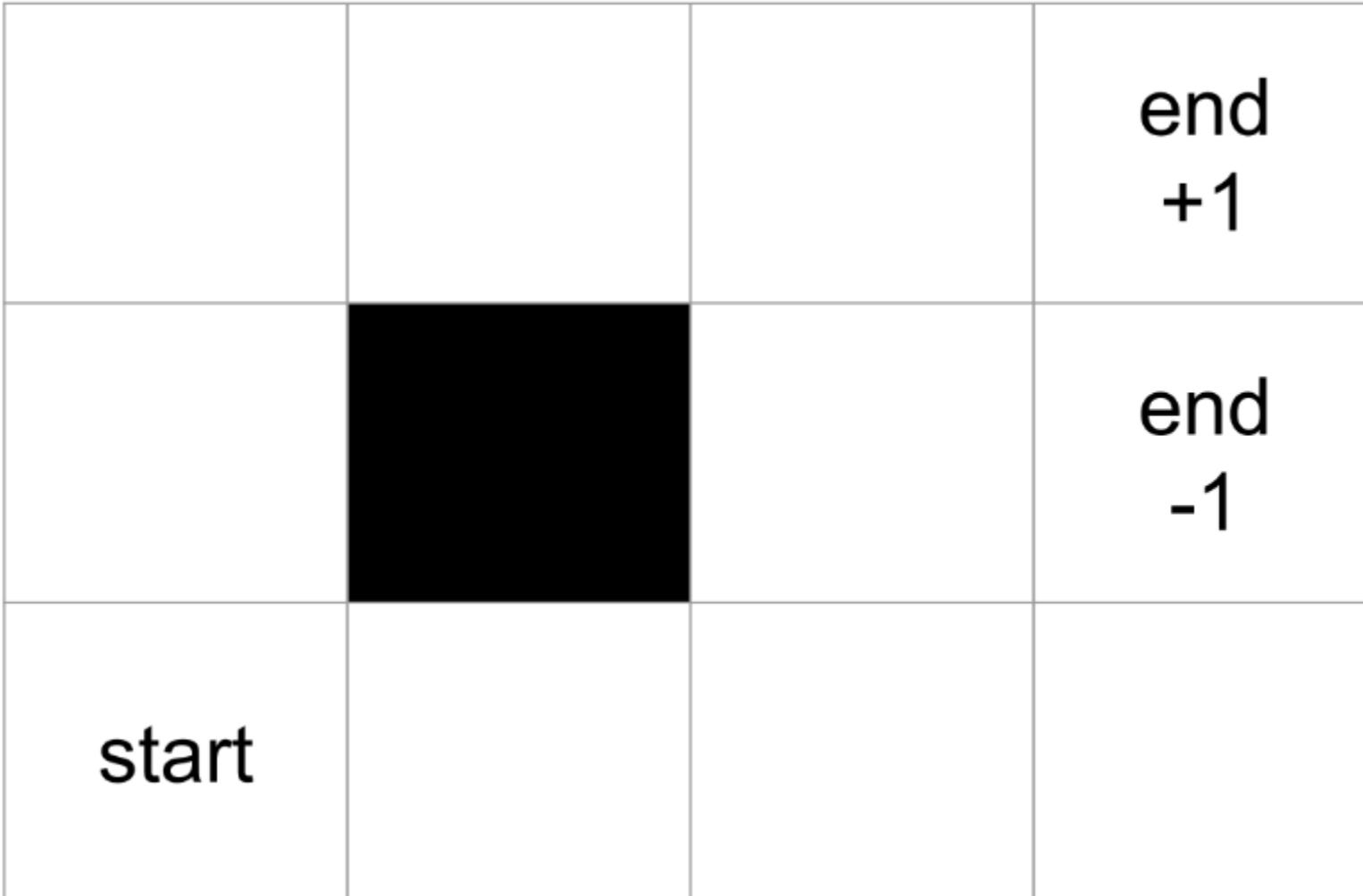
# Classical Planning = No Uncertainty

- We assume the state is known.
- We assume all states in the problem are known.
- We assume that we know all actions from the current state.
- **We assume that a given action will always execute as intended.**
- **We assume that the successor states of action execution are also known.**



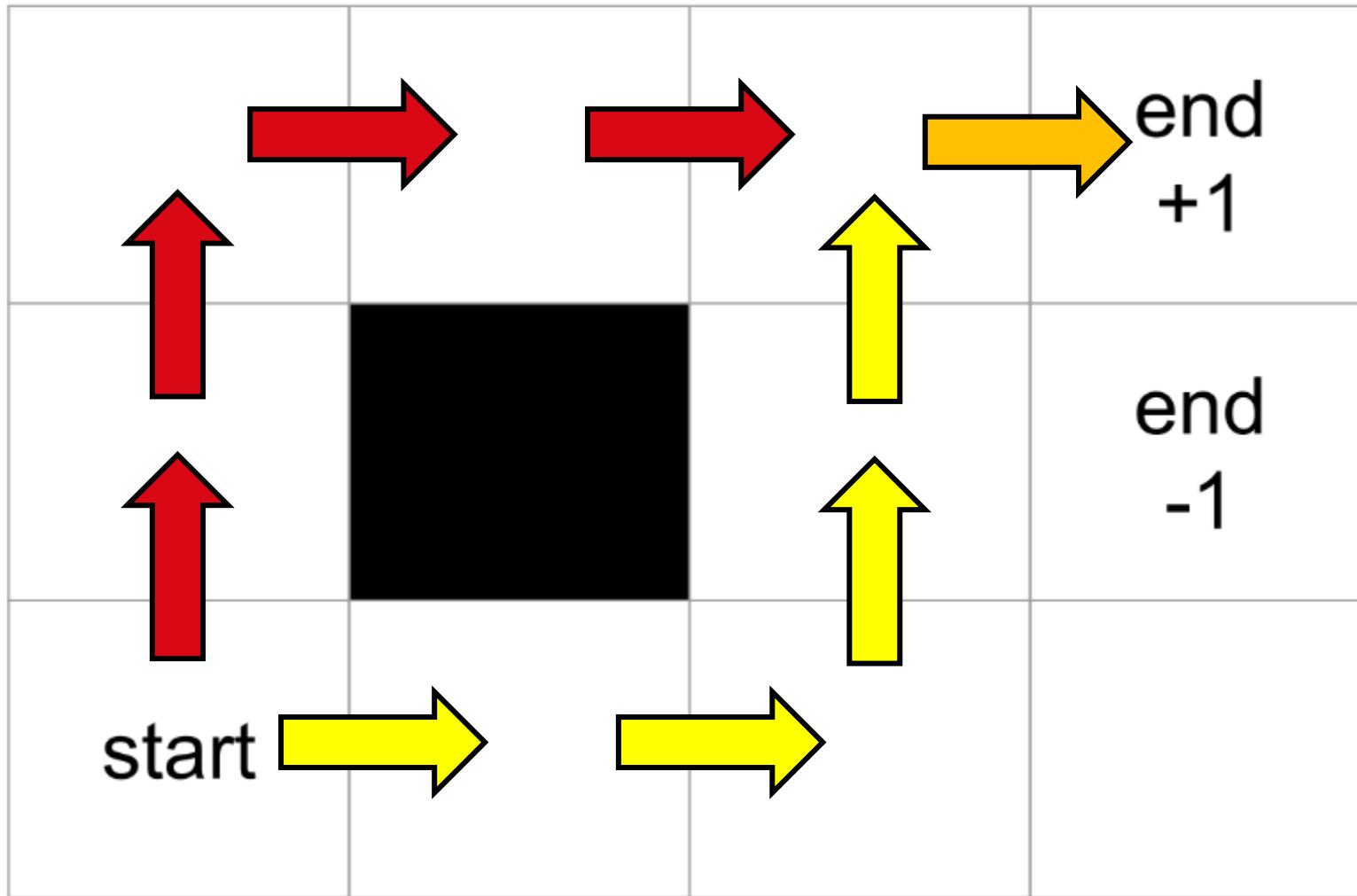
# Example

- Navigating in a maze.
- Simple to solve using uninformed or heuristic-driven search.
- But immediately becomes more challenging when the problem space is not deterministic.



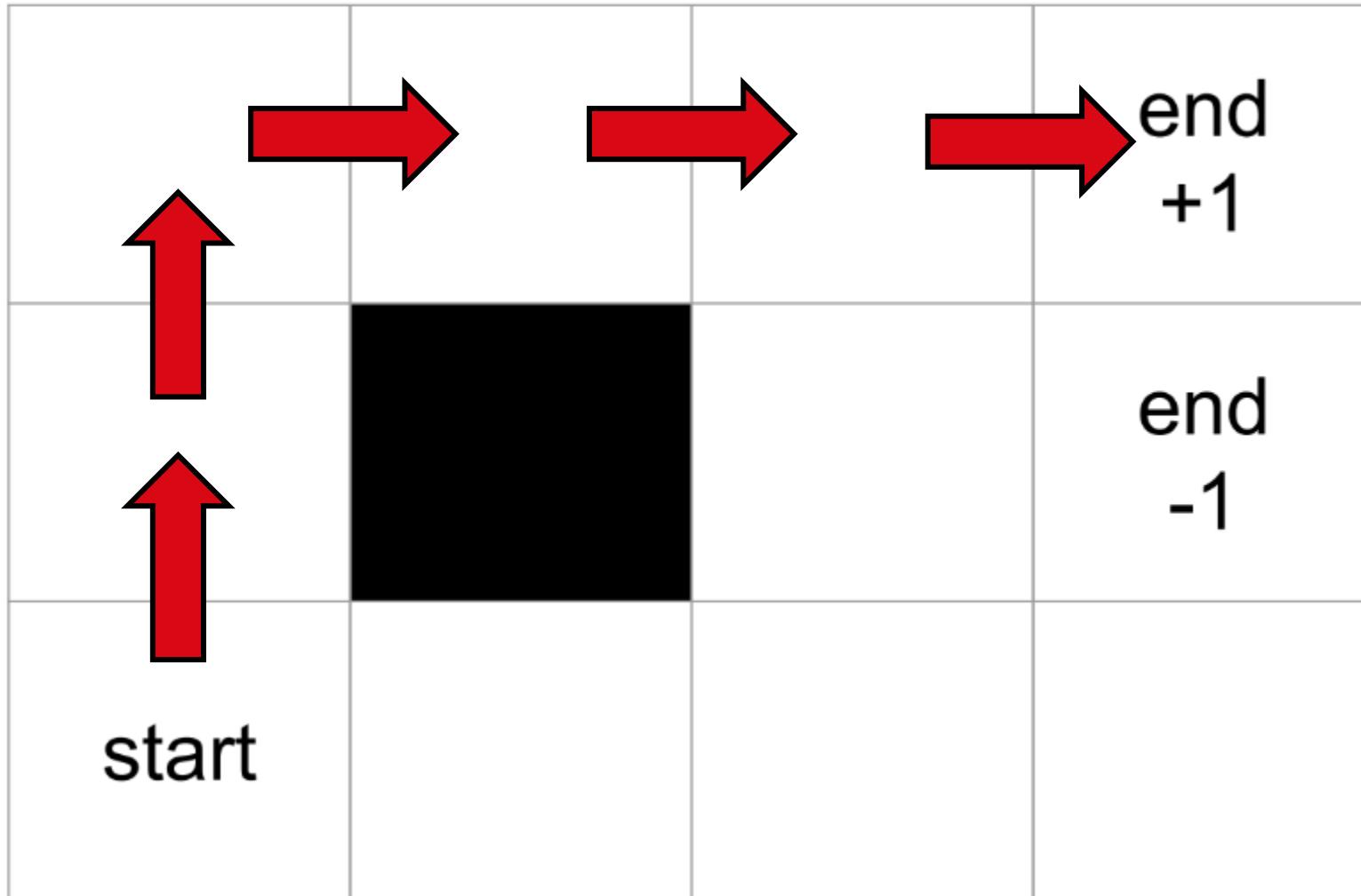
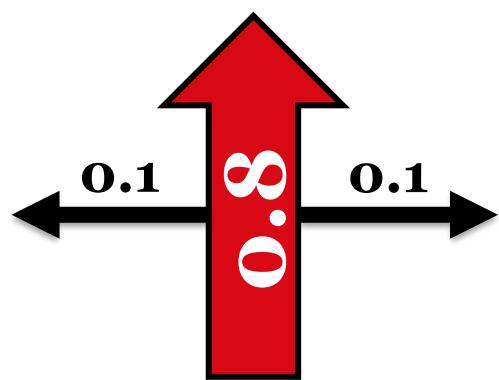
# Example

- Navigating in a maze.
- Simple to solve using uninformed or heuristic-driven search.
- But immediately becomes more challenging when the problem space is not deterministic.



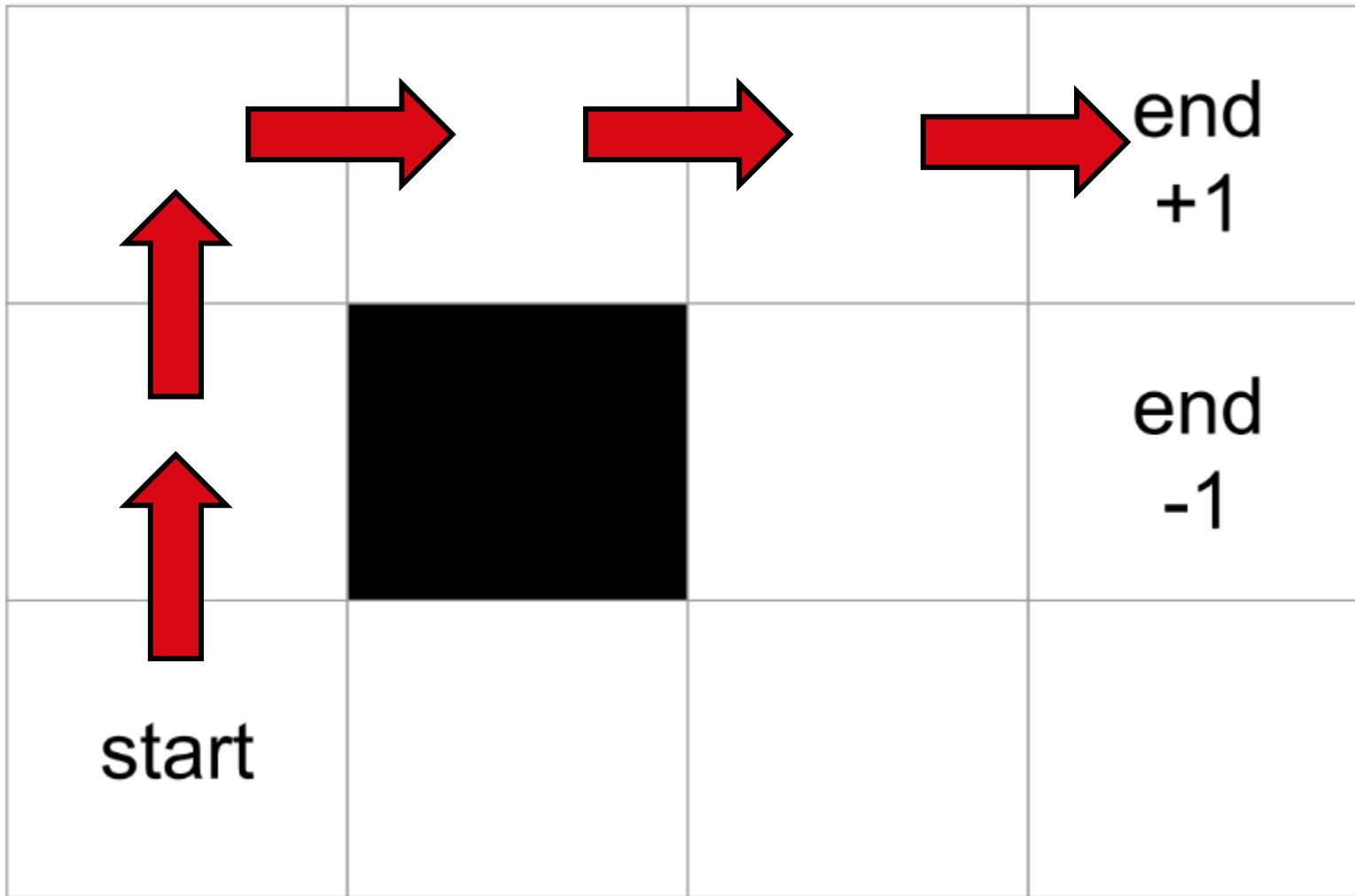
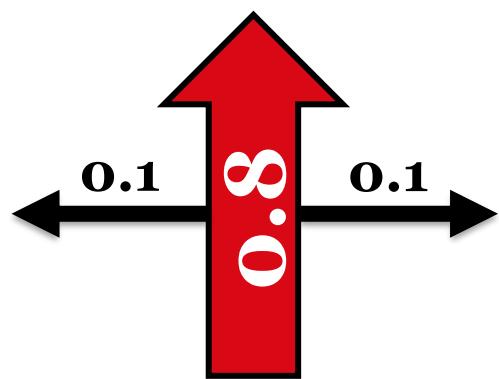
# Example

- Non-deterministic actions mean that the state transition system is no longer consistent.



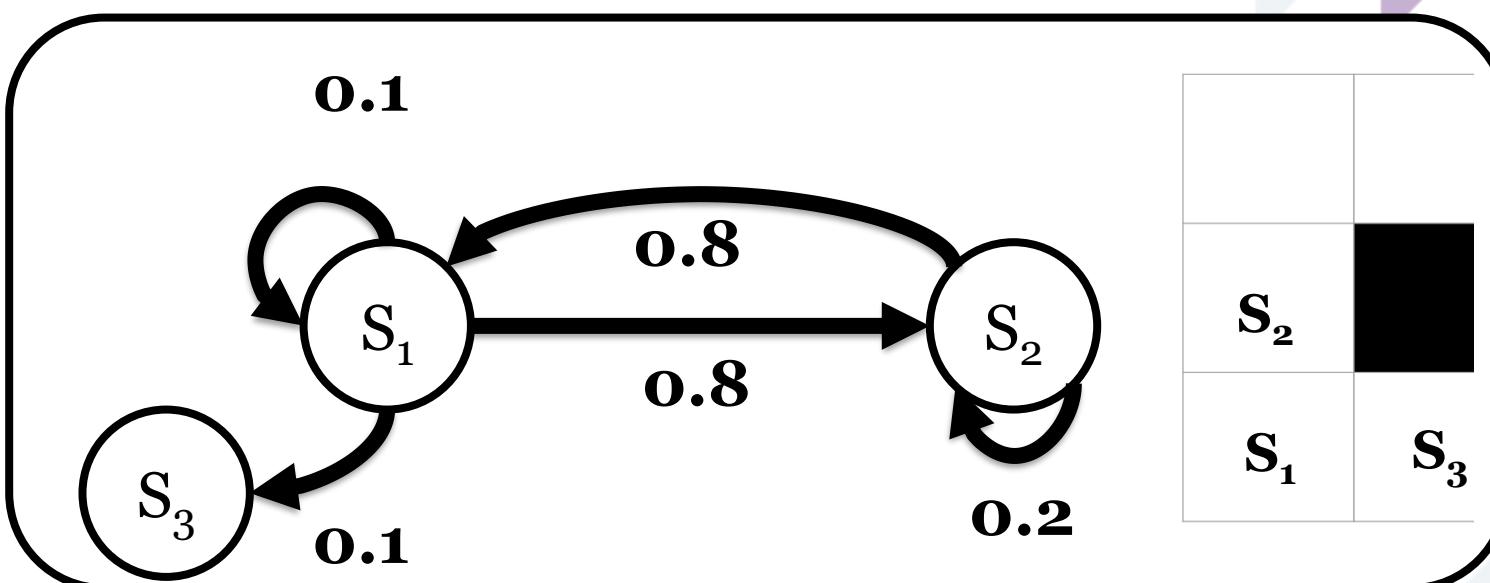
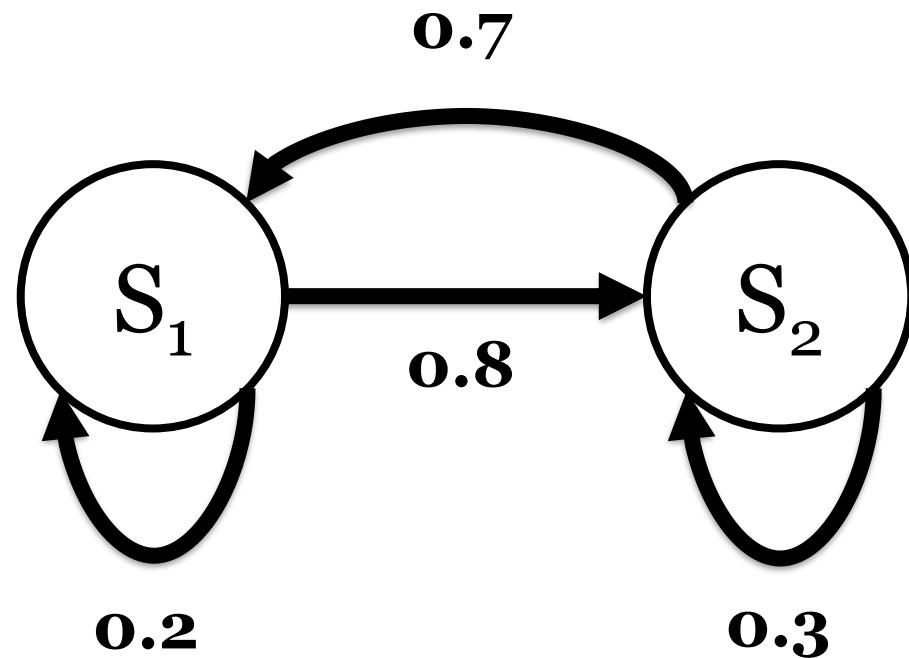
# Example

- Non-deterministic actions mean that the state transition system is no longer consistent.
- Original Plan:
  - Up, Up, Right, Right, Right
  - Success? <33%



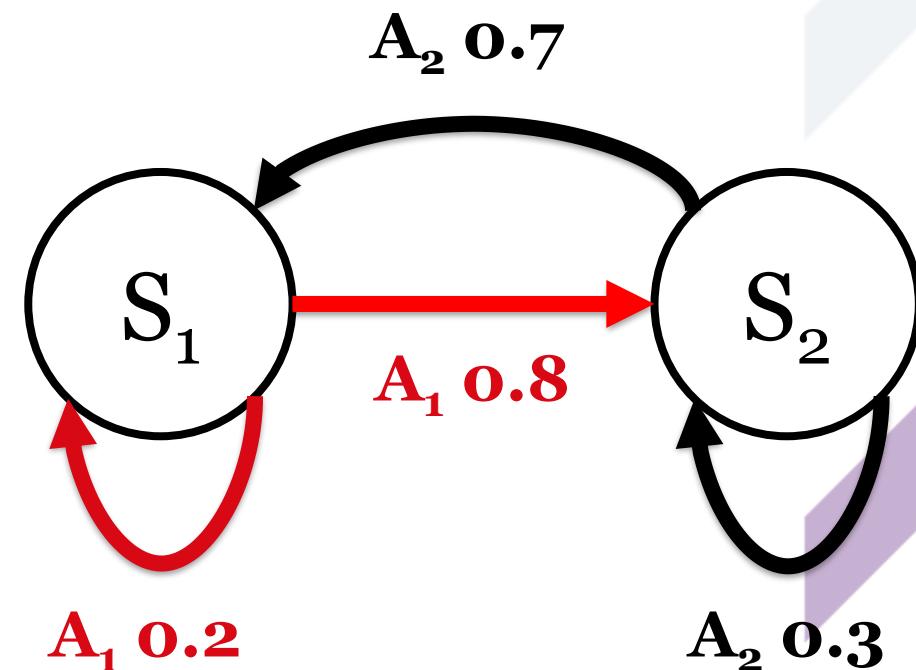
# Markov Chains

- Modelling probabilistic transitions between states.
- Next state is only determined by probabilistic transition between states.
- Does not factor the history of previous states.
  - Known as the Markov Property.



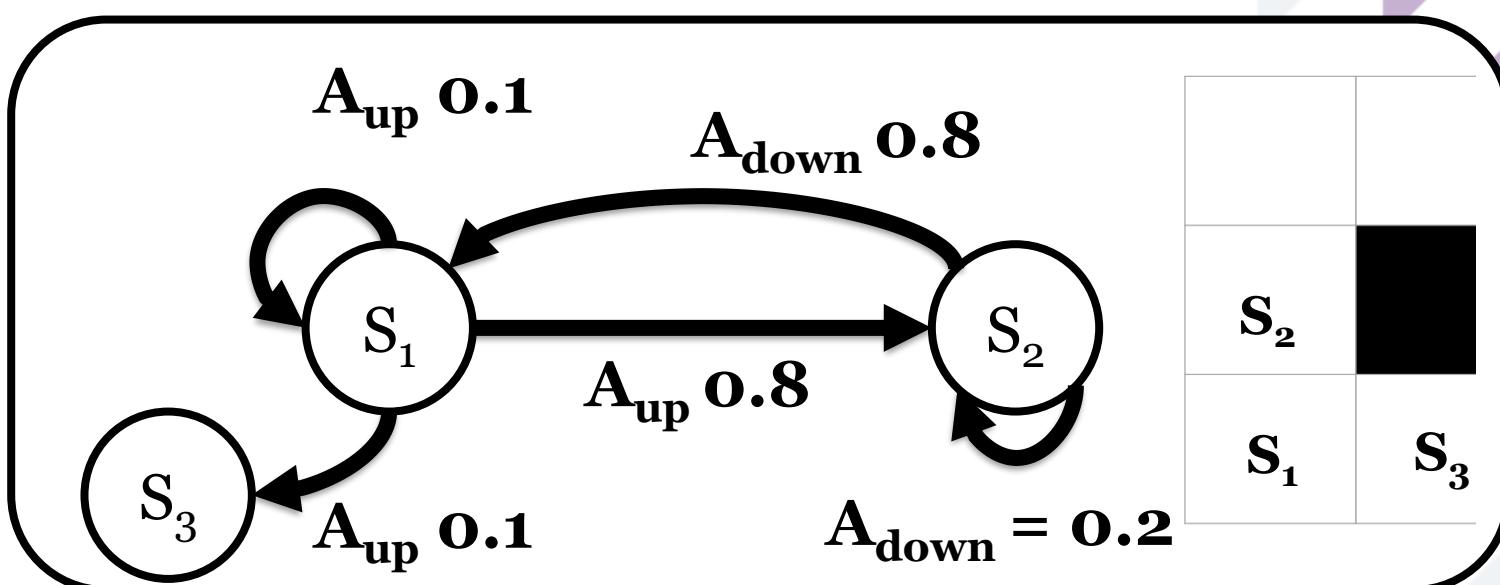
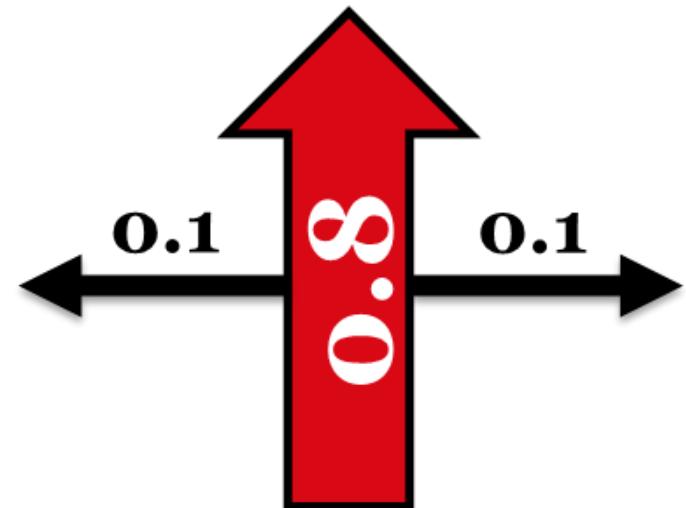
# Markov Decision Process (MDP)

- A sequential decision making problem for a fully observable but stochastic environment.
- Markov Chain holds, but now with probability of actions between states and rewards for action execution.
- Reliant on two principles:
  - Assume that Markov property holds for action transitions (which it should).
  - Probability distributions are stationary and don't change.



# Markov Decision Process (MDP)

- MDP is now built around transition model:  $T(s, a, s')$
- Similar to existing state transition, but now encodes  $P(s' | s, a)$
- But how can we then determine value of an action against the probability distribution?
- We will need a utility function, that will help calculate this based on state rewards.



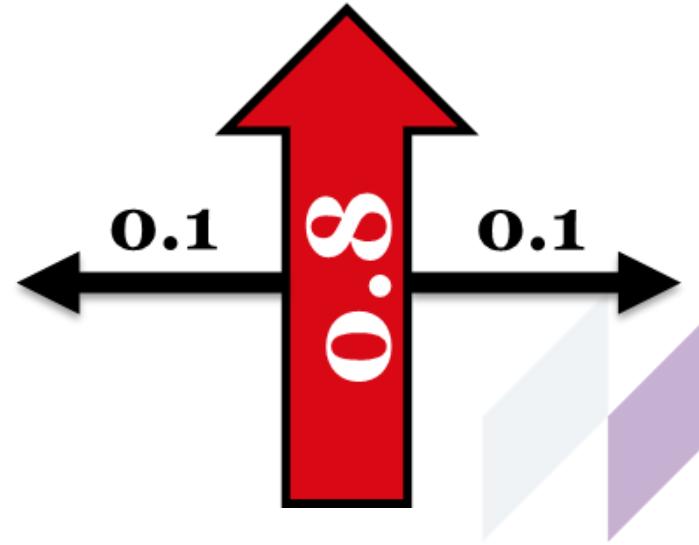
# MDP Example

- We now encode a reward for each state.
- Total utility of the agent is the sum of all rewards from all states visited.
- Assuming our original solution
  - Up, Up, Right, Right, Right
  - $(5 \times -0.04) + 1 = 0.8$
- Negative reward incentivises optimality.

-0.04	-0.04	-0.04	end +1
-0.04		-0.04	end -1
-0.04		-0.04	
-0.04	-0.04	-0.04	-0.04

# Markov Decision Process (MDP)

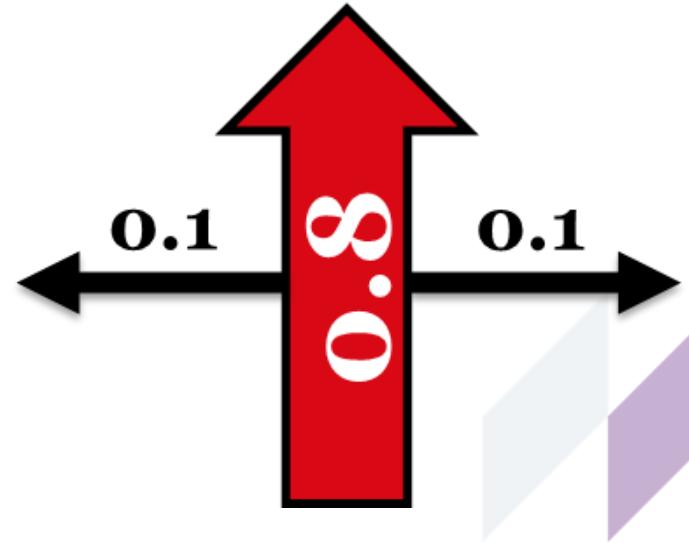
- MDP components for all states  $s \in S$ 
  - Transition Model:  $T(s, a, s')$
  - Initial State of the Problem:  $s_0$
  - Reward Function for a given state:  $R(s)$



-0.04	-0.04	-0.04	end +1
-0.04		-0.04	end -1
start	-0.04	-0.04	-0.04

# Markov Decision Process (MDP)

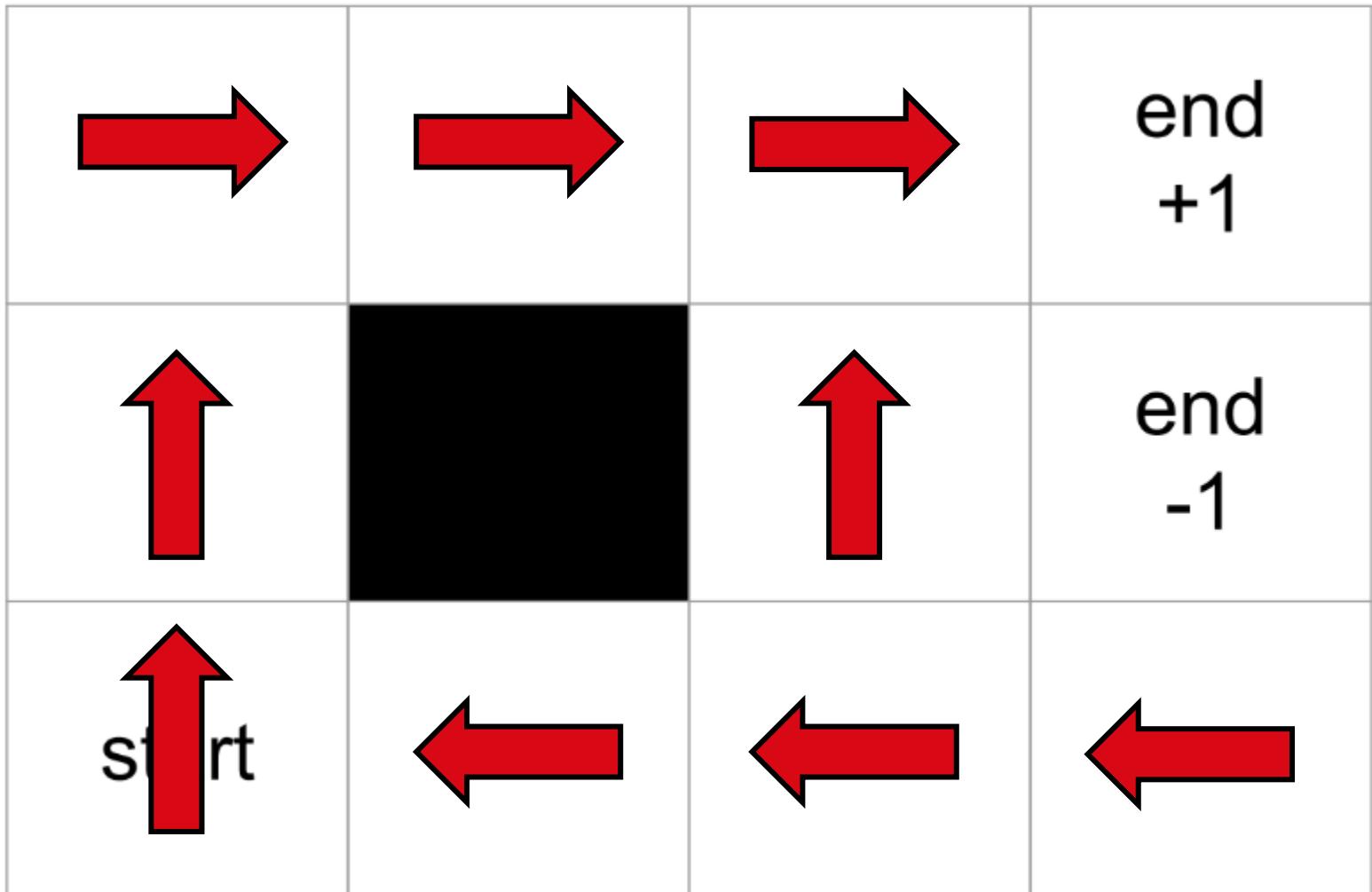
- MDP components for all states  $s \in S$ 
  - Transition Model:  $T(s, a, s')$
  - Initial State of the Problem:  $s_0$
  - Reward Function for a given state:  $R(s)$
- MDP solution is not a plan of actions.
- MDP solution is a policy:  $\pi$ 
  - For any state  $s \in S$ ,  $\pi(s)$  denotes what action should be taken in that state.
  - This means you can different resulting solutions from the initial to goal state, given the stochastic nature of the environment.
  - Hence the optimal policy ( $\pi^*$ ) will yield the highest expected utility in a given situation.



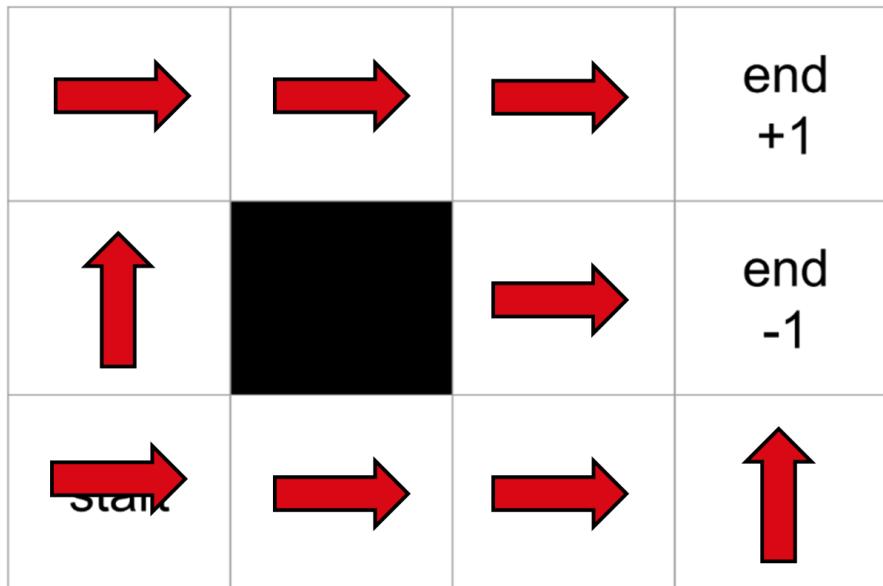
-0.04	-0.04	-0.04	end +1
-0.04		-0.04	end -1
start	-0.04	-0.04	-0.04

# MDP Example

- Optimal policy attempts to balance the risk against the reward.
- Results in actions that will yield the best utility regardless of how successful we are in execution.
- Highly sensitive to the action probabilities.

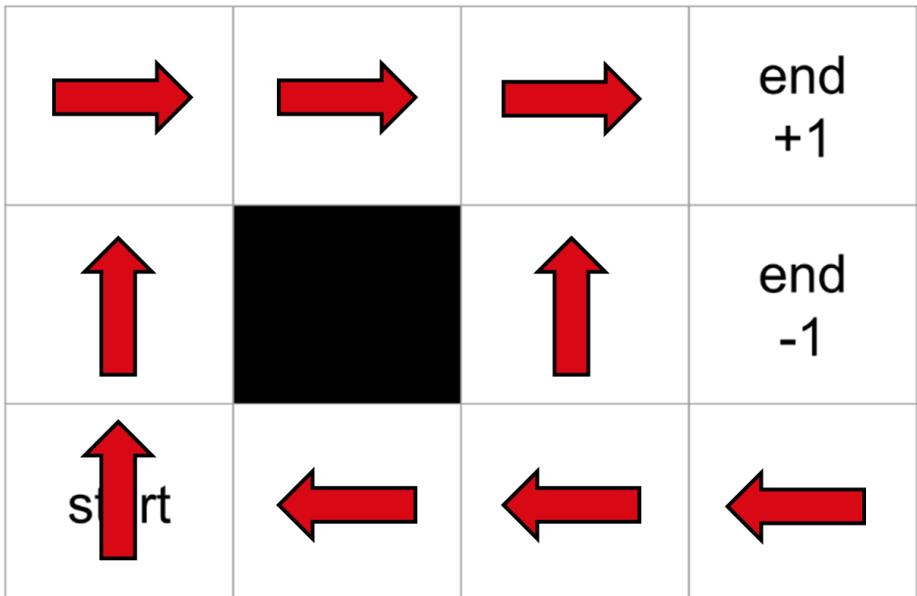


# MDP Example

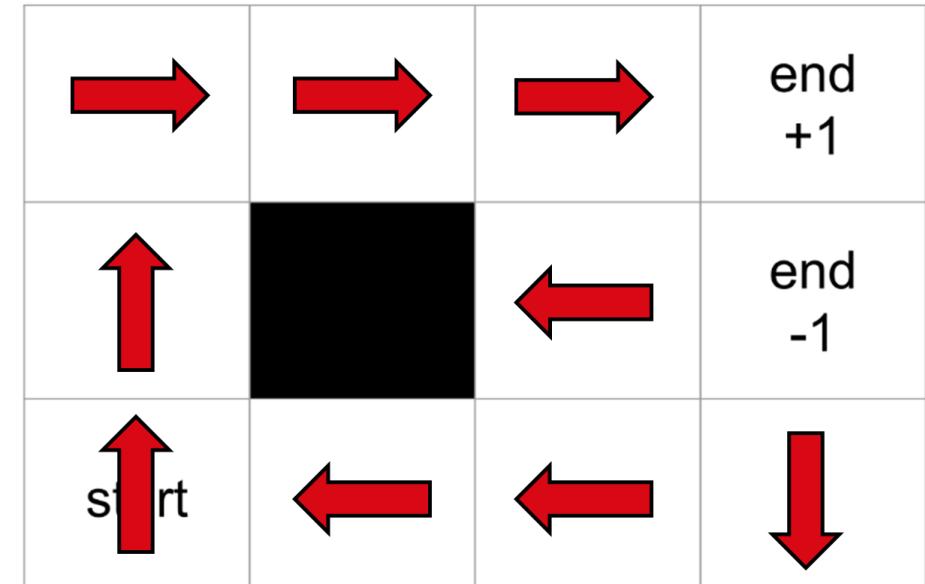


$$R(s) = -1.65$$

$$R(s) = -0.04$$

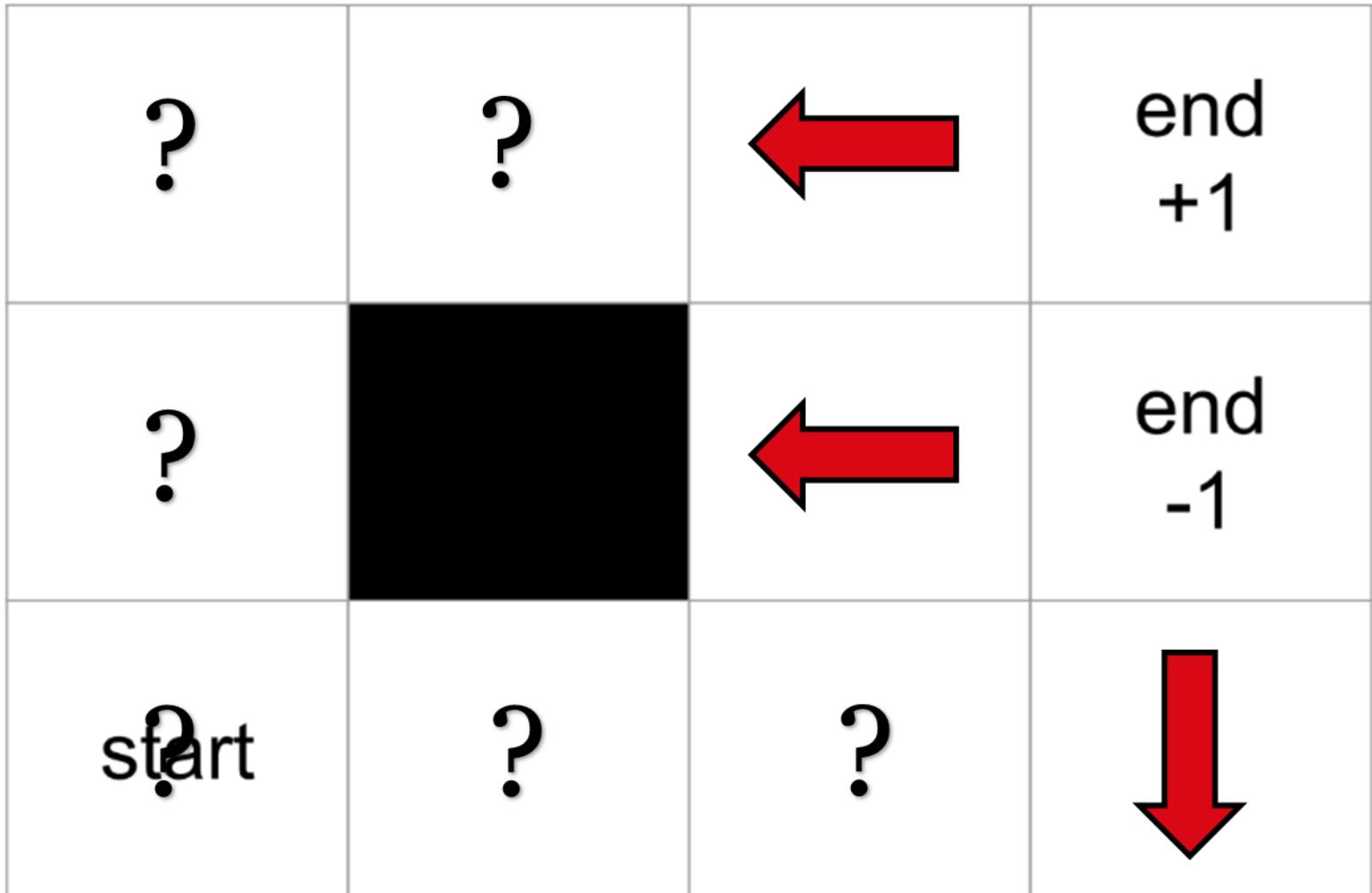


$$R(s) = -0.01$$



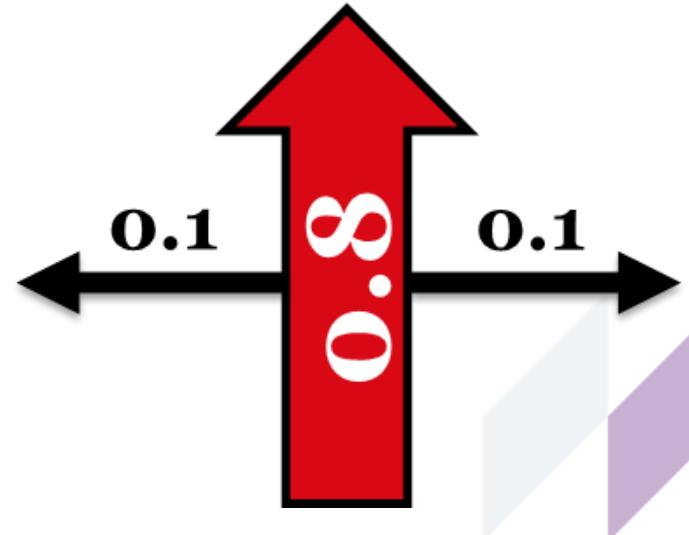
# MDP Example

$R(s) > 0$



# Markov Decision Process (MDP)

- We use alternative approaches to solve MDPs.
  - Value Iteration
  - Policy Iteration
  - Linear Programming
- Planning Under Uncertainty holds provided we have this complete model.
  - i.e. The transitions, reward functions and fully-observable states.
- When one or more are not available, this becomes a reinforcement learning problem (model free).
  - Need to interact with the environment to learn these features.



-0.04	-0.04	-0.04	end +1
-0.04		-0.04	end -1
start	-0.04	-0.04	-0.04

# Classical Planning Planning Under Uncertainty Markov Decision Process

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



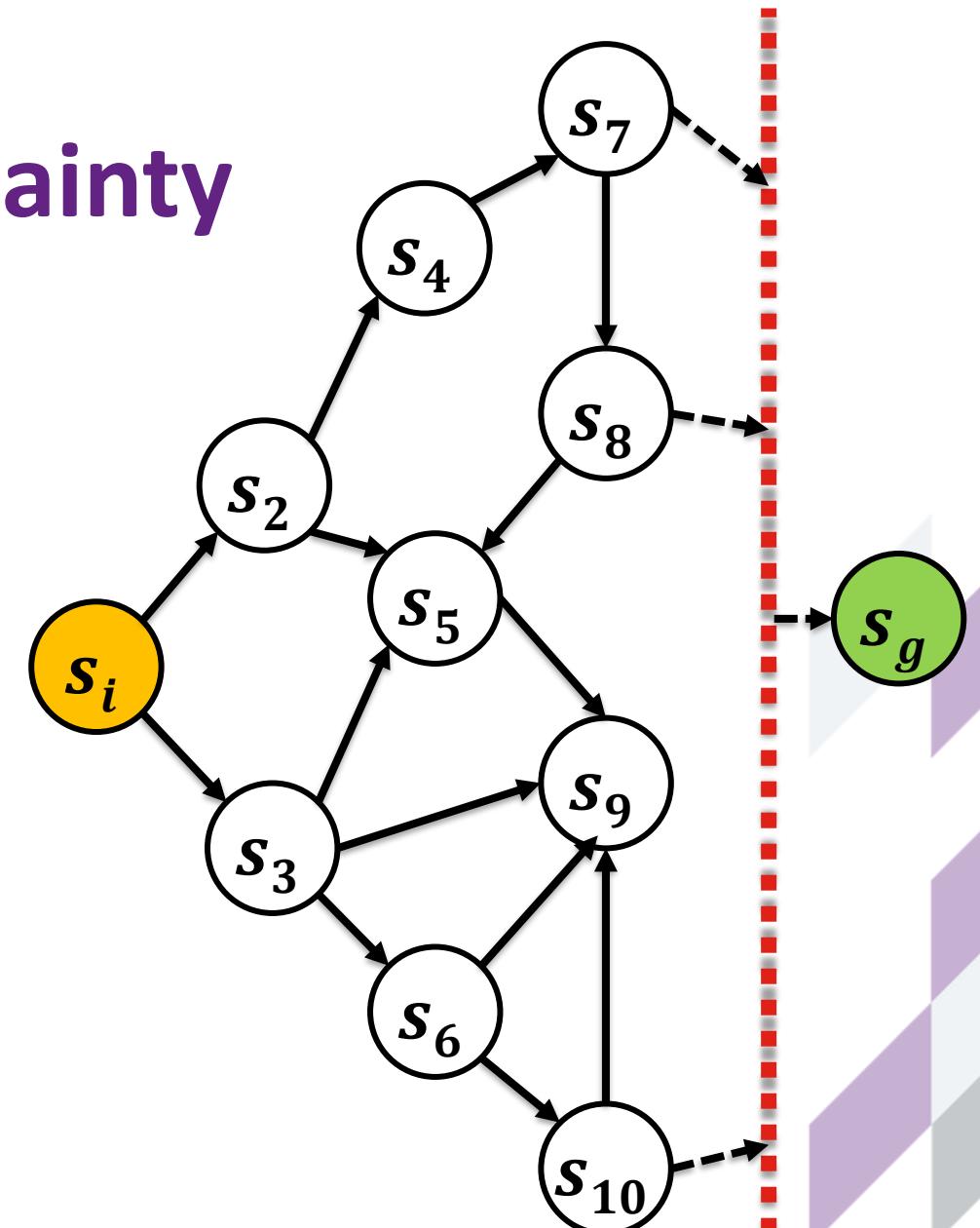
# Classical Planning Partially Observable Markov Decision Process

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



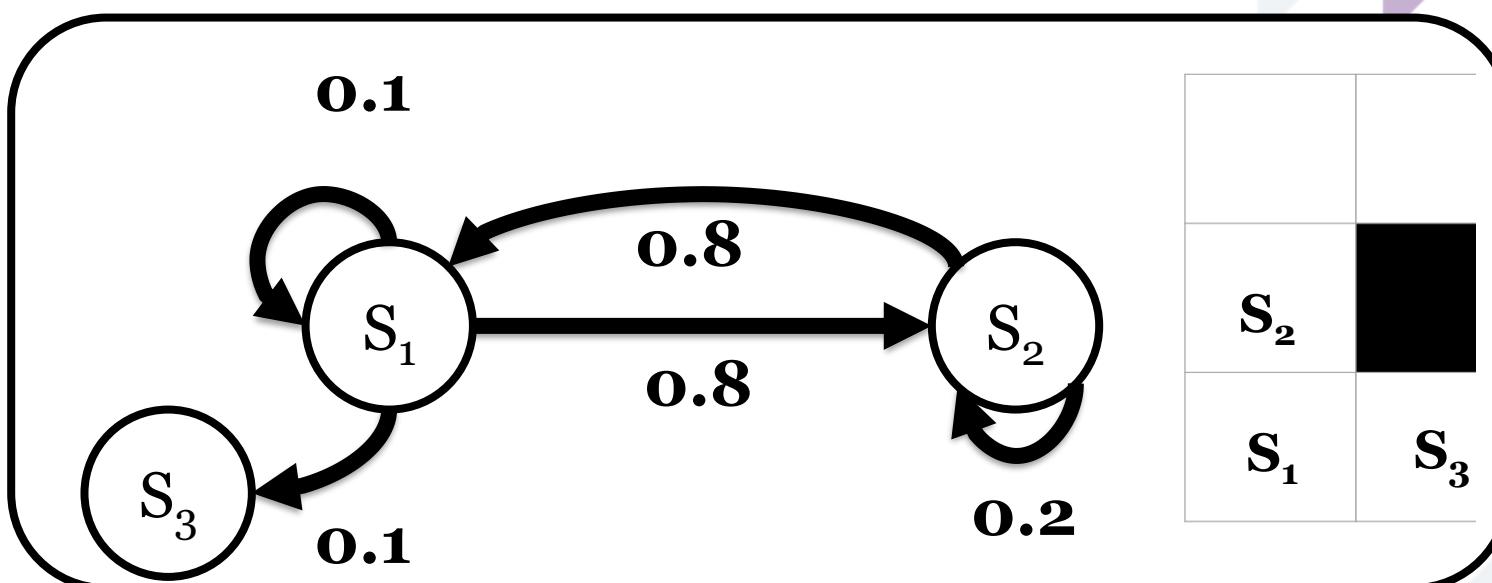
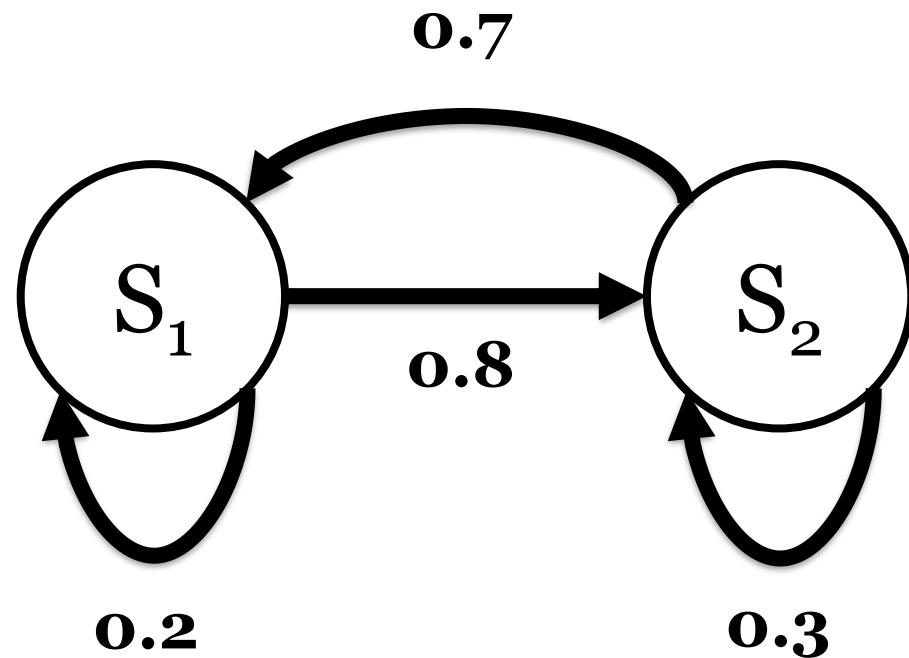
# Classical Planning = No Uncertainty

- We assume the state is known.
- We assume all states in the problem are known.
- We assume that we know all actions from the current state.
- We assume that a given action will always execute as intended.
- We assume that the successor states of action execution are also known.



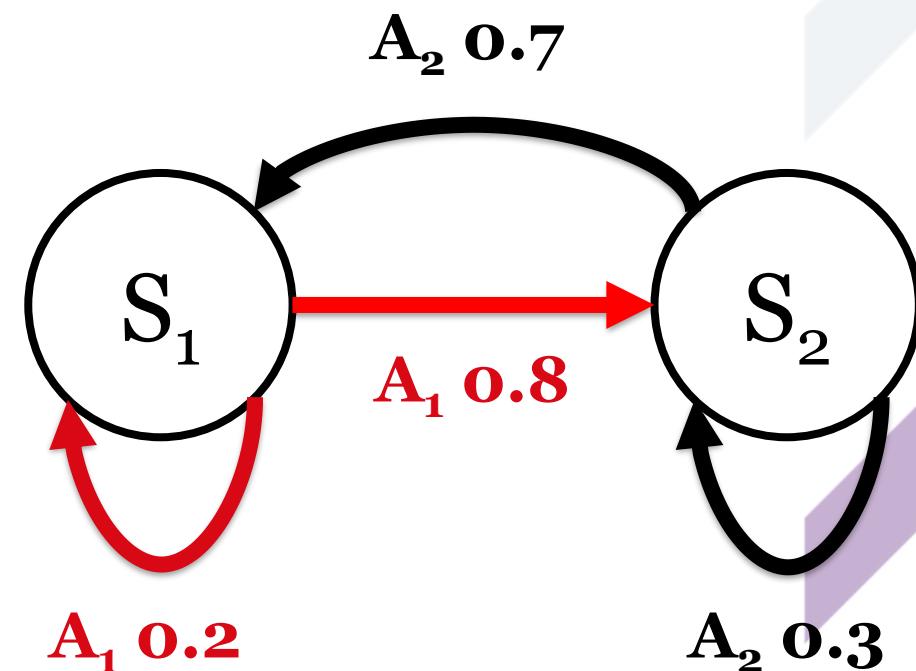
# Recap: Markov Chains

- Modelling probabilistic transitions between states.
- Next state is only determined by probabilistic transition between states.
- Does not factor the history of previous states.
  - Known as the Markov Property.



# Recap: Markov Decision Process (MDP)

- A sequential decision making problem for a fully observable but stochastic environment.
- Markov Chain holds, but now with probability of actions between states and rewards for action execution.
- Reliant on two principles:
  - Assume that Markov property holds for action transitions (which it should).
  - Probability distributions are stationary and don't change.



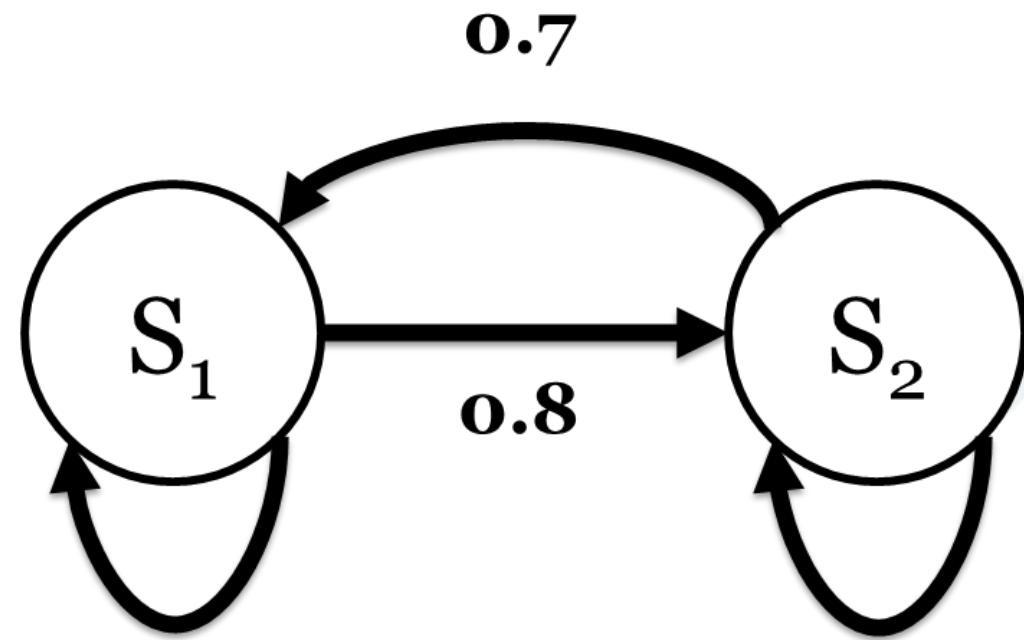
# Classical Planning = No Uncertainty

- We assume the state is known.
- We assume all states in the problem are known.
- We assume that we know all actions from the current state.
- We assume that a given action will always execute as intended.
- We assume that the successor states of action execution are also known.



# Hidden Markov Model

- Same as before...
- Modelling probabilistic transitions between states.
- Probabilistic transition between states.
- Rewards received at states.
- But, the Markov Chain is now hidden.
- States emit observations with a probability – which we can see.
- Based on the observations, we can then guess what state we're in.

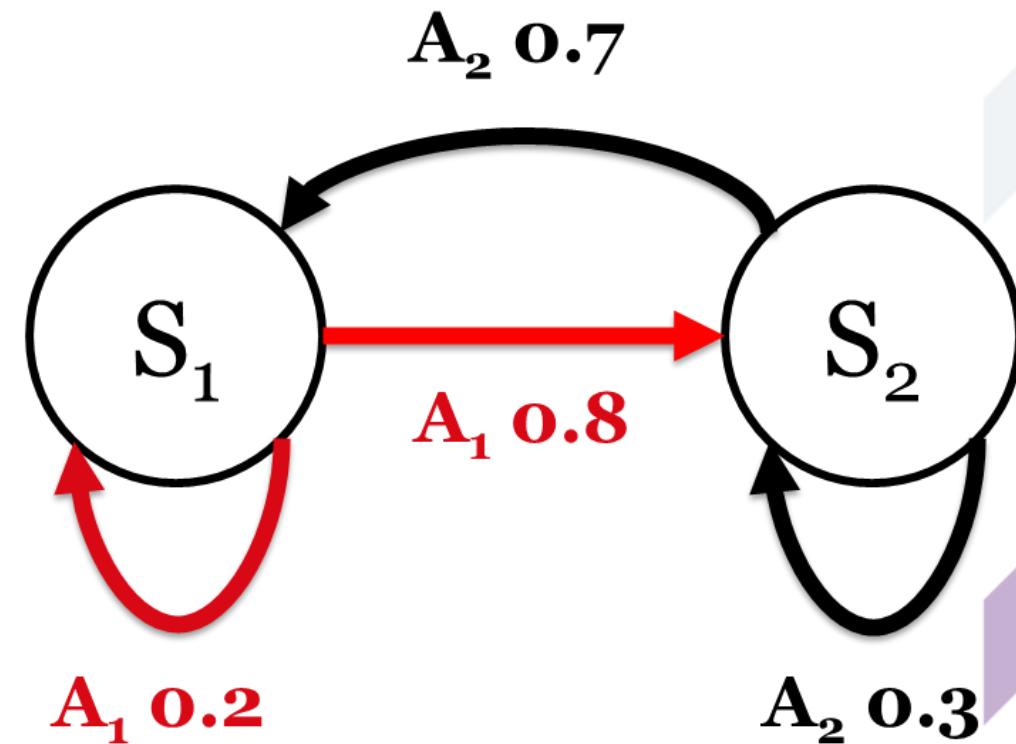


S<sub>1</sub> emits with  $p = 0.75$   
S<sub>2</sub> emits with  $p = 0.75$

# Partially Observable Markov Decision Process (POMDP)

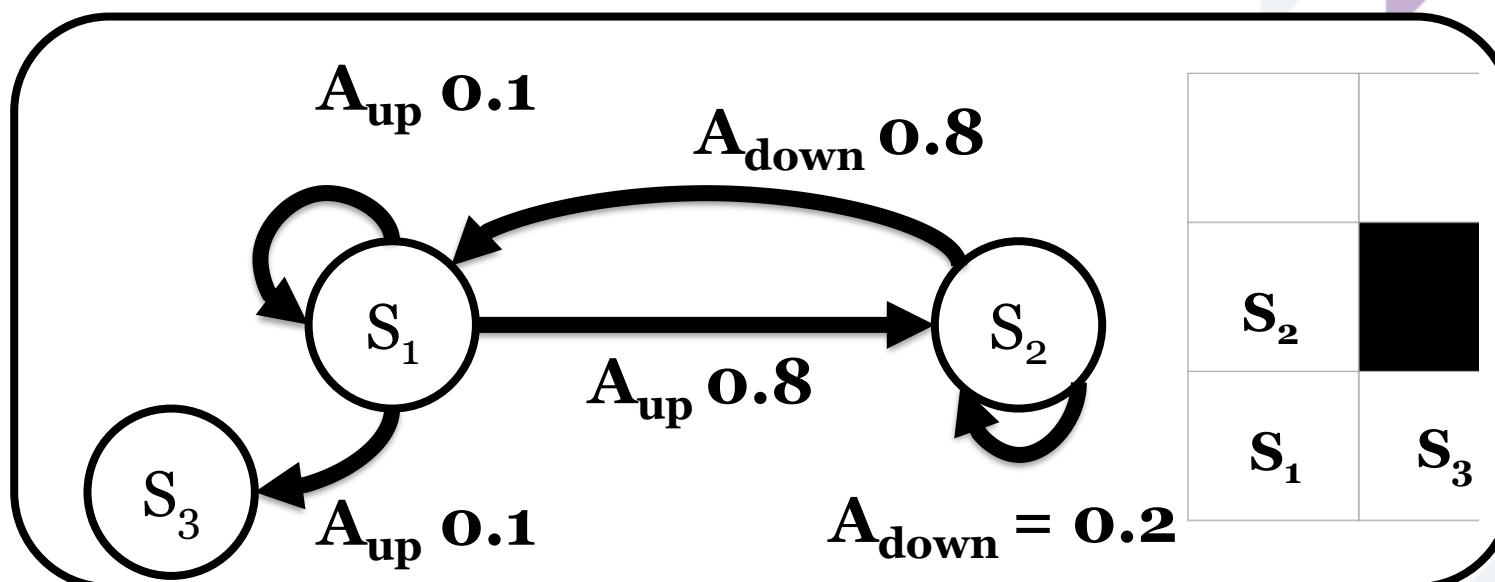
- Modelling probabilistic transitions between states.
- Next state is only determined by probabilistic transition between states.
- Does not factor the history of previous states.
- But is unsure of what state we are in, reliant on observations to build history of states visited.

S<sub>1</sub> emits with p = 0.75  
S<sub>2</sub> emits with p = 0.75



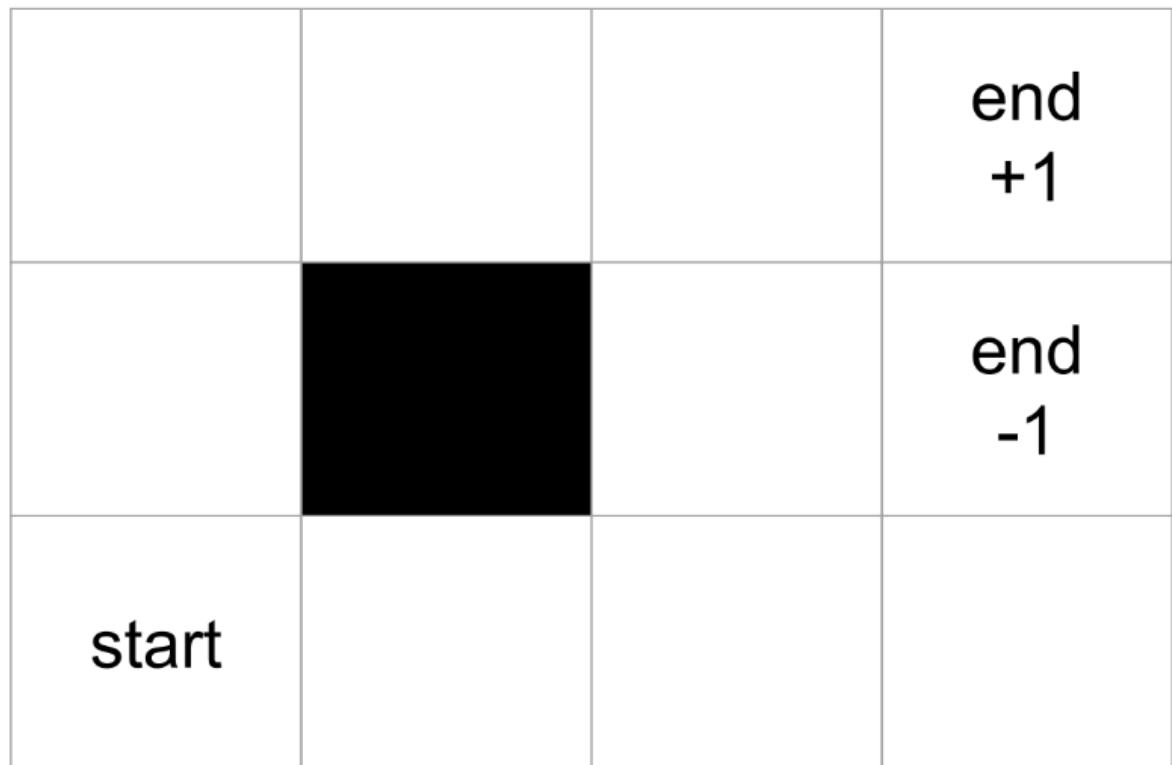
# Partially Observable Markov Decision Process (POMDP)

- POMDP is comprised of...
- States ( $S$ ), Actions ( $A$ )
- Transition probabilities:  $P(s'|s, a)$
- Reward accrued  $R(s)$
- A belief vector  $b$  - the probability distribution of which state we are in.
- A sensor model  $O(o | s)$  which indicates based probability we would have seen an observation in a given state.



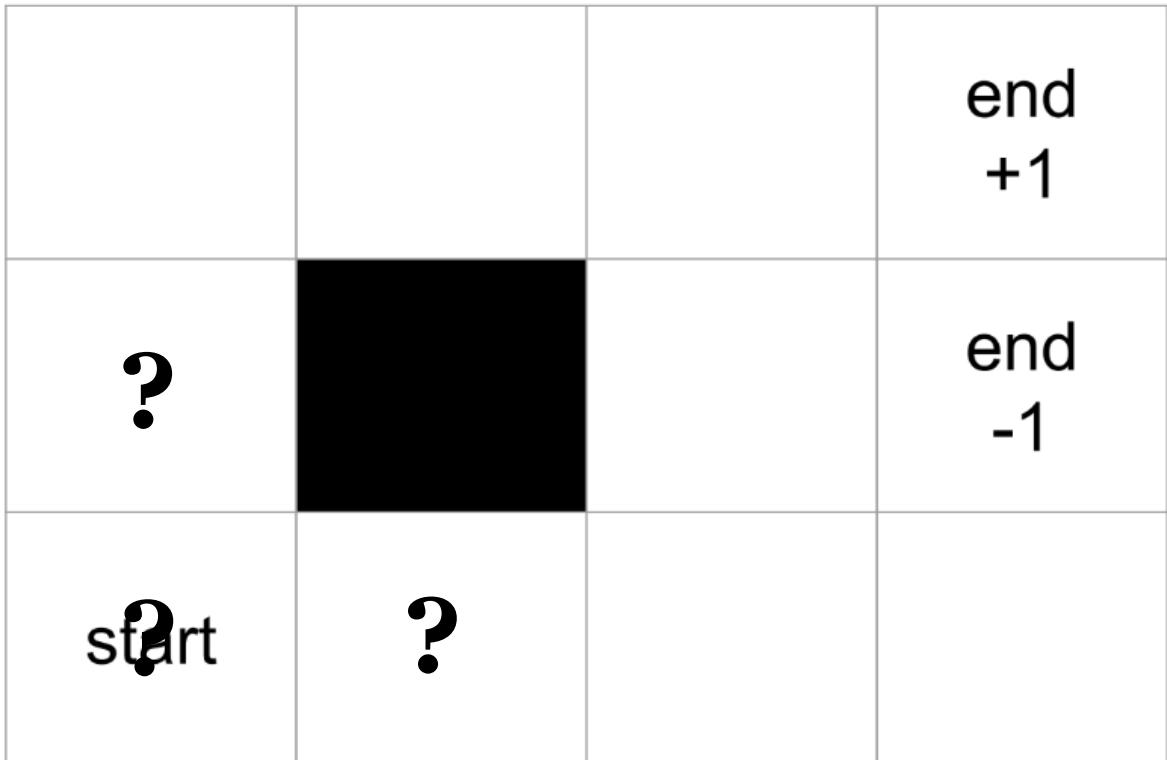
# Partially Observable Markov Decision Process (POMDP)

- So hang on... if we cannot see the Markov Model, how can we work with this?
- We know it exists, we know what it looks like, we just can't say what the current state is.
- We rely on the observations to make assumptions of which state we are in.
- We then take actions and rely on the subsequent observations to build a belief vector.



# Belief Vector

- Belief vector is a collection of probabilities of what states we believe the agent is in at this point in time.
- So for example, we may start in (1,1), but we need observations tell us we could be in this state.
- E.g. – we're 80% confident we're in (1,1) and 10% confident we're in (1,2) and (2,1)
- i.e.  $O(o | s_{11}) = 0.8$



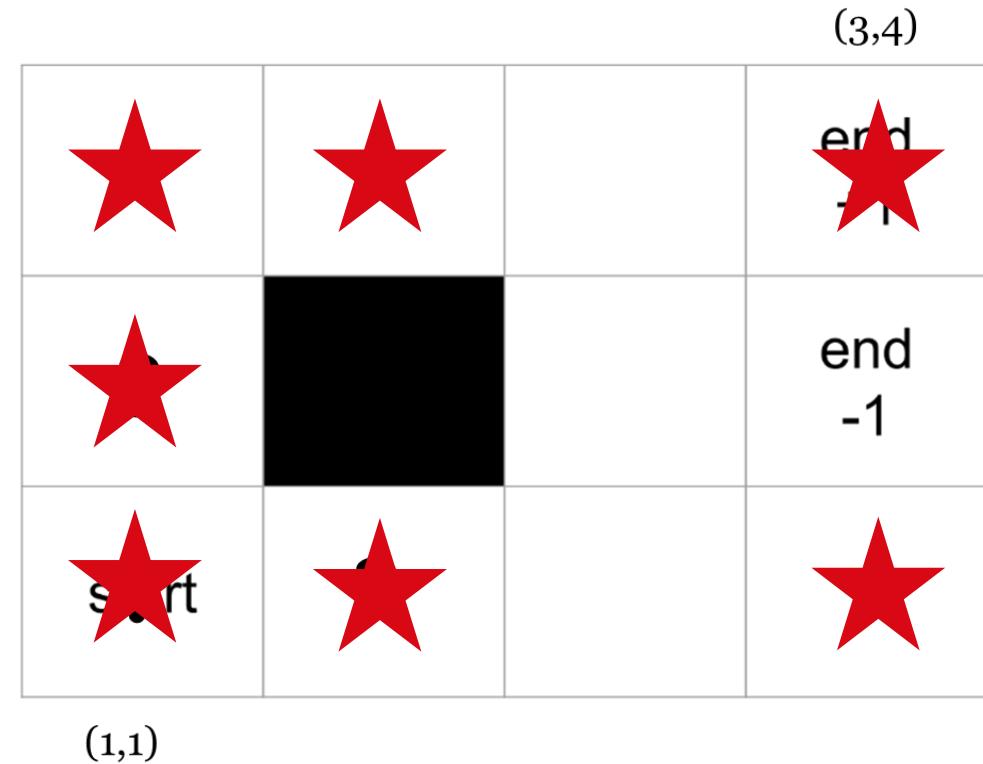
(1,1)

$b(s_{init}) = \{0.8, 0.1, 0, 0, 0.1, 0, 0, 0, 0, 0\}$   
given observation, ignoring the blank space in (2,2)

$$b(s_{init}) = \left\{\frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0\right\}$$

# Calculating Belief from Observation

- Now consider the observation of the robot: it can see the number of visible walls at the current cell.
  - Can't tell their orientation, just their number.
  - Hence this observation  $o$  is only ever 1 or 2.
  - Observations can also have probabilities of occurring.
- Hence if you then apply the observation to the sensor model, it will tell us which states we could be in right now.

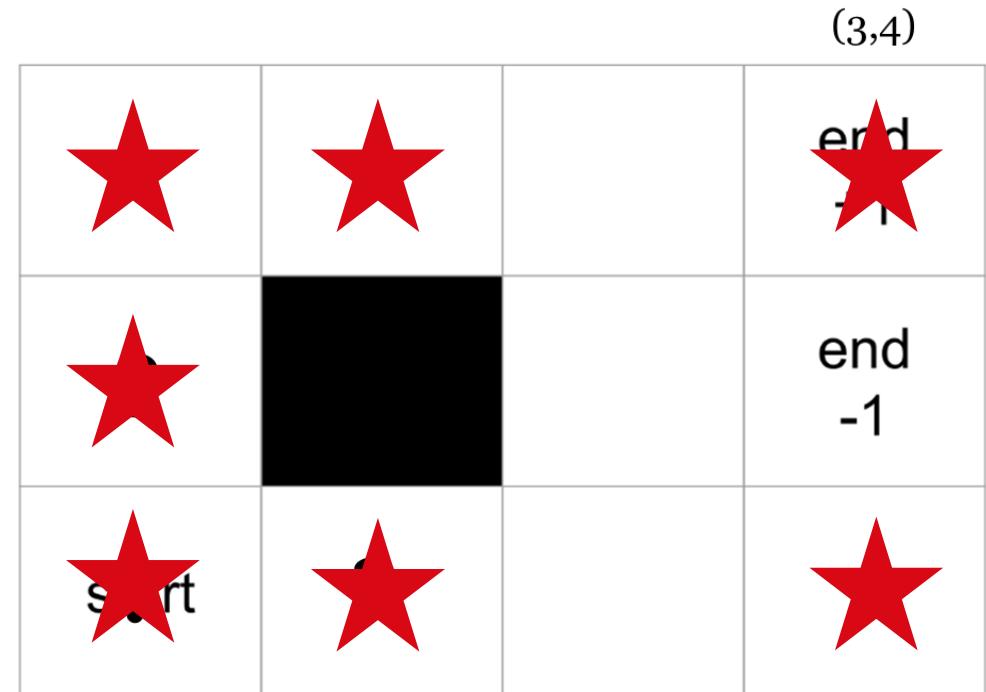


$$O(o = 2 | s) = \frac{1}{7} \quad \forall s \in \{s_{11}, s_{21}, s_{41}, s_{12}, s_{31}, s_{32}, s_{34}\}$$

# Calculating Belief from Observation

- The belief state needs to be updated based on actions taken in the world.
- But given we're unsure of the state we're in and whether action execution is successful, we have to factor that into the belief state update.
- Update upon taking action  $a$  and observing  $o$ :

$$\mathbf{b}'(s') = \alpha O(o|s') \sum_{s \in S} P(s'|s, a) \mathbf{b}(s)$$

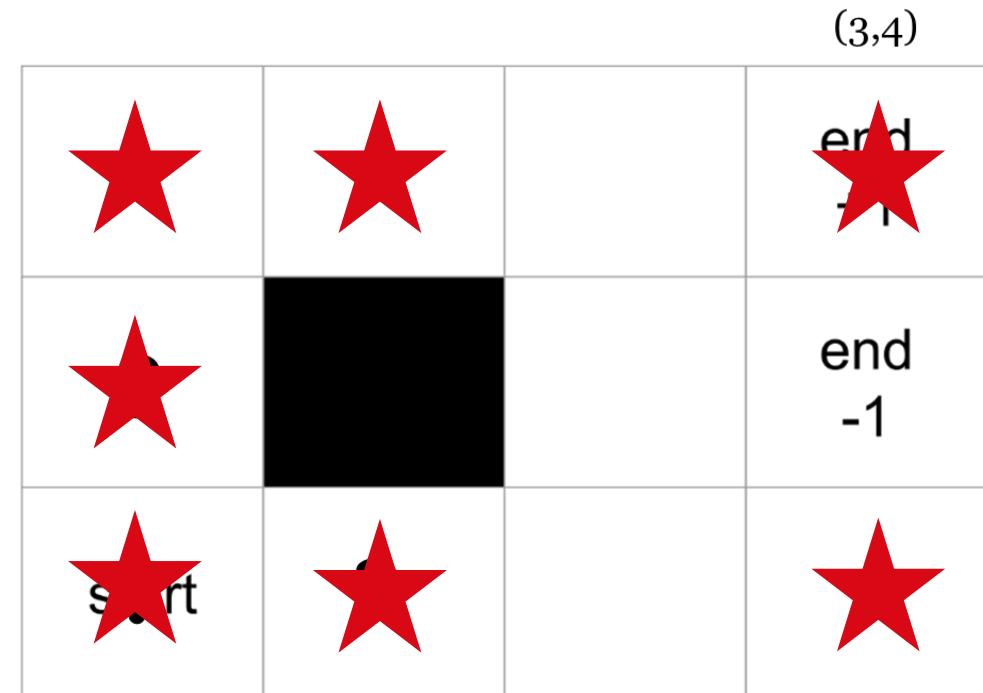


(1,1)

# Calculating Belief from Observation

- The belief state needs to be updated based on actions taken in the world.
- But given we're unsure of the state we're in and whether action execution is successful, we have to factor that into the belief state update.
- Update upon taking action  $a$  and observing  $o$ :

$$\mathbf{b}'(s') = \alpha \mathbf{O}(o| s') \sum_{s \in S} P(s'| s, a) \mathbf{b}(s)$$

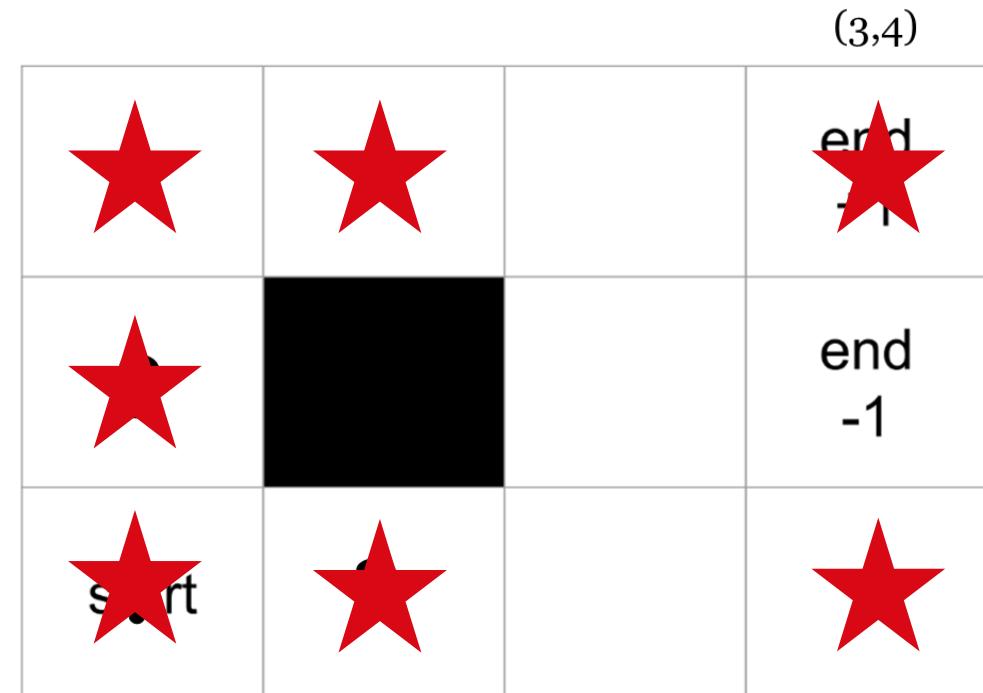


(1,1)

# Calculating Belief from Observation

- The belief state needs to be updated based on actions taken in the world.
- But given we're unsure of the state we're in and whether action execution is successful, we have to factor that into the belief state update.
- Update upon taking action  $a$  and observing  $o$ :

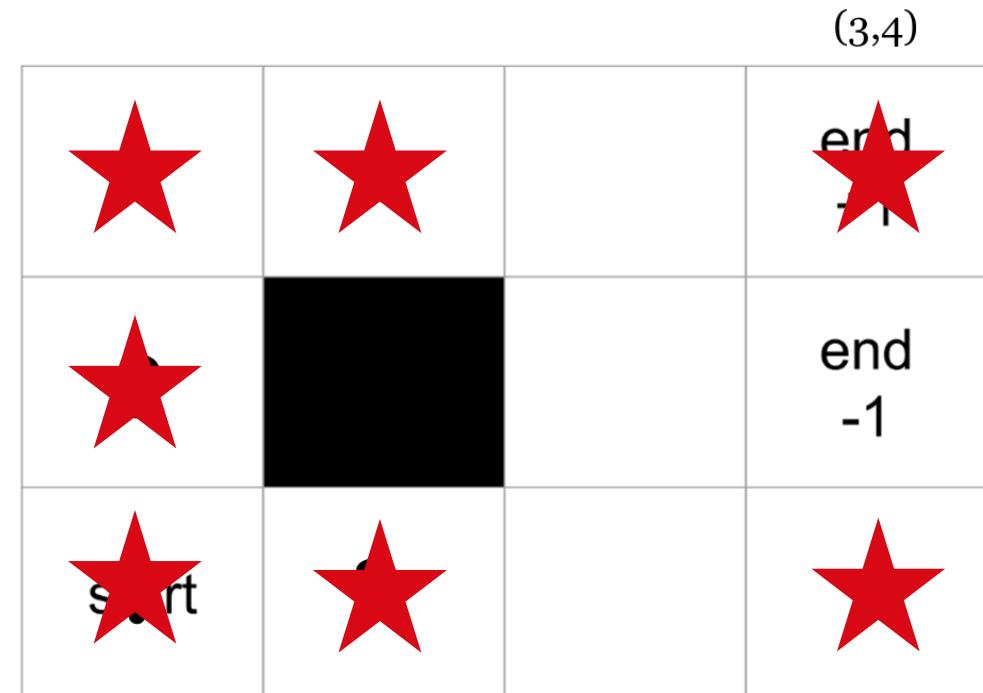
$$b'(s') = \alpha O(o|s') \sum_{s \in S} P(s'|s, a) b(s)$$



# Calculating Belief from Observation

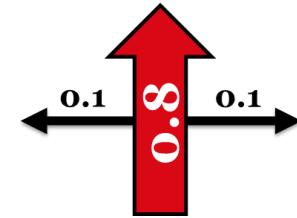
- The belief state needs to be updated based on actions taken in the world.
- But given we're unsure of the state we're in and whether action execution is successful, we have to factor that into the belief state update.
- Update upon taking action  $a$  and observing  $o$ :

$$b'(s') = \alpha O(o|s') \sum_{s \in S} P(s'|s, a) b(s)$$



(1,1)

# Calculating Belief from Observation



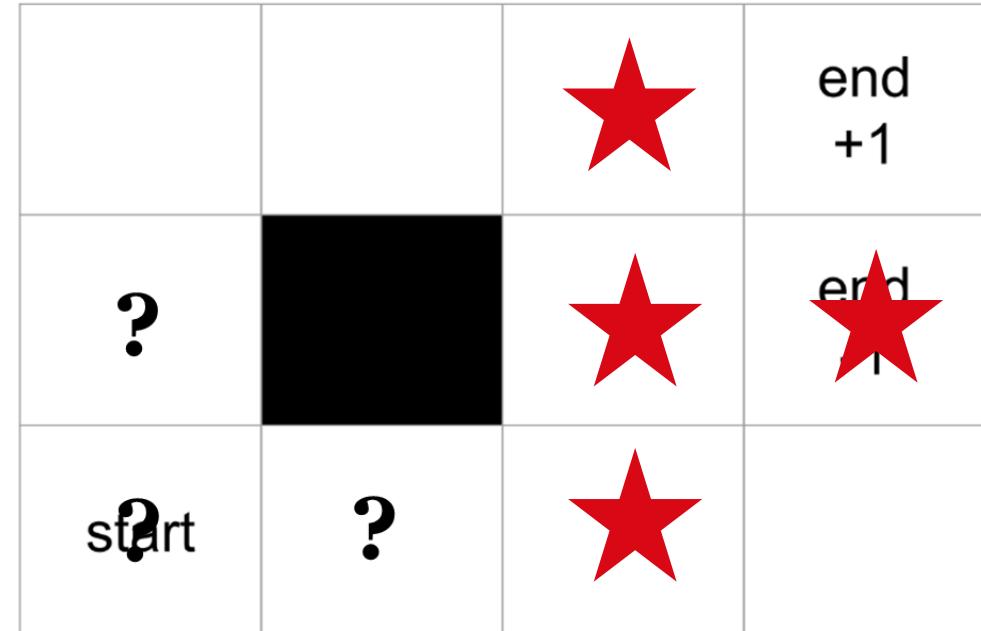
(3,4)

- E.g. Assuming we have no observations and execute the Up action, and then observe one wall, what's the new belief state?

- Assuming  $\alpha = 1$

$$b(s_{init}) = \left\{ \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0 \right\}$$

$$b(s') = \{0, 0, 0.02, 0, 0, 0.1, 0.1, 0, 0, 0.1, 0\}$$



- Problem that the normalisation constant needs to be more useful.

$$b'(s') = \alpha O(o|s', a) \sum_{s \in S} P(s'|s, a) b(s)$$

# Calculating Belief from Observation

- E.g. Assuming we have no observations and execute the Up action, and then observe one wall, what's the new belief state

$$b'(s') = \frac{O(o|s', a) \sum_{s \in S} P(s'|s, a) b(s)}{\sum_{s' \in S} O(o|s', a) \sum_{s \in S} P(s'|s, a) b(s)}$$

$$b(s_{init}) = \left\{ \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0 \right\}$$

$$b(s') = \{0, 0, 0.06, 0, 0, 0.31, 0.31, 0, 0, 0.31, 0\}$$



# Summary

Markov Models	Uncontrolled Transitions	Actions
States Fully Observable	Markov Model	Markov Decision Process
States Partially Observable	Hidden Markov Model	Partially Observable Markov Decision Process

# Classical Planning Partially Observable Markov Decision Processes

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Planning with Preferences

# Planning with Preferences

## Given:

- An initial state: a set of propositions and assignments to numeric variables,  
e.g. (at rover waypoint1) (= (energy rover) 10).
- A goal: a desired set of propositions/assignments,  
e.g. (at rover waypoint4) (have-soil-sample waypoint3).
- A set of actions each with:
  - Preconditions on execution;
  - Effects that describe how the world changes upon their execution.

```
(:action navigate
:parameters
(?r - rover ?y - waypoint ?z - waypoint)
:precondition (and
                (available ?r)
                (at ?r ?y)
                (visible ?y ?z)
                (>= (energy ?r) 8))
:effect (and
          (decrease (energy ?r) 8)
          (not (at ?x ?y))
          (at ?x ?z)))
```

## Find:

- A sequence of actions that when applied in the initial state leads to a state that satisfies the goal condition.

# Planning with Preferences

- Find: **A** sequence of actions: any plan will do?
  - Even the shortest might not be necessarily the best.
  - We might care about *how* the goal is achieved.
- What if we can't reach the goal:
  - Report 'No Plan Exists'.
  - Search Indefinitely.
  - Maybe we could satisfy some of it?

# Planning with Preferences

- Simple Preferences (*soft goals and preconditions*):
  - $(p0 \text{ (at end (at rover waypoint3))})$
- *Trajectory preferences (Conditions on the plan)*:
  - $(p1 \text{ (always } >= \text{ (energy rover) } 2\text{)})$
  - $(p2 \text{ (sometime (at driver costa-coffee))})$
  - $(p3 \text{ (at-most-once (at truck Birmingham))})$
  - $(p4 \text{ (sometime-after (at Birmingham) (at Glasgow))})$
  - $(p5 \text{ (sometime-before (at Birmingham) (had-lunch))})$
- *Temporal Preferences (not covered)*.
- *Metric Function*:
  - $(\text{minimize } (+ (\text{fuel-used}) (*2 (\text{is-violated } p0)) (*5 (\text{is-violated } p1))))$

# Planners Handling Preferences

- Simple Preferences: *YochanPS*, *Keyder & Geffner translation*.
- Simple and Trajectory Preferences: *HPlan-P* (*non-numeric*),  
*LPRPG-P* (*propositional and numeric*).
- Simple, Trajectory and Temporal Preferences: *Mips-XXL*  
(*numeric*), *OPTIC*.
- Competition Domains: *SGPlan*.

# Modelling Preferences in PDDL 3.0

# Goals for today

- Get to work
- If I do that, get a coffee on the way
- If I get a coffee, go to the loo after
- Play the piano



# Goals for today

- Get to work
- If I do that, get a coffee on the way
- If I get a coffee, go to the loo after
- Play the piano



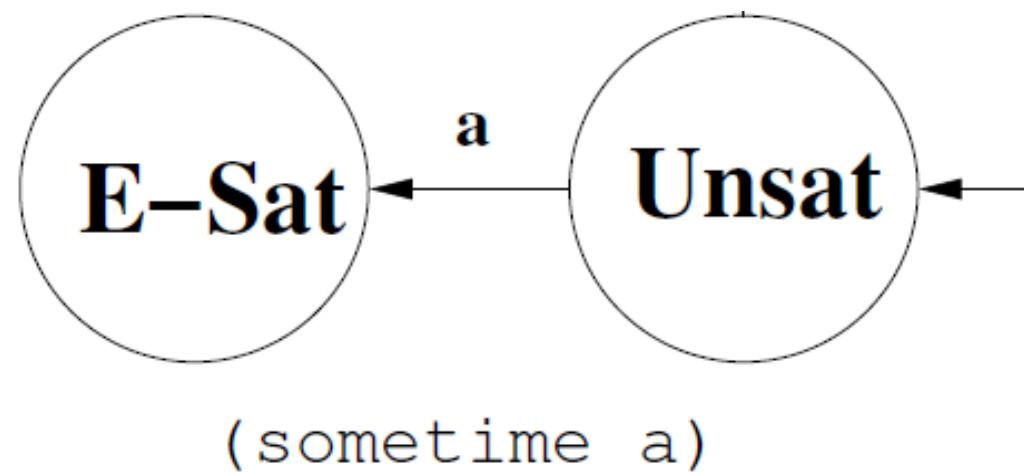
# Goals for \*the end\* of today?

```
(:goals (and  
        (at amanda work)  
      )  
)
```



# Work/life balance

- (sometime (at amanda work))
- (sometime (at amanda piano))



# Selling out...

- (preference p0 (sometime (at amanda work)))
- (preference p1 (sometime (at amanda piano)))
  
- $\text{cost}(p0) = 100$
- $\text{cost}(p1) = 5$

# With goal(s)

```
(:goal (and  
       (at amanda mybed)  
     ))
```

```
(:constraints (and  
              (preference p0 (sometime (at amanda work)))  
              (preference p1 (sometime (at amanda piano))))  
            ))
```

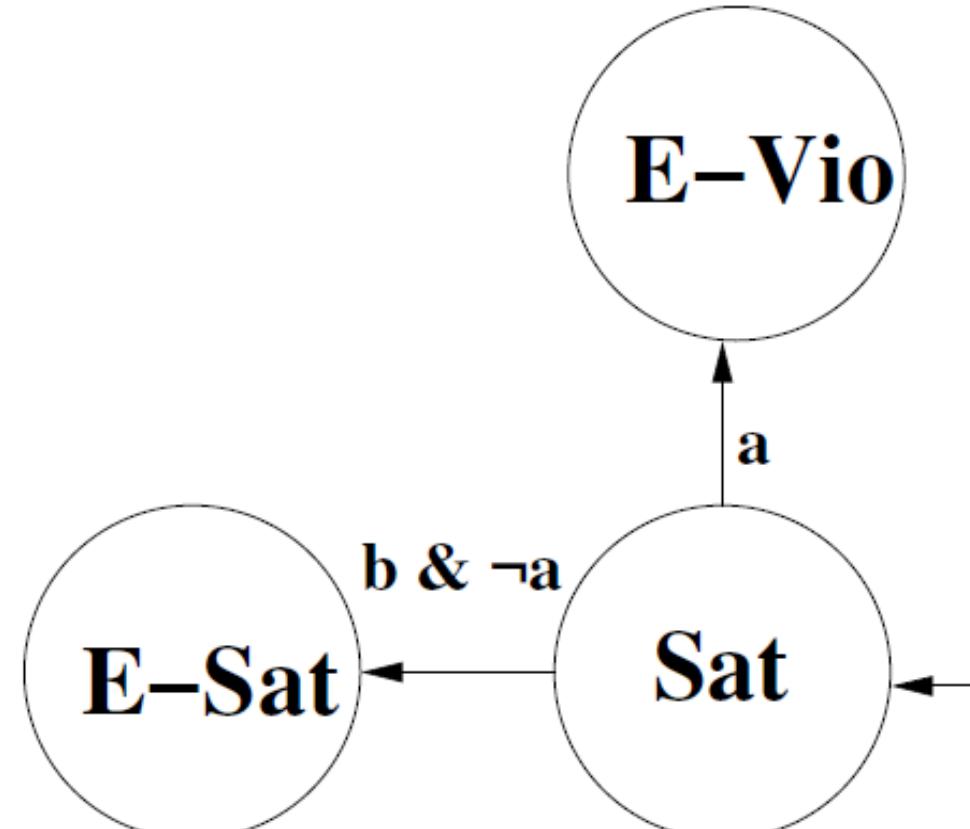
If I do that, get a coffee on the way

(sometime (at amanda coffeeshop)) ?

(sometime-before (at amanda work)  
(at amanda coffeeshop))

$a = (\text{at amanda work})$

$b = (\text{at amanda coffeeshop})$



(sometime-before  $a \ b$ )

If I get a coffee, go to the loo

- (sometime (at amanda loo)) ???

If I get a coffee, go to the loo

(sometime-before (at amanda work)

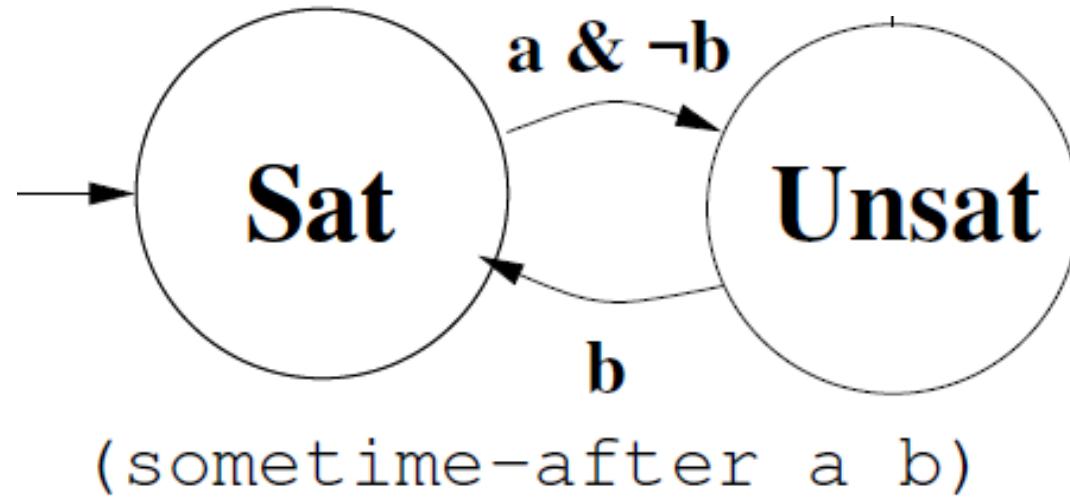
(at amanda coffeeshop))

(sometime-after (at amanda coffeeshop)

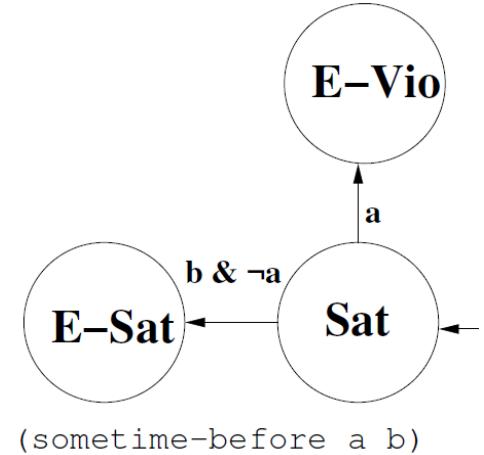
(at amanda loo))

$a = (\text{at amanda coffeeshop})$

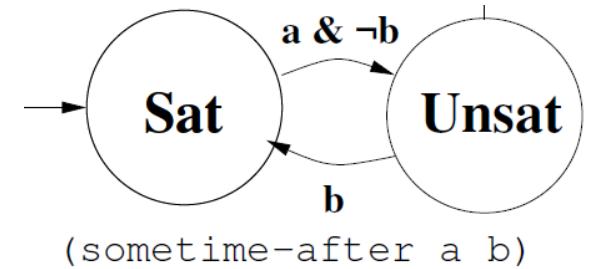
$b = (\text{at amanda loo})$



# Sometime-Before vs Sometime-After



- These are not inverses:
    - If I go to work then I have to have coffee first;
    - If I drink coffee I have to go to work afterwards.



# Watch Out!

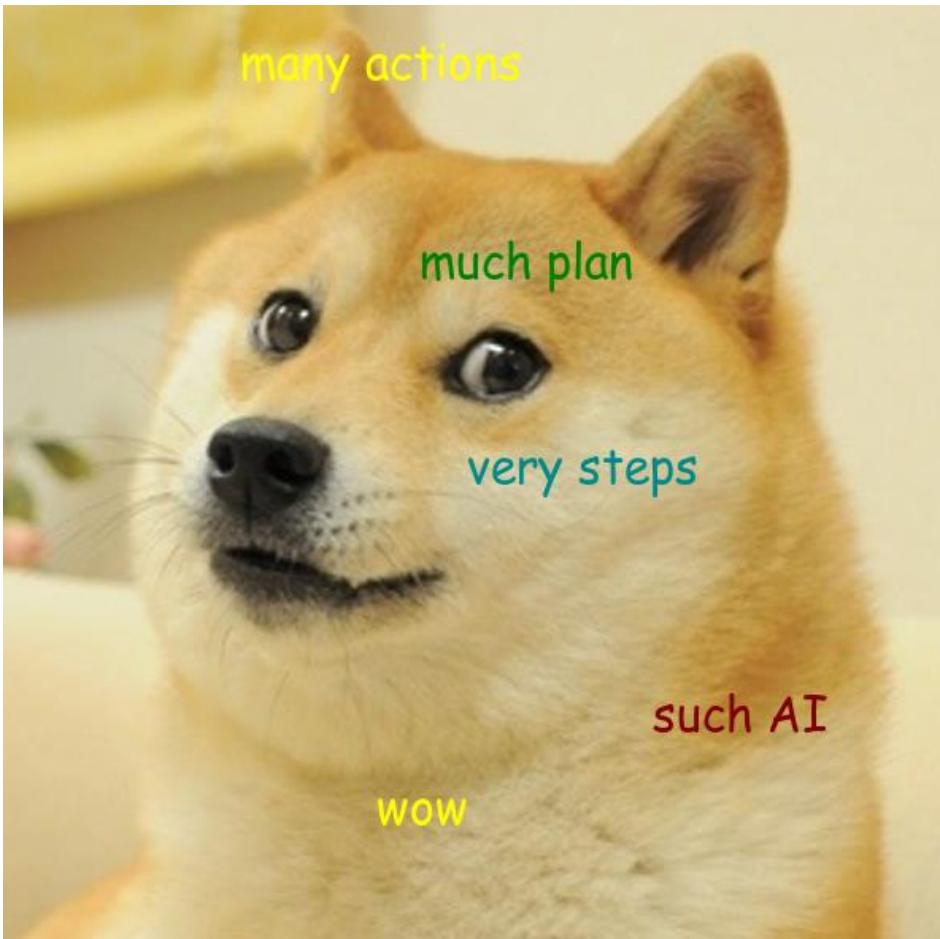
# One last sort of preference

```
(:goal (and  
       (at amanda mybed)  
       (preference p4 (switched-off phone)))
```

# Solving Planning Problems With Soft Goals

# Goal preferences must be easy?

- It's almost a goal, c'mon...



# Compiling goal preferences

- Add a fact **normal-mode** as a precondition to every action
- Add an action **end** that deletes **normal-mode** and adds **end-mode**
- Two actions for each goal preference  $p_i$ 
  - **collect( $p_i$ )** – has  $p_i$  as its precondition, adds  $p'_i$
  - **forgo( $p_i$ )** – has  $(\text{not } (p_i))$  as its precondition, adds  $p'_i$ , and **increases plan cost by  $\text{cost}(p_i)$**
- Set goal to: **(and ( $p'_0$ ) ( $p'_1$ ) ( $p'_2$ ) ...)**

# RPG Heuristic?

- What's a relaxed plan for a problem with four compiled-away goal preferences?

# RPG Heuristic?

- What's a relaxed plan for a problem with four compiled-away goal preferences?
- (end)
- (forgo p0) (forgo p1) (forgo p2) (forgo p3)
- $h(S) = 5$  ('distance to go')
- Regardless of how close it is to meeting a goal

# Keyder & Geffner, JAIR, 2009

SOFT GOALS CAN BE COMPILED AWAY

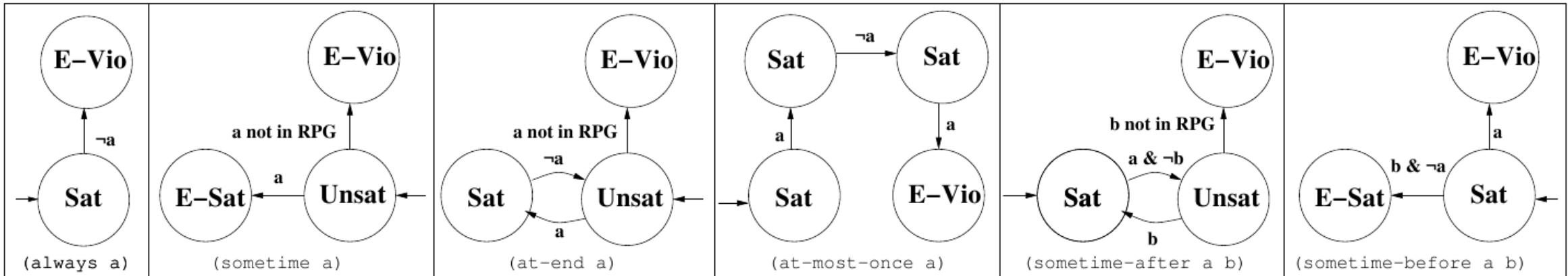
Domain	Net-benefit satisfying planners			Cost satisfying planners Lama
	SGPlan	YochanPS	Mips-XXL	
elevators (30)	0	0	8	<b>23</b>
openstacks (30)	2	0	2	<b>28</b>
pegsool (30)	0	5	23	<b>29</b>
rovers (20)	8	2	1	<b>17</b>
total	10	7	34	<b>97</b>

Table 2: Coverage and quality for satisfying planners: The entries indicate the number of problems for which the planner generated the best quality plan.

# LPRPG-P: Reasoning with Preferences

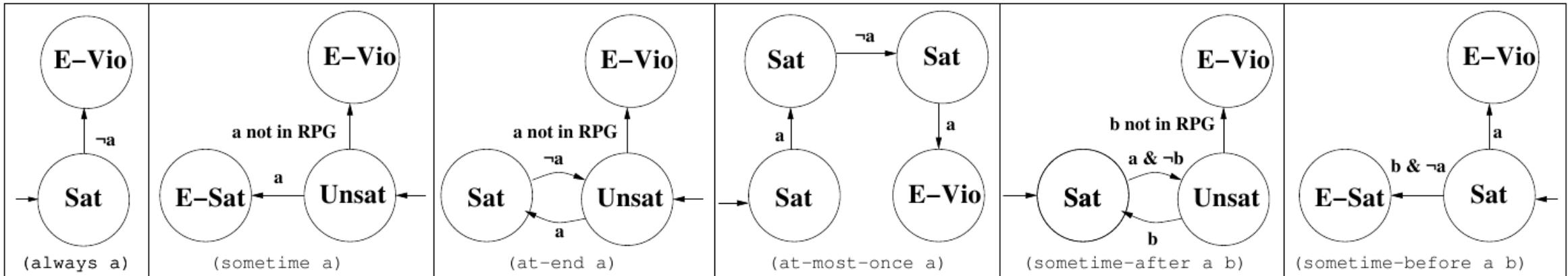
# Tracking Preference Satisfaction in Search

- Preferences have corresponding automata:
  - Automated translation from LTL into automata.
- At each state maintain an automaton corresponding to each preference.
  - Each starts in initial position, update according to initial state.
- Each time an action is applied (state expanded):
  - Update the automaton if condition from current position fires.



# Search in LPRPG-P

- Preference violation cost of a state PVC(S):
  - Sum of violation costs of all automata that are in E-Vio;
- Search until a solution is found:
  - i.e. a plan that satisfies all the hard goals.
- Continue searching once a goal is found:
  - But prune all states where  $\text{PVC}(S) > \text{cost of best solution found so far}$ .

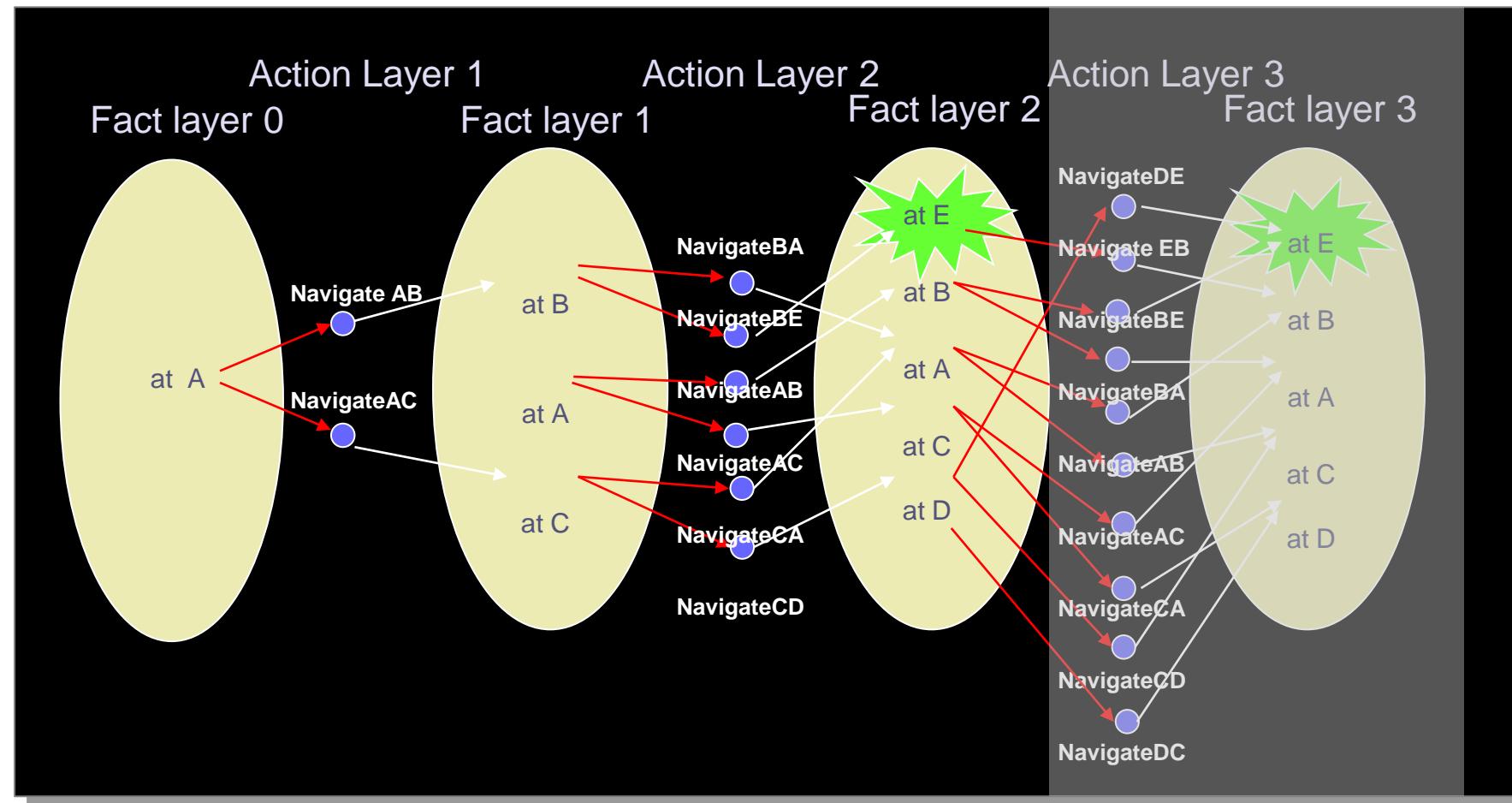
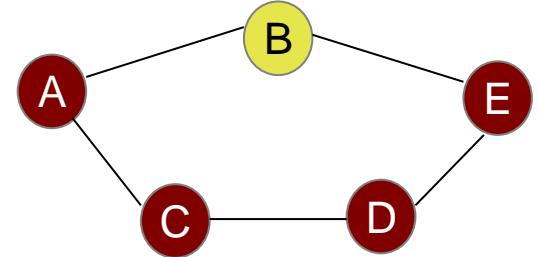


# The Challenge: Heuristic Guidance

- Planning heuristics generally focus on finding a path to the goal in a *relaxed* problem.
  - Relaxed problem ignores delete (negative) effects of actions.
- Typically these heuristics look for a short path to the goal.
- If we have a problem with no hard goals and only preferences/soft goals the RPG heuristic value is zero for every state!
- We need guidance about how to satisfy preferences and how difficult it will be to do so.

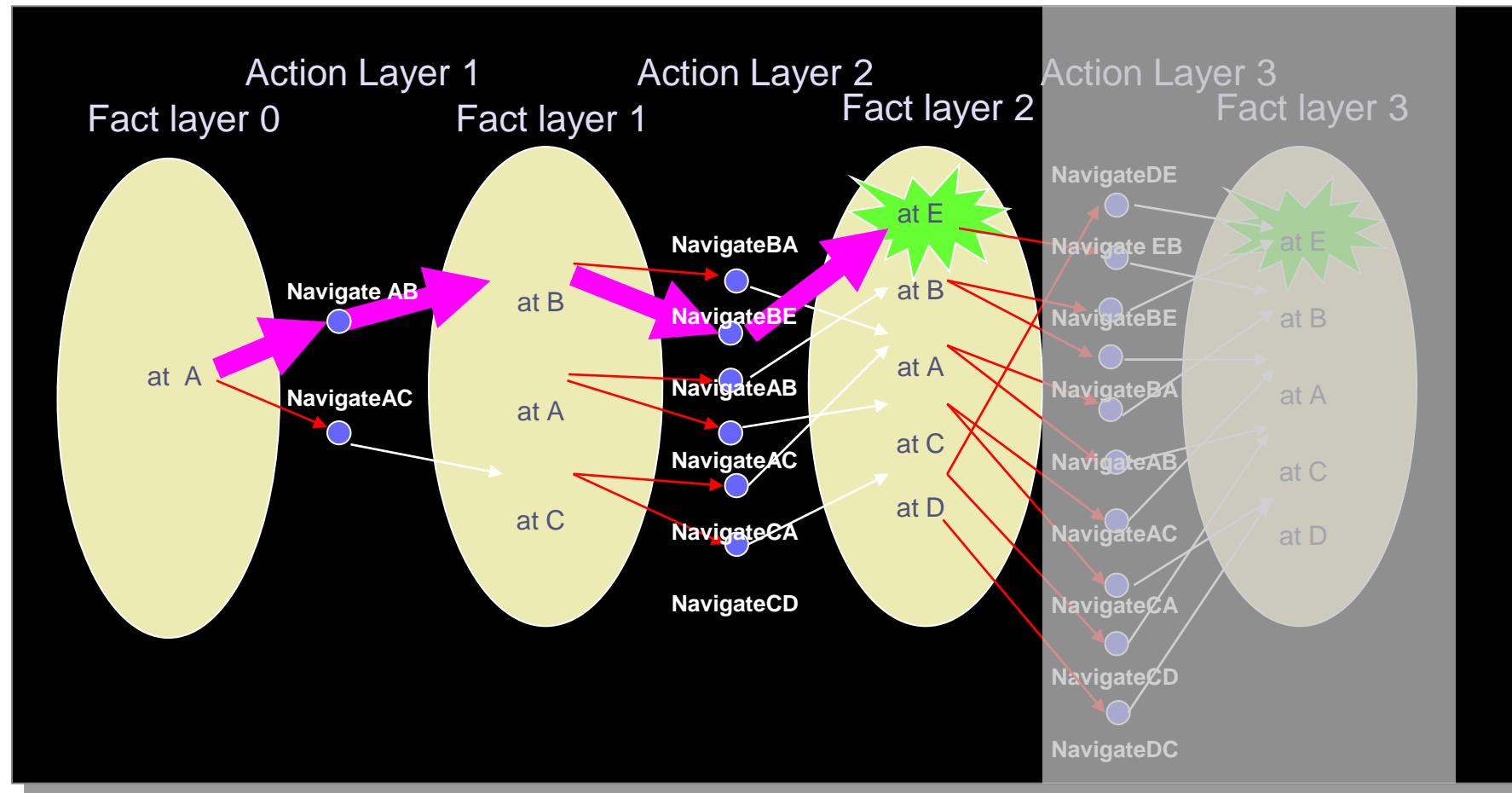
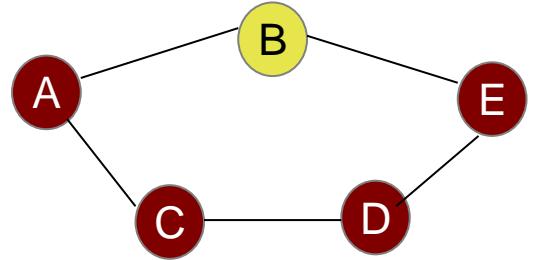
# Relaxed Planning Graph (RPG) Example

- Initial state: at A.
- Goal state: at E.



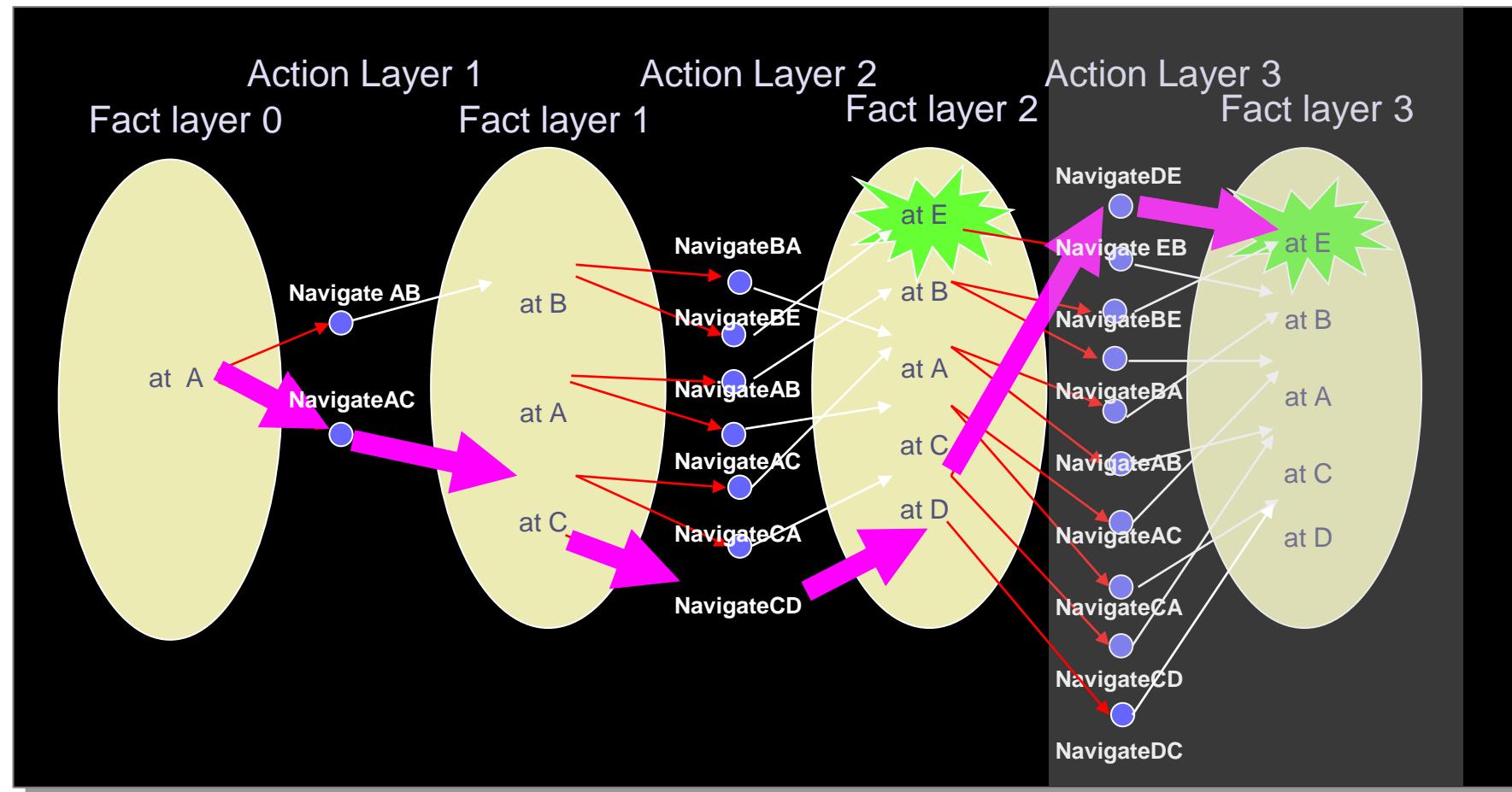
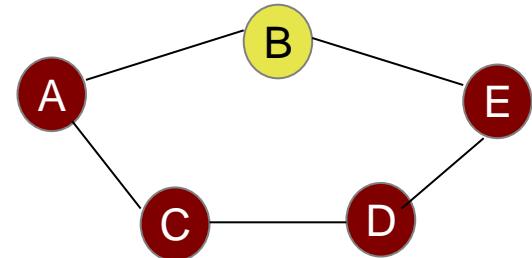
# Relaxed Plan Extraction

- Select a Goal;
- Select achiever;
- Add Preconditions to Goals.



# Relaxed Plan Extraction

- Select a Goal;
  - Select achiever;
  - Add Preconditions to Goals.
- (preference p1 (always (not (at B))))



# What do we Need?

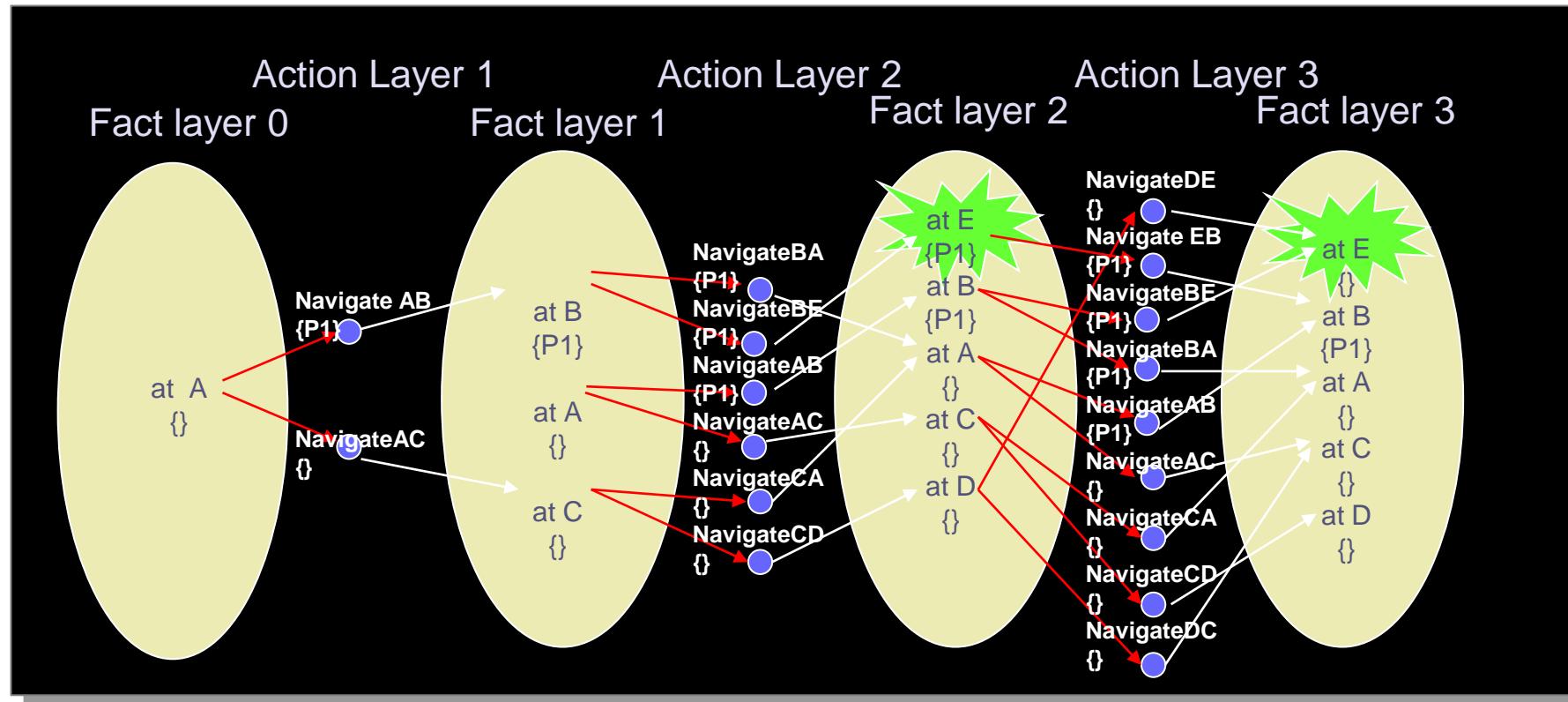
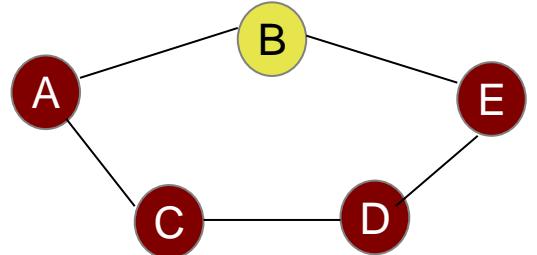
- Termination Criterion.
  - Stop building graph when goals appear insufficient
- A mechanism for selecting the right achiever.
  - Earliest not always best;
  - Arbitrary could miss something good.
- Effectively we need to track knowledge about preferences whilst building the RPG.

# A Preference Aware RPG

- At each fact layer we maintain a set of preferences for each fact.
  - These are the preferences that are violated in achieving the fact at this layer;
  - For facts that appear in the current state this is empty;
  - The preference violation set for a newly appearing fact is that of the action that achieves it, and the union of those for the action's preconditions.
  - If a new path to a fact is discovered, use the lowest cost of its existing/new sets;
  - Otherwise the set at that layer is the set from the previous layer.
- Now we can build the RPG to the point at which:
  - No new actions appear **and**;
  - No preference violation sets are changing.
- This means we have to build the RPG to more layers, but it does mean that we can generate potentially longer relaxed plans that satisfy preferences.

# Preference Aware RPG

- Initially violations empty;
- Build forward generating sets;
- Violation set cost = sum cost of preferences in it.  
*(preference p1 (always (not (at B)))*

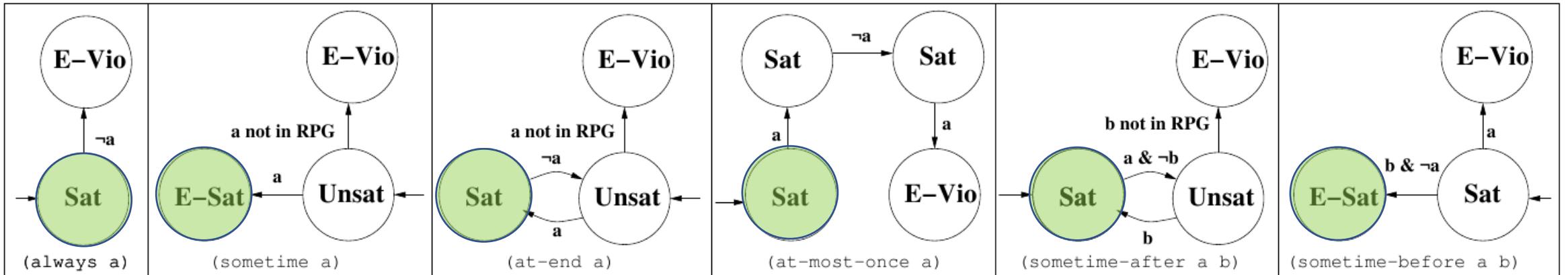


# Being More Specific

- Earlier we said: “The preference violation set for a newly appearing fact is that of the action that achieves it, and the union of those for the action's preconditions.”
- We didn't define the preference violation set for an action, it depends on:
  - The preference type; The trajectory so far; Whether that preference is already violated.
- Solution: back to automata.

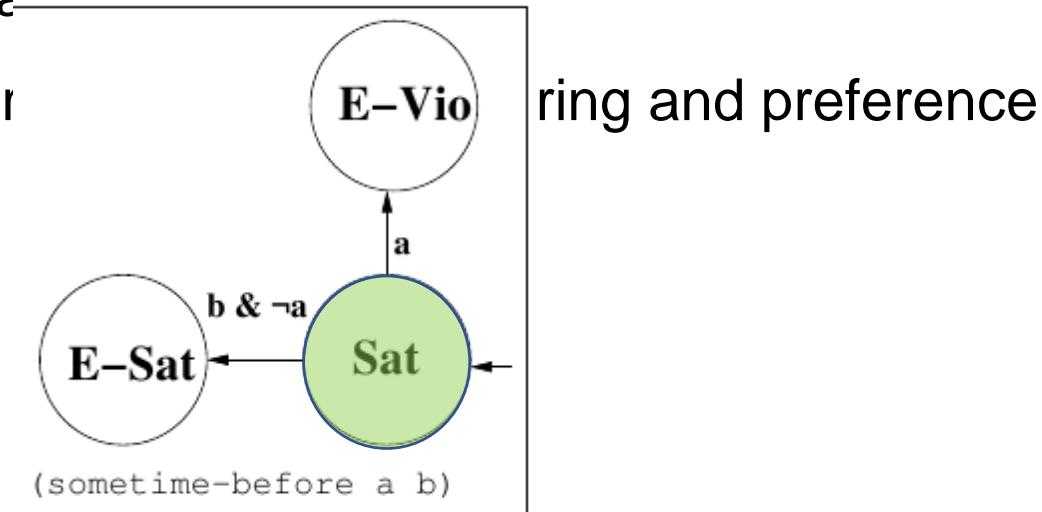
# Optimistic Automaton Positions

- In PDDL 3 there is always a 'best' position in the automata so we can maintain the best position possible for RPG layers (the further from E-Vio the better).
  - In general LTL we'd have to maintain *all* possible positions.
- A violates P when applied in I if, it activates a transition from the (all) automaton position(s) in I:
  - to E-Vio (e.g. sometime-before), (at-most-once);
  - to Unsat and the facts required to activate the transition back to a Sat state are not yet in the RPG (e.g. sometime after).



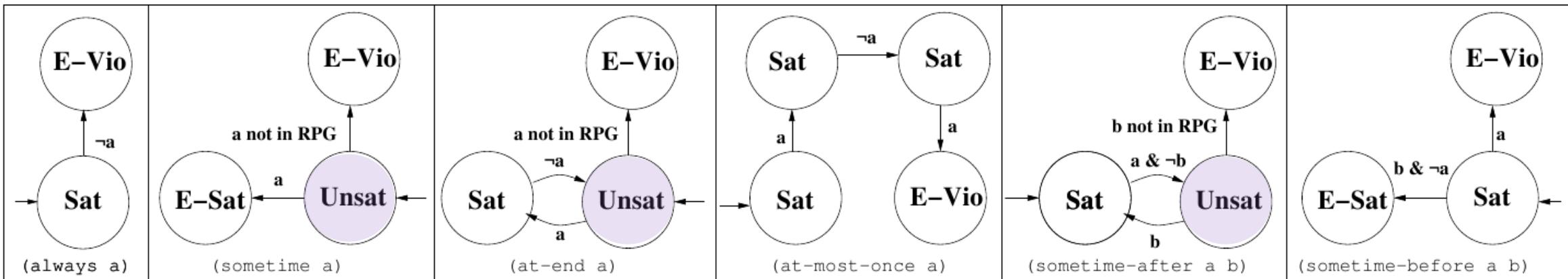
# Reappearing Actions

- If an action appears later in the RPG with a lower preference violation cost:
  - Add it again with the precondition that made the automaton transition;
  - Propagate to its effects and actions that rely on them by building further action layers until no further change in violation sets.
- Then continue building RPG as normal
- Termination criterion: no other violation sets unchanging.



# Solution Extraction I: Goals

- Not all preference problems have hard goals, so we need to decide what goals to achieve:
  - Some soft goals are explicit in the problem (at-end);
  - Others are implicit in currently Unsat preferences (sometime-after a b) (sometime p);
- For goal generation look to automata: any automaton which is currently unsat, add the condition on its transition to Sat as a goal in the RPG;
- Again more generally disjunction over paths to Sat (in practice pick one).



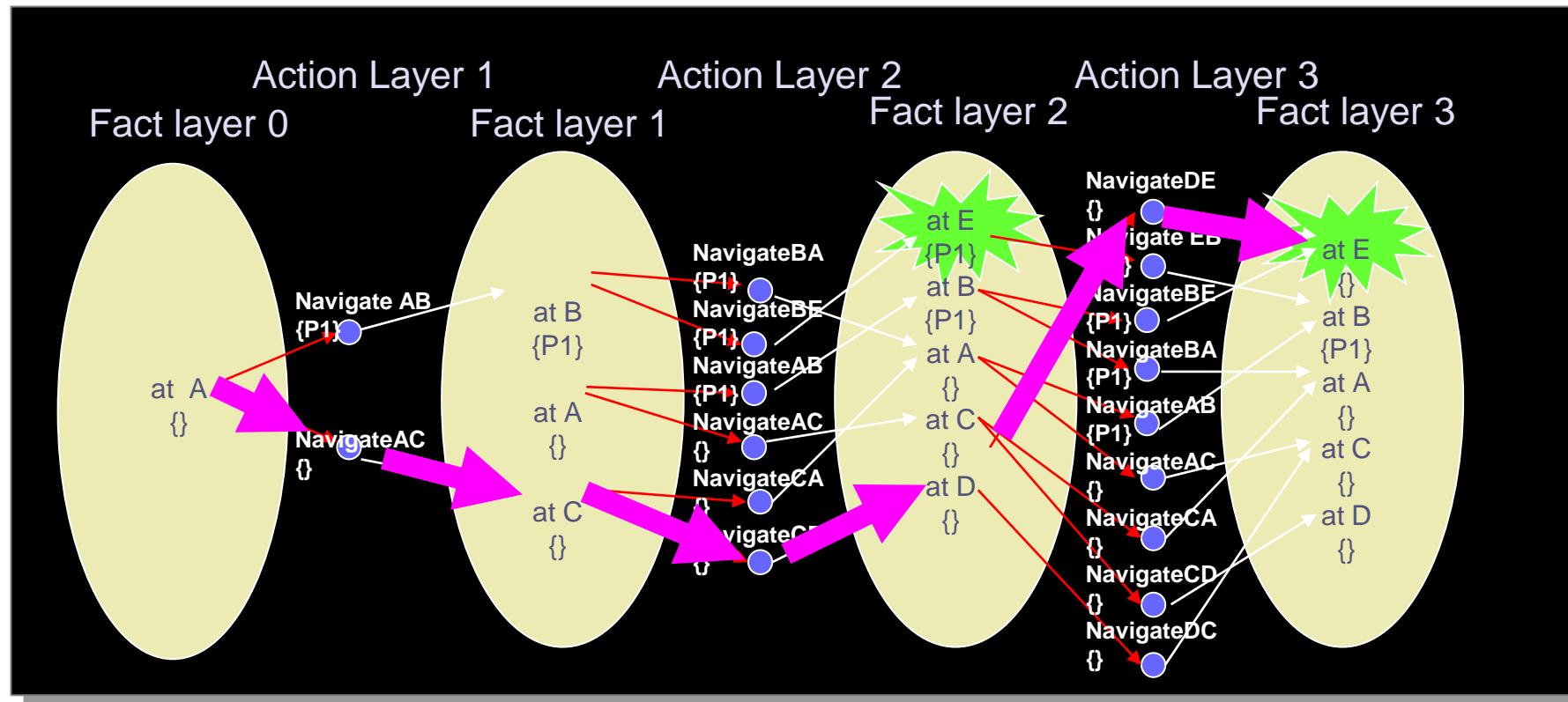
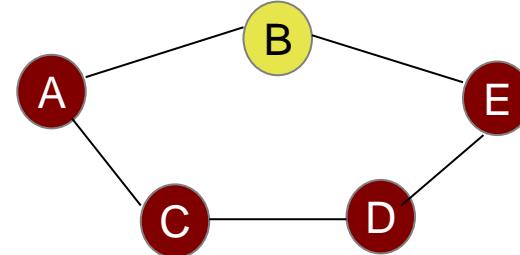
- If the transition does not appear in the RPG then preference is E-Vio not Unsat.

# Solution Extraction (II) RP Generation

- This now needs to be a bit more sophisticated:
  - Add preconditions/goals at earliest layer they appear with their lowest cost violation set;
  - This must be before the layer the action for which it is a precondition was applied.
- We must also think about which achievers to use:
  - Select the achiever that caused this fact's cost to be updated at this layer (recorded during graph building).
  - This is the achiever that caused its cost to decrease, and is thus on the lowest cost path.
- There's no need to worry about adding extra goals (e.g. if we had a (sometime-before a b) we don't need to add b as a goal if we chose a in extraction because the action that doesn't break that preference already has b as a precondition.

# Preference Aware Relaxed Plan Extraction

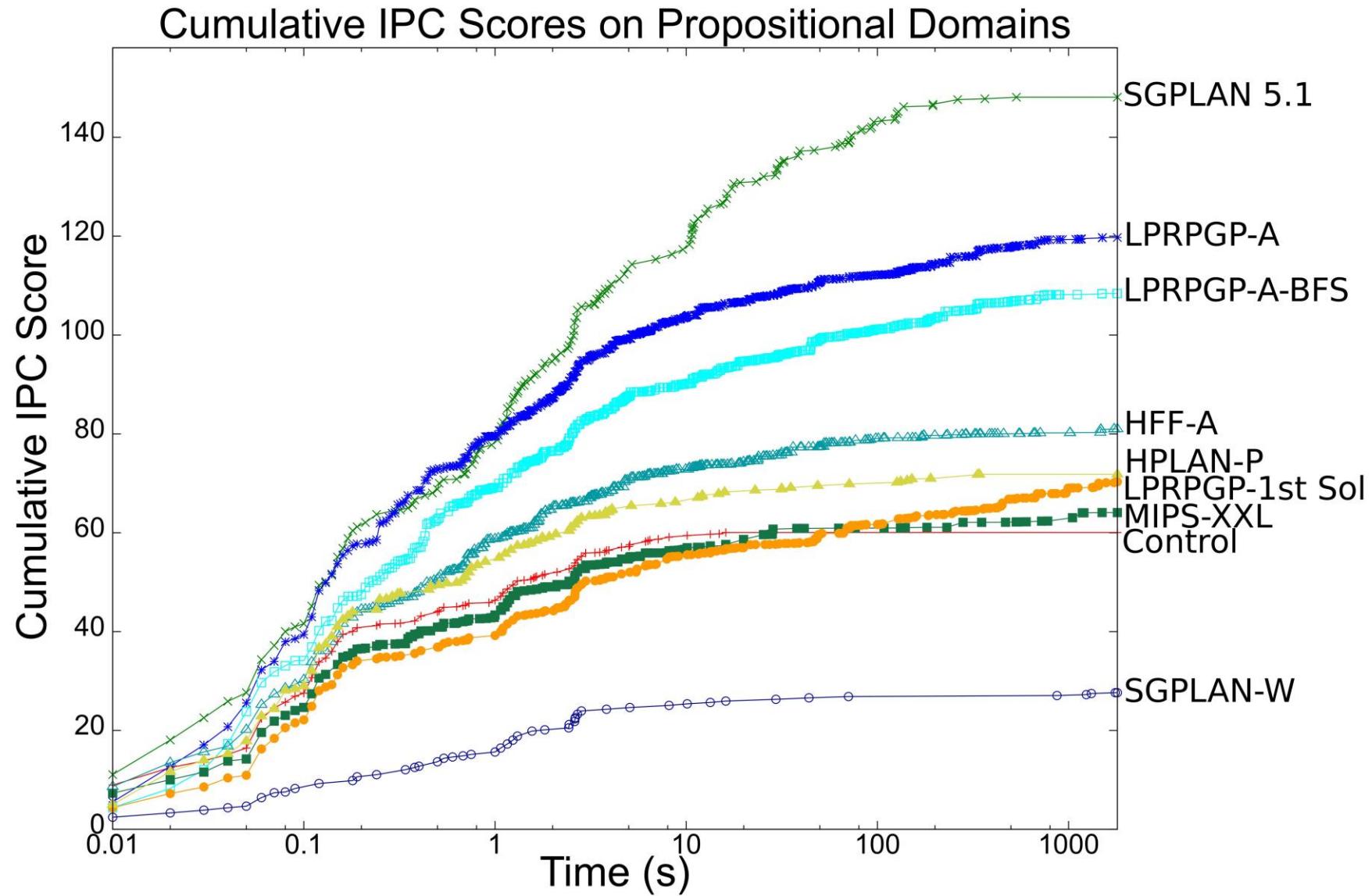
- Select a Goal;
- Select achiever;
- Add Preconditions to Goals.



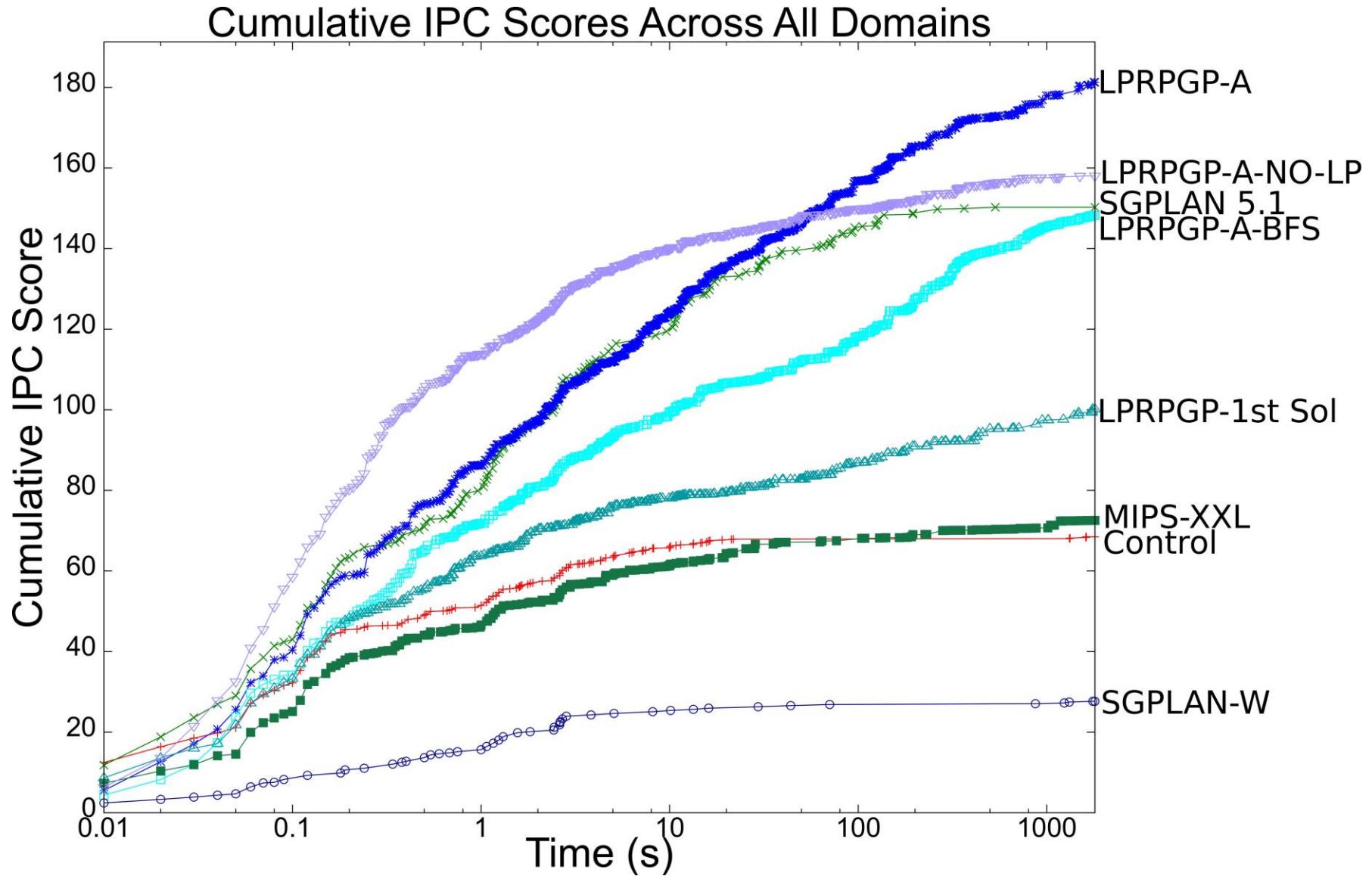
# LPRPG-P: Comparison to Other Planners

# Results

- State of the art:
  - **HPlanP** (propositional only);
  - **MIPS-XXL** (everything);
  - **SGPlan 5.1** (competition, generic);
  - **Control**: FF solving hard goals; or empty plan if no hard goals.
- Domains (IPC 2006):
  - Pathways (Chemical Reaction Synthesis);
  - Trucks (Logistics);
  - Storage (Warehouse Organisation);
  - Travelling Purchaser Problem (similar to Travelling Salesman);
  - Martian Rover Operations.



IPC Score:  $C^*/C$ :  $C = C^*$  implies 1,  $C > C^*$  implies  $< 1$

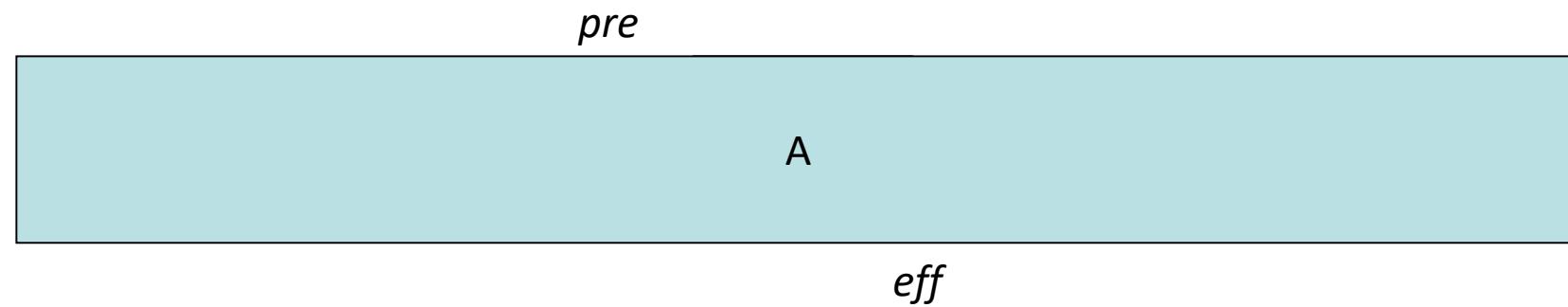


# Planning with Time

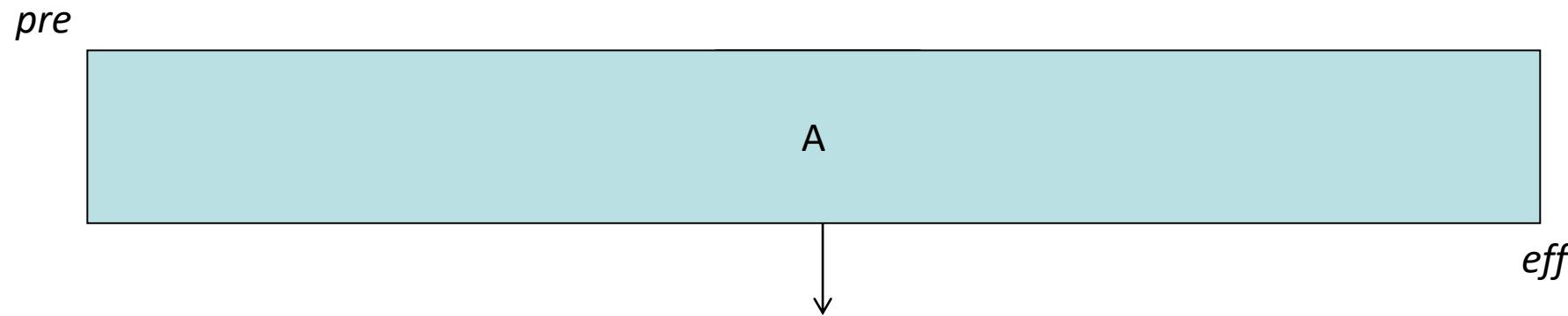
# Temporal Planning: Motivation

- .In general, activities have **varying durations**:
  - Loading a package onto a truck is much quicker than driving the truck;
  - Drinking a cup of tea takes longer than making it;
  - Procrastinating tasks takes longer than doing them;
  - ...

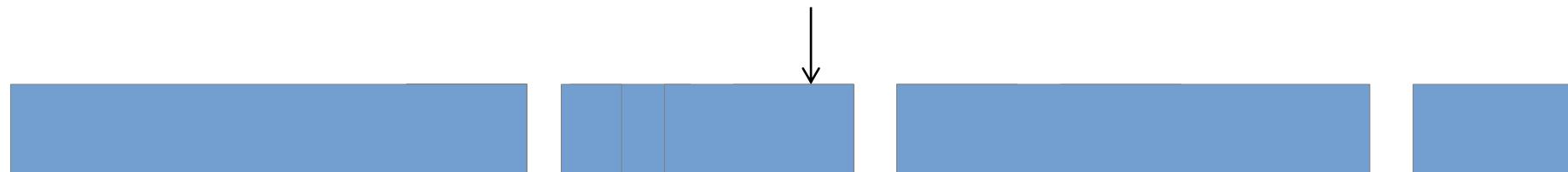
# Durative Actions?



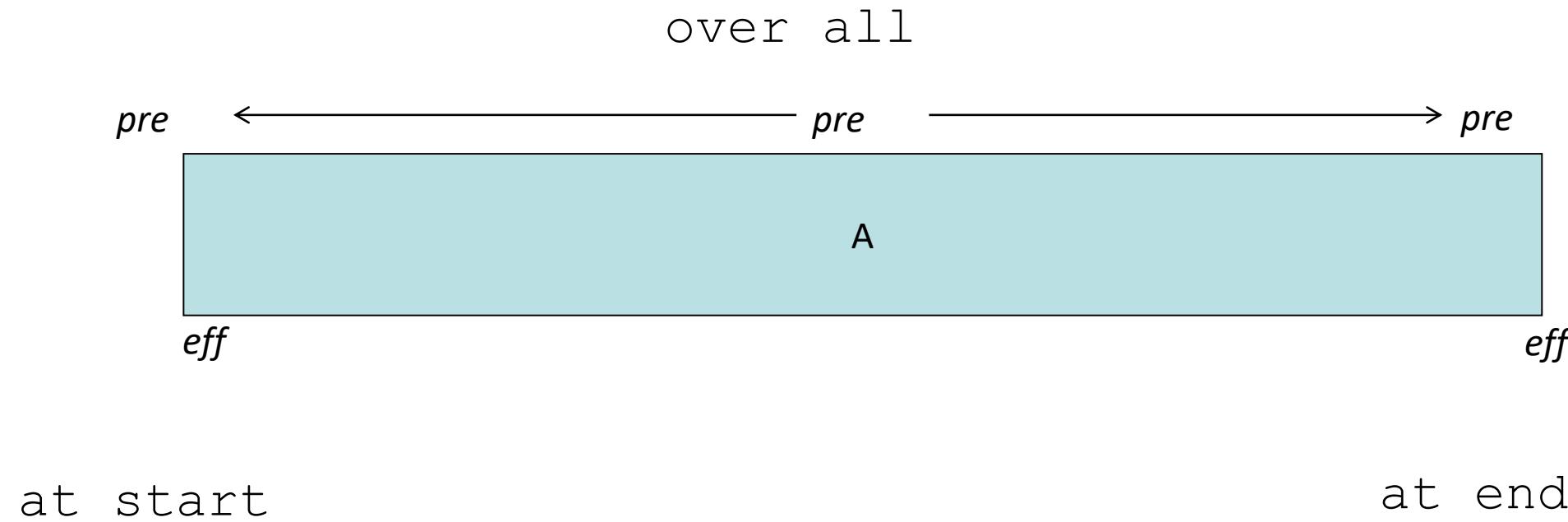
# Durative Actions?



***FF***



# Durative Actions in PDDL 2.1



# PDDL Example (i)

```
( :action LOAD-TRUCK
  :parameters (?obj - obj ?truck - truck ?loc - location)
  :duration (= ?duration 2)
  :precondition
    (and
      (at ?truck ?loc)
      (at start (at ?obj ?loc)))
  :effect
    (and (at start (not (at ?obj ?loc)))
         (at end (in ?obj ?truck))))
```

# Beware of Self-Overlapping Actions

```
( :durative-action LOAD-TRUCK
  :parameters
    (?obj - obj ?truck - truck ?loc - location)
    •duration' = ?duration 2)
  :precondition
    :condition
      (and (over all (at ?truck ?loc))
            (at start (at ?obj ?loc)))
    :effect (at end
              (and (at start (not (at ?obj ?loc)))
                    (at end (in ?obj ?truck)))))
```

# PDDL Example (ii)

```
( :durative-action open-barrier
  :parameters
    (?loc - location ?p - person)
  :duration (= ?duration 1)
  :condition
    (and (at start (at ?loc ?p)) )
  :effect
    (and (at start (barrier-open ?loc) )
          (at end (not (barrier-open ?loc)) ) )
```

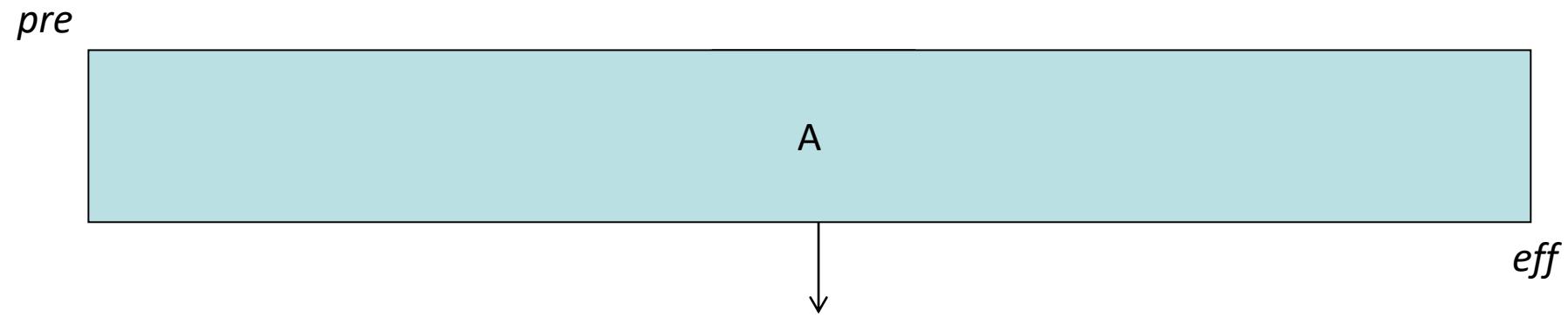
# PDDL Example (ii)

```
( :durative-action open-barrier
  :parameters
  (?loc - location ?p - person)
  :duration (= ?duration ?d)
  :condition
  (and (at start (at ?loc ?p)))
  :effect
  (and (at start (barrier-open ?loc))
        (at end (not (barrier-open ?loc)))))
```



gifbin.com

# Durative Actions?



**FF**

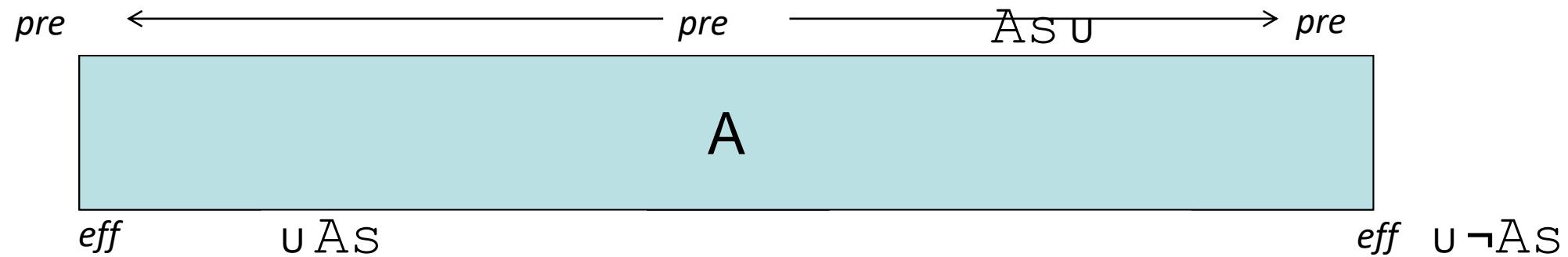
# PDDL Example (ii)

```
( :durative-action open-barrier
  :parameters
    (?loc - location ?p - person)
  :duration (= ?duration 1)
  :condition
    (and (at start (at ?loc ?p)) )
  :effect
    (and (at end start (barrier-open ?loc) )
          (at end (not (barrier-open ?loc)) ) )
```

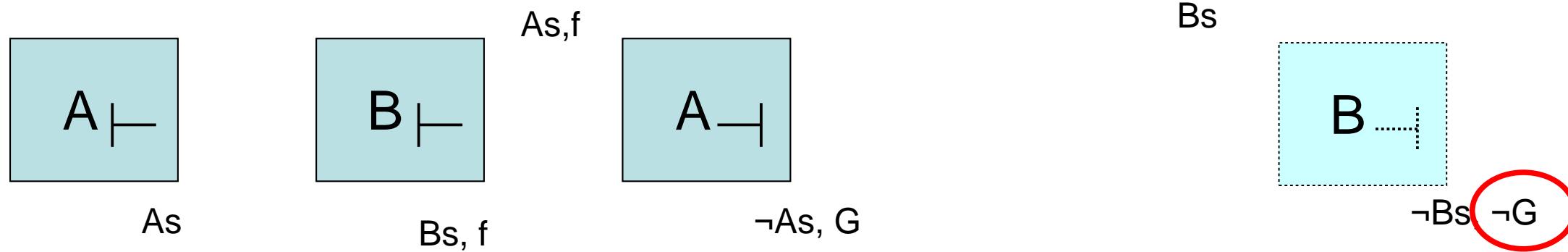
# Reasoning with PDDL 2.1 Durative Actions: Challenges

# Durative Actions in LPGP

(Fox and Long, ICAPS 2003)



# Planning with Snap Actions (i)



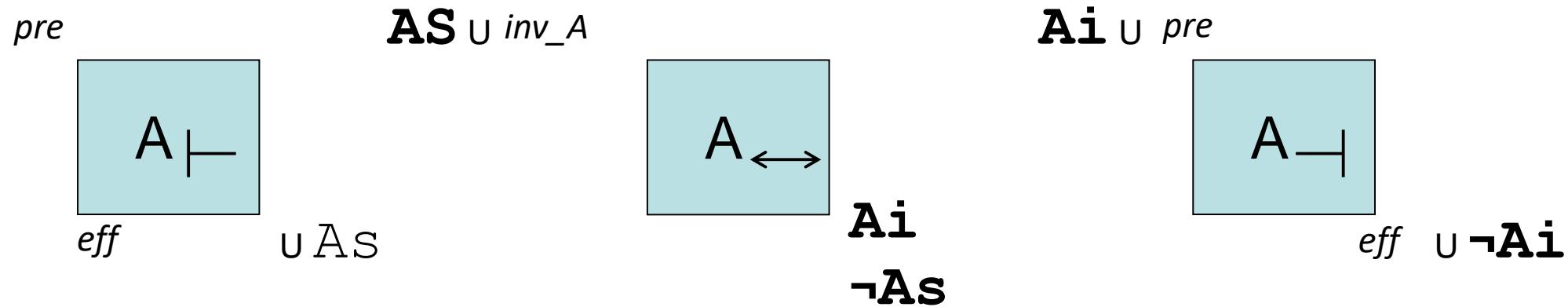
Challenge 1: What if  $B \dashv$  interferes with the goal?

.PDDL 2.1 semantics: **no actions can be executing in a goal state.**

**Solution:** add  $\neg$ As,  $\neg$ Bs,  $\neg$ Cs... to the goal

-(Or make this implicit in a temporal planner.)

# Planning with Snap Actions (ii)



.Challenge 2: what about **over all** conditions?

-If A is executing, inv\_A must hold.

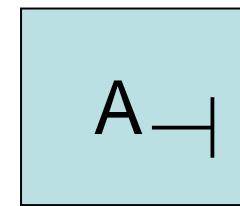
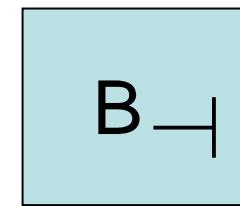
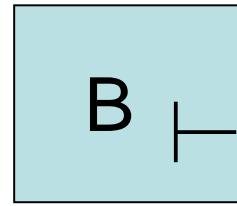
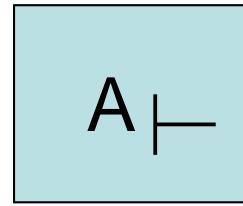
**.Solution:**

-In every state where As is true: inv\_A must also be true

-Or: (imply (As) inv\_A)

-Violating an invariant then leads to a **dead-end**.

# Planning with Snap Actions (iii)



**.Challenge 3: where did the durations go?**

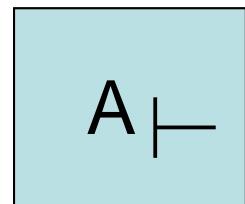
-More generally, what are the temporal constraints?

**-Logically sound ≠ temporally sound.**

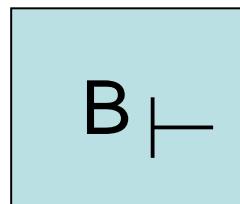
# Option 1: Decision Epoch Planning

- .Search with **time-stamped states** and a **priority queue** of pending end snap-actions.
  - See Temporal Fast Downward (Eyerich, Mattmüller and Röger, ICAPS 2009); Sapa (Do and Kambhampati, JAIR 2003), and others.
- .In a state  $S$ , at time  $t$  and with queue  $Q$ , either:
  - Apply a start snap-action  $A$  (at time  $t$ )
  - .Insert  $A$  into  $Q$  at time  $(t + \text{dur}(A))$
- . $S'.t = S.t + \varepsilon$

# Running through our example...

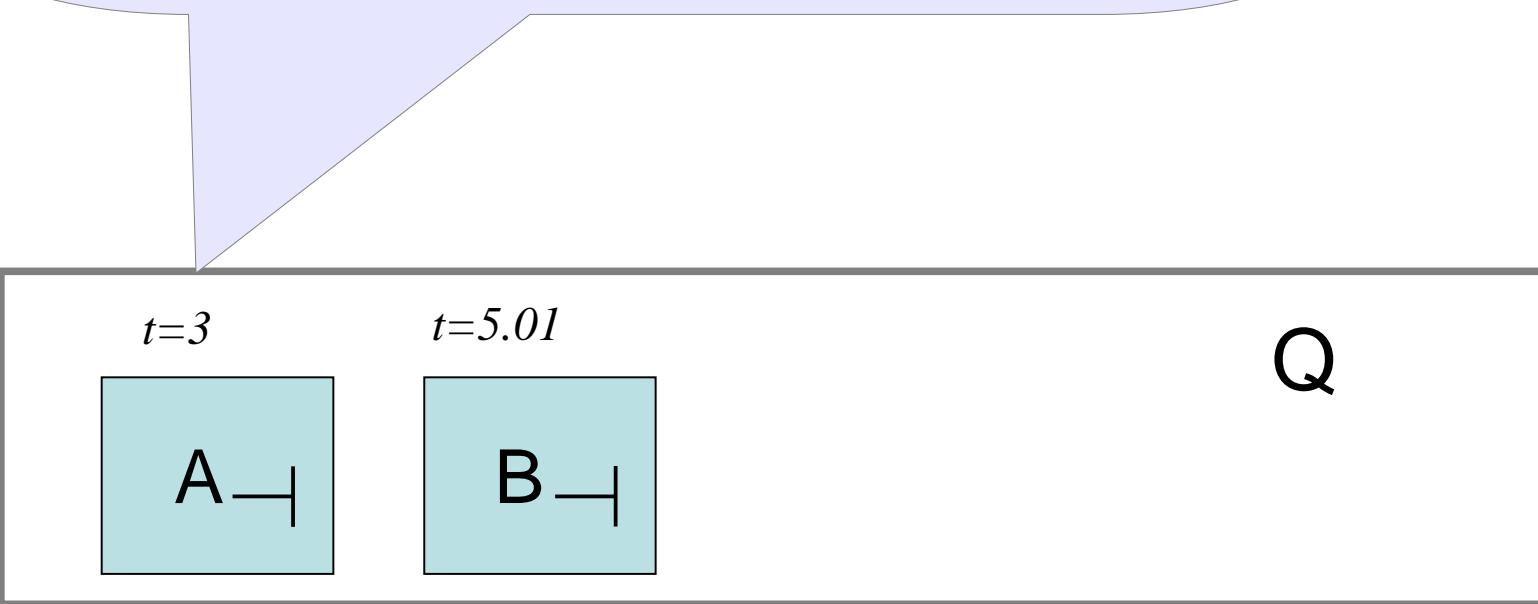


$t=0$



$t=0.01$

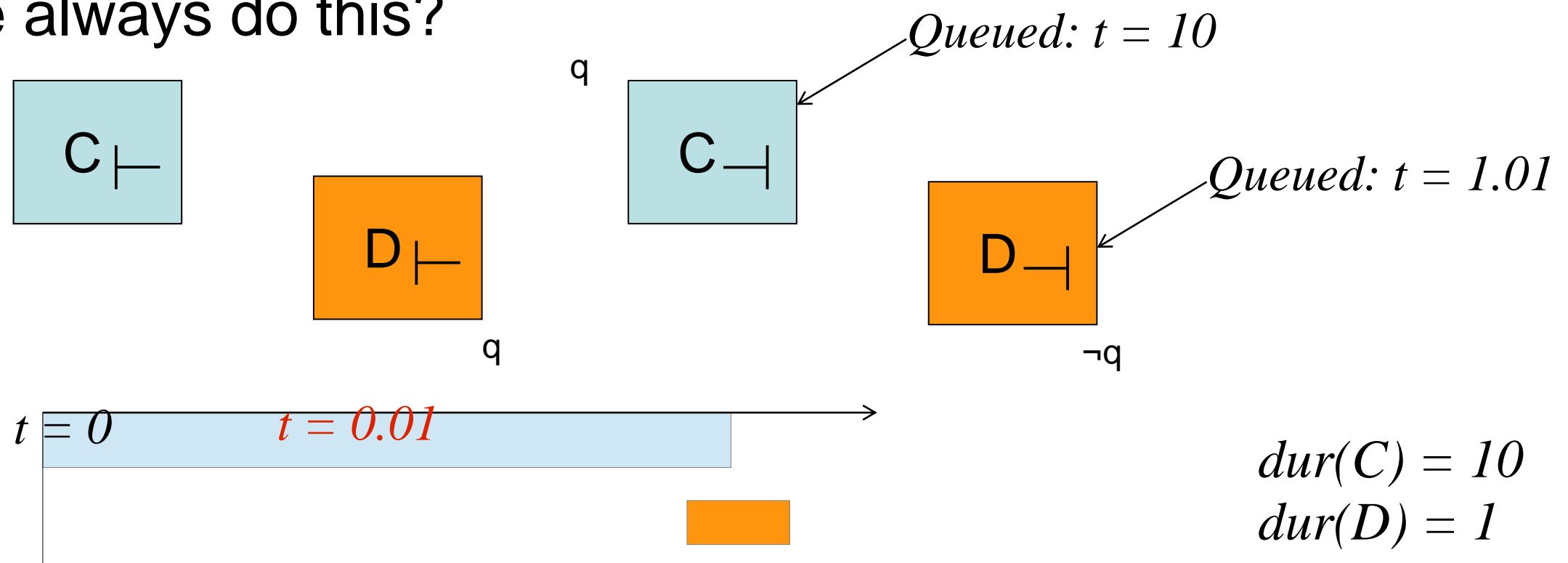
Can only choose A  $\perp$   
- eliminated the  
**temporally inconsistent**  
option (B  $\perp$  before A  $\perp$ )



What does this look like if we do B start first?

# Decision Epoch Planning: The snag

- Must fix start- and end-timestamps at the point when the action is started.
- Used for the priority queue
- Can we always do this?



# CRIKEY! 3

## Using Simple Temporal Networks

*"Planning with Problems Requiring Temporal Coordination."* A. I. Coles, M. Fox, D. Long, and A. J. Smith. AAAI 08.

*"Managing concurrency in temporal planning using planner-scheduler interaction."* A. I. Coles, M. Fox, K. Halsey, D. Long, and A. J. Smith. Artificial Intelligence.

# Option 2: a Simple Temporal Problem

.All our constraints are of the form:

. $\varepsilon \leq t(i+1) - t(i)$       (*c.f. sequence constraints*)

. $\text{dur}_{\min}(A) \leq t(A \rightarrow) - t(A \leftarrow) \leq \text{dur}_{\max}(A)$

.Or, more generally,  $lb \leq t(j) - t(i) \leq ub$

-Is a **Simple Temporal Problem**

.Good news – is **polynomial**

-Bad news – in planning, we need to solve it a lot....

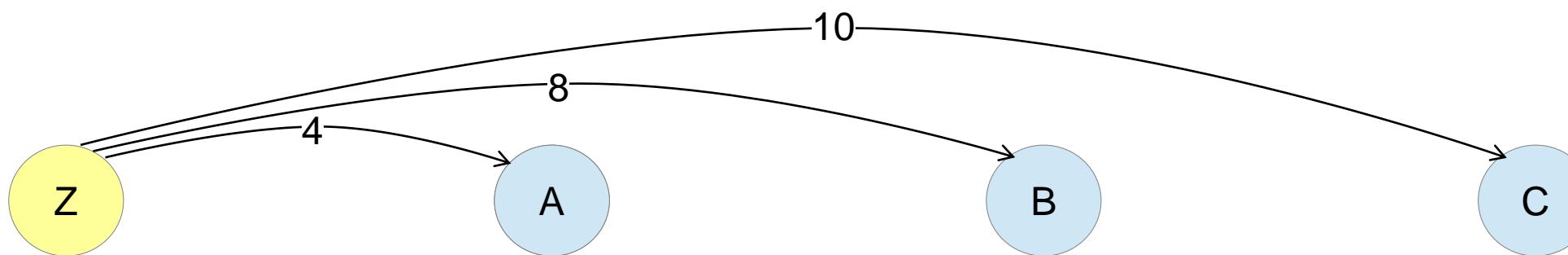
# Latest possible times? (Maximum Separation)

$$t(A) - t(Z) \leq 4$$

$$t(B) - t(Z) \leq 8$$

$$t(C) - t(Z) \leq 10$$

('A comes no more than 4 time units after Z')



# Latest possible times? (Maximum Separation)

$$t(A) - t(Z) \leq 4$$

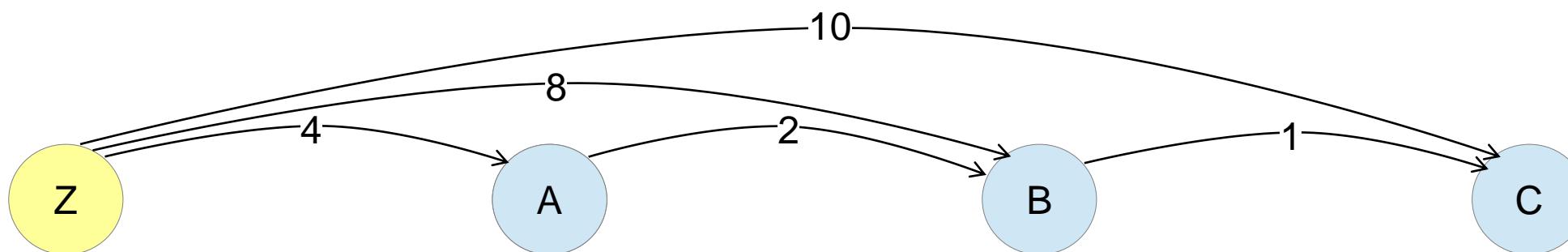
$$t(B) - t(Z) \leq 8$$

$$t(C) - t(Z) \leq 10$$

$$t(B) - t(A) \leq 2$$

$$t(C) - t(B) \leq 1$$

('B comes no more than  
2 time units after A')



# Earliest possible times? (Minimum Separation)

- .For latest possible time: find the **shortest path**
- .For earliest possible times...?

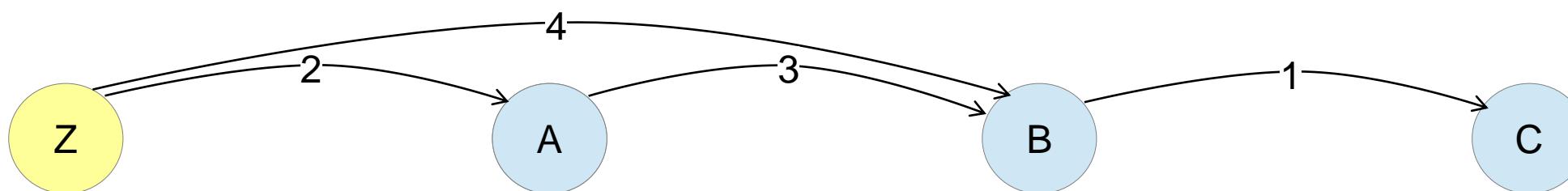
# Earliest possible times?

$$2 \leq t(A) - t(Z)$$

$$4 \leq t(B) - t(Z)$$

$$3 \leq t(B) - t(A)$$

$$1 \leq t(C) - t(B)$$



# Hacking algorithms

- .Longest path from Z to C?
- .= Shortest **negative** path from C to Z

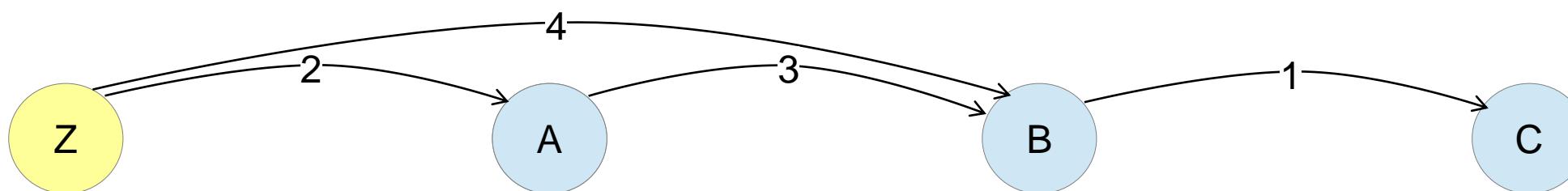
# Earliest possible times?

$$2 \leq t(A) - t(Z)$$

$$4 \leq t(B) - t(Z)$$

$$3 \leq t(B) - t(A)$$

$$1 \leq t(C) - t(B)$$



# Hacking algorithms

- .Longest path from Z to C?
- .= Shortest **negative** path from C to Z

$$2 \leq t(A) - t(Z)$$

Multiply both sides by -1:

$$-2 > -t(A) + t(Z)$$

$p \geq q$  is the same as  $q \leq p$ :

$$-t(A) + t(Z) < -2$$

Rearrange LHS:  
 $t(Z) - t(A) < -2$

# Earliest Possible Times?

$$2 \leq t(A) - t(Z)$$

$$4 \leq t(B) - t(Z)$$

$$t(Z) - t(A) \leq -2$$

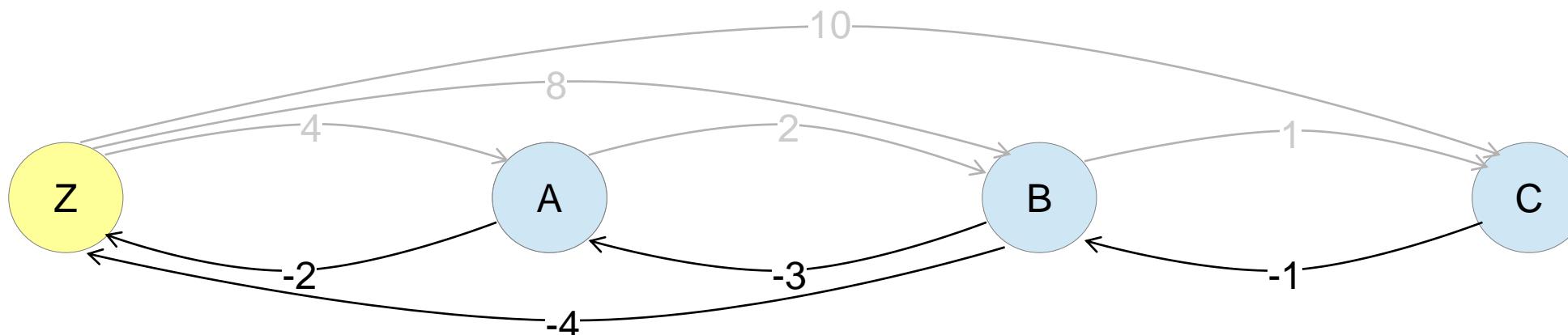
$$t(Z) - t(B) \leq -4$$

$$3 \leq t(B) - t(A)$$

$$1 \leq t(C) - t(B)$$

$$t(A) - t(B) \leq -3$$

$$t(B) - t(C) \leq -1$$



# Earliest Possible Times?

$$2 \leq t(A) - t(Z)$$

$$4 \leq t(B) - t(Z)$$

$$t(Z) - t(A) \leq -2$$

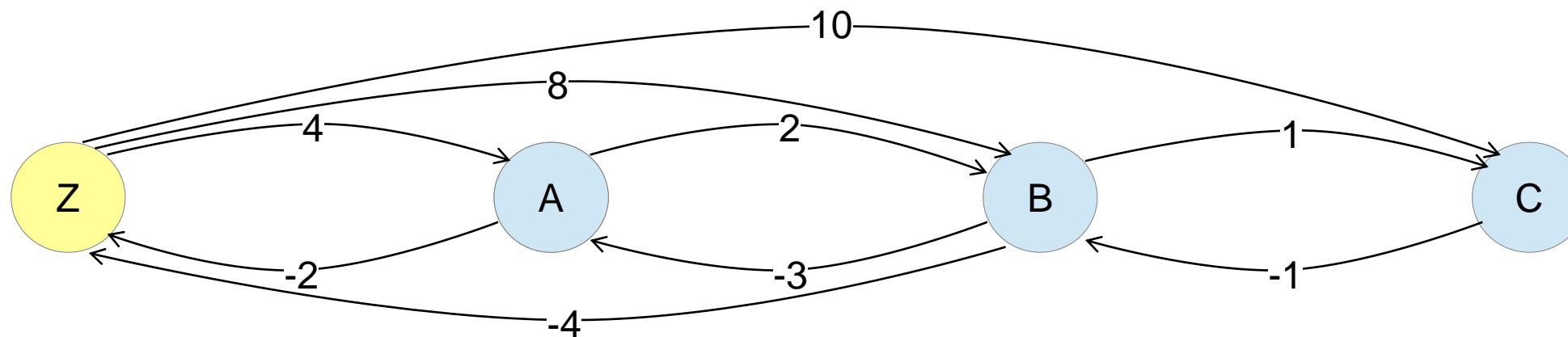
$$t(Z) - t(B) \leq -4$$

$$3 \leq t(B) - t(A)$$

$$1 \leq t(C) - t(B)$$

$$t(A) - t(B) \leq -3$$

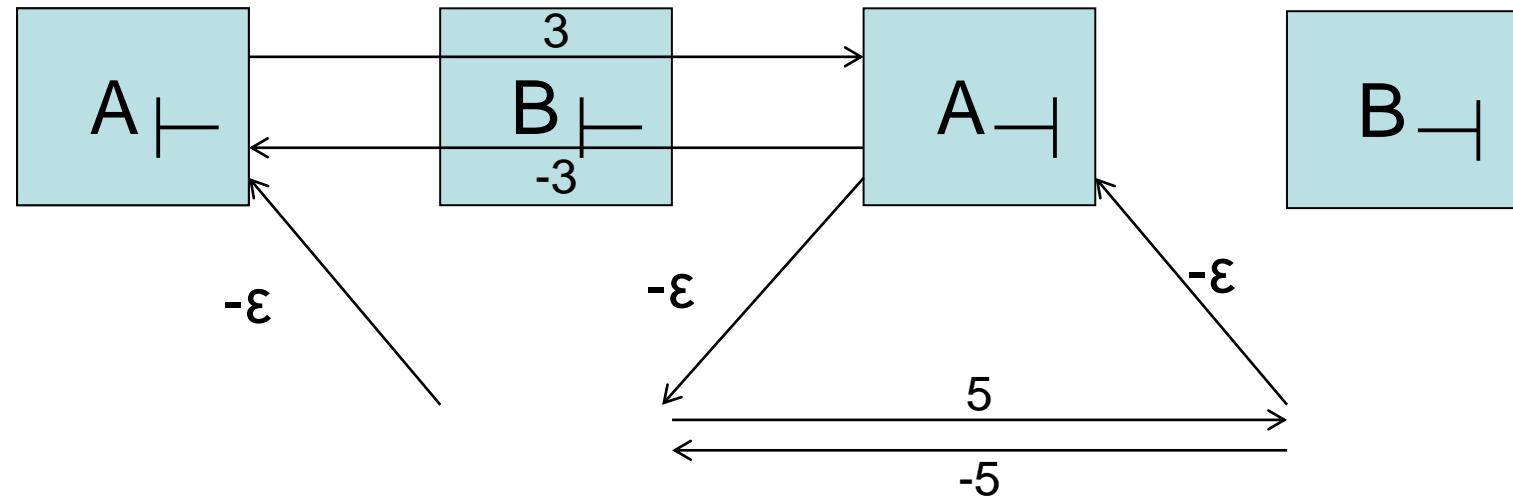
$$t(B) - t(C) \leq -1$$



# Simple Temporal Networks (i)

- .Can map STPs to an equivalent digraph:
  - One vertex per time-point (and one for 'time zero');
  - For  $lb \leq t(j) - t(i) \leq ub$ :
    - .An edge  $(i \rightarrow j)$  with weight  $ub$ .
    - .An edge  $(j \rightarrow i)$ , with weight  $-lb$
  - (c.f.  $lb \leq t(j) - t(i) \rightarrow t(j) - t(i) \leq -lb$ )

# Example STN



0.00: (A) [3]  
0.01: (B) [5]

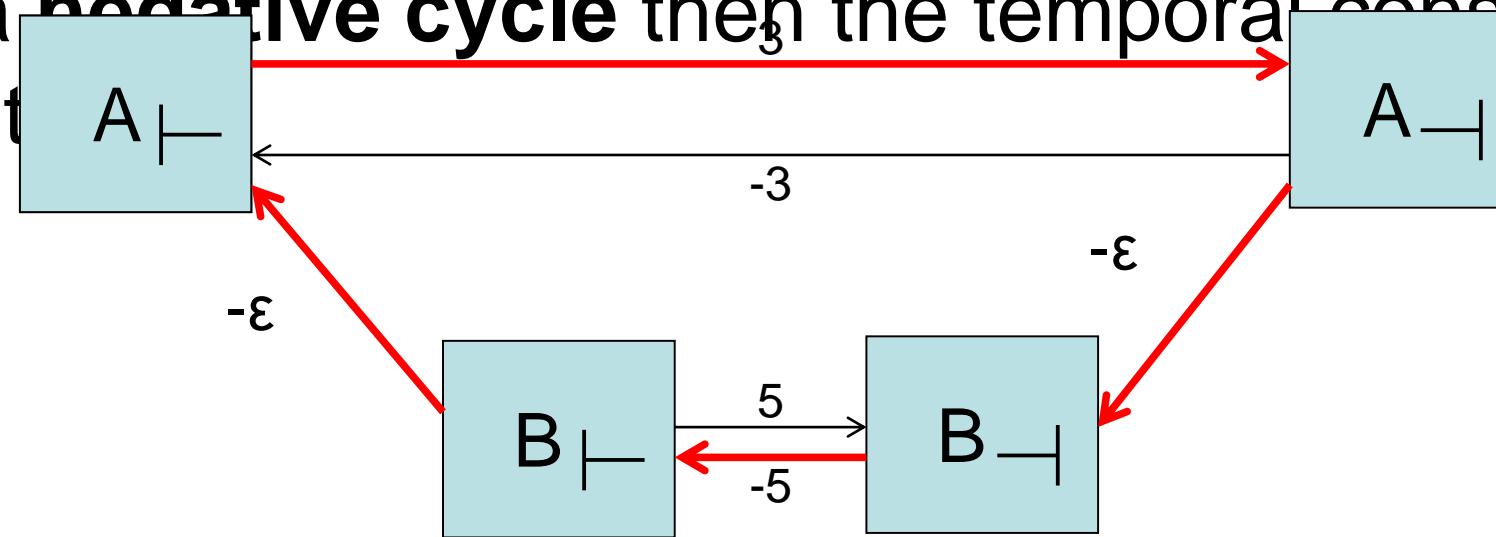
# Simple Temporal Networks (ii)

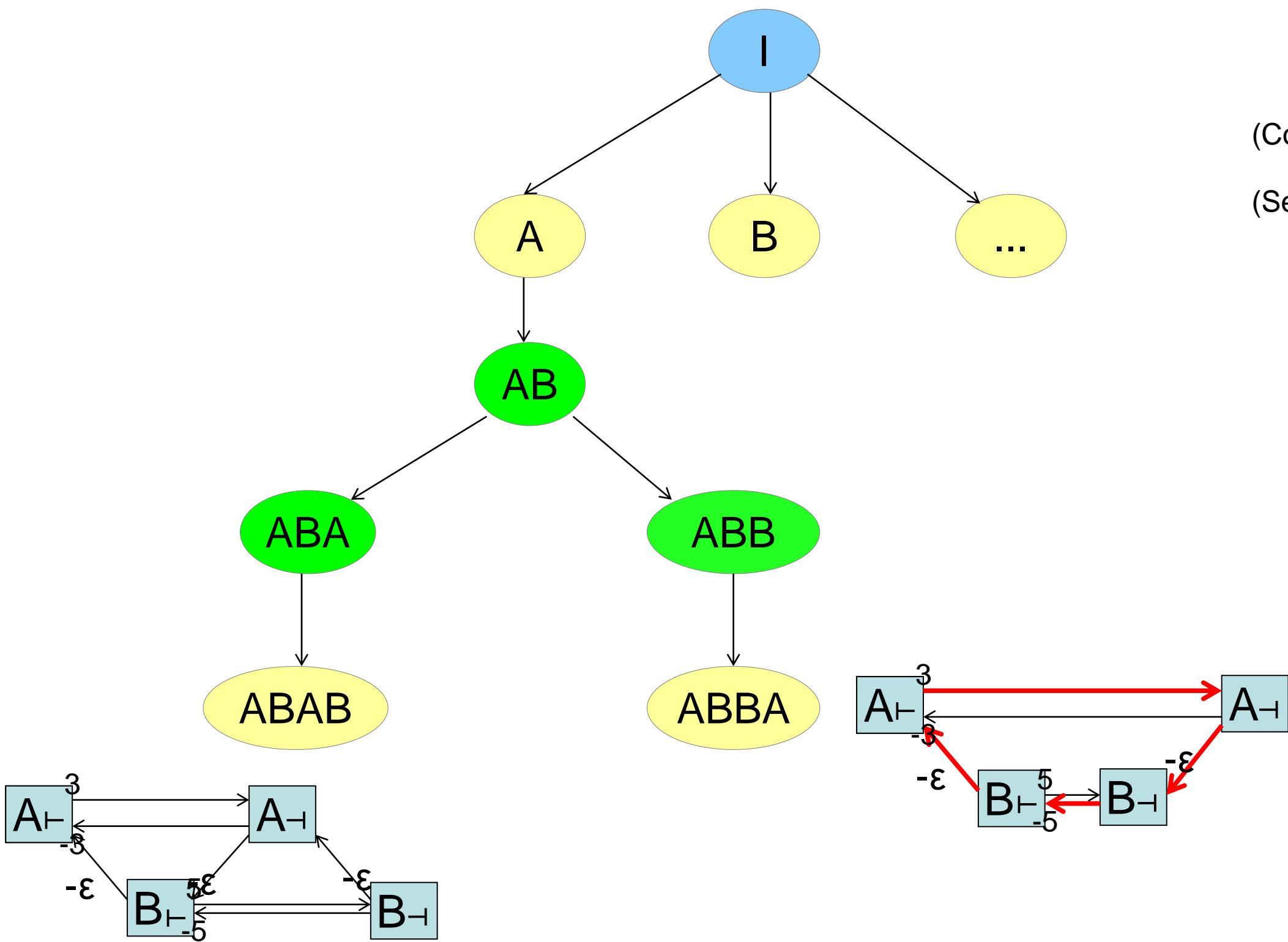
.Solve the shortest path problem (e.g. using Bellman-Ford) from/to zero

- $\text{dist}(0,j)=x \rightarrow$  maximum timestamp of  $j = x$

- $\text{dist}(j,0)=y \rightarrow$  minimum timestamp of  $j = -y$

.If we find a **negative cycle** then the temporal constraints are inconsistent





(Coles, Fox, Long and Smith,  
 (See also Halsey, Fox and Lo

# Memoisation in Temporal Planning

- .The **closed list** is a headache;
- .Classical planning:
  - Discard states that are the same (in terms of facts, or same/worse cost) as states already seen.
- .Temporal planning:
  - Facts don't tell us everything** – due to the temporal constraints, the plan steps matter too.
  - ...as does their order – plans with different **permutations** of actions are interestingly different

# Heuristics for Temporal Planning: Temporal Relaxed Planning Graph

# Heuristic Guidance

- .Essentially, three options:
  - i.Throw away time, use a non-temporal heuristic
  - .CRIKEY – FF's RPG heuristic, with snap actions.
  - ii.Use a **temporal reachability analysis**
    - .Sapa, CRIKEY3 – use an RPG with timestamped layers
    - .(c.f. TGP's temporal planning graph, Smith and Weld, 1999)
  - iii.Approximate time as **action costs**
    - .e.g.  $\text{cost}(A) = \text{dur}(A)$ : **inadmissible**
    - .(Haslum, 2009) – **admissible**, by finding **bounded concurrency**.

# Revisiting the RPG

- The RPG consists of time-stamped fact and action layers.
- Without time, to evaluate a state  $S$ :
  - Fact layer  $f=0.0$  contains the facts in  $S$ ;
  - Action layer  $a=0.0$  contains actions whose preconditions are satisfied in  $f=0.0$ ;
  - Effects of actions appear in the next layer: Actions at 0.0 give facts at 1.0; actions at 1.0 give facts at 2.0; etc...

# A Temporal RPG

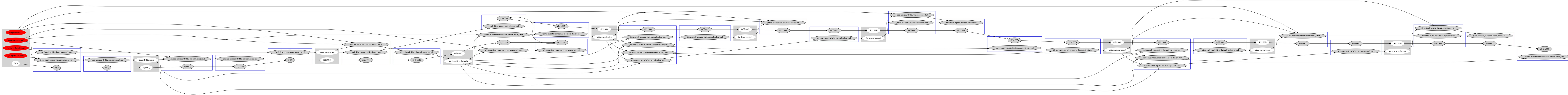
- .Sure why not
- .Instead of being delayed by ‘1.0’ delay effects by epsilon
  - e.g. facts from actions at 1.0 appear in layer 1.01
- .What about durations?
  - If the start appears in layer  $t$ , the end cannot appear until layer  $t + \text{dur}$

# Solution Extraction

- .Similar Process starting from the goals.
- .If we add  $A_F$  to the RP we had better also add  $A_{\neg}$ .
  - And achieve it's preconditions.
  - Similarly for adding  $A_{\neg}$  need  $A_F$ .
- .Also need to achieve invariants in order to be able to apply  $A_F$ .
  - No need to maintain them as this is relaxed planning.

# Properties of the TRPG Heuristic

- .Is the relaxed plan length admissible?
- .Can we get an admissible estimate of the time we can achieve a fact in the TRPG?
- .Is the relaxed plan makespan admissible?



# POPF

# Temporal Planners

- .Forward Search + STN = CRIKEY 3
  - Order each action *after* the previous action in the plan
  - Create a total order
- .POPF: Partial Order Planning Forwards
  - Based on CRIKEY 3
  - Only put ordering constraints between actions when they are needed.

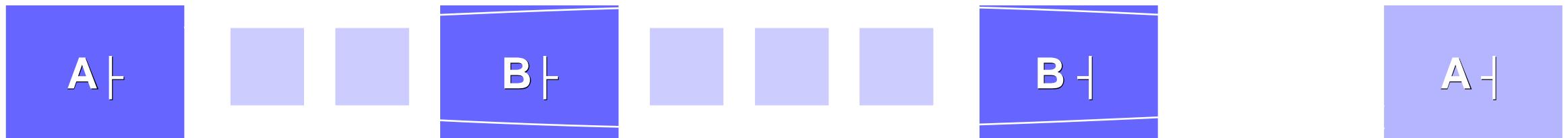
# Forward Chaining Temporal Planning

- .A state  $S$  is a tuple  $\langle F, V, P, T \rangle$  of:
  - Propositional Facts
  - Values of numeric task variables
  - The Plan to reach  $S$
  - The Temporal Constraints on the steps in  $P$
- .The plan consists of the starts and ends of actions:
  - $A \vdash$  and  $A \dashv$  denote the start/end of  $A$ , resp.

# Total Orders of Start/End Actions

- .Two actions, A and B:
  - B is longer than A;
  - No interaction between  $A \uparrow$  and  $B \uparrow$ ;
  - But,  $B \downarrow$  must precede  $A \downarrow$
- .The planner chooses a (partial) plan:

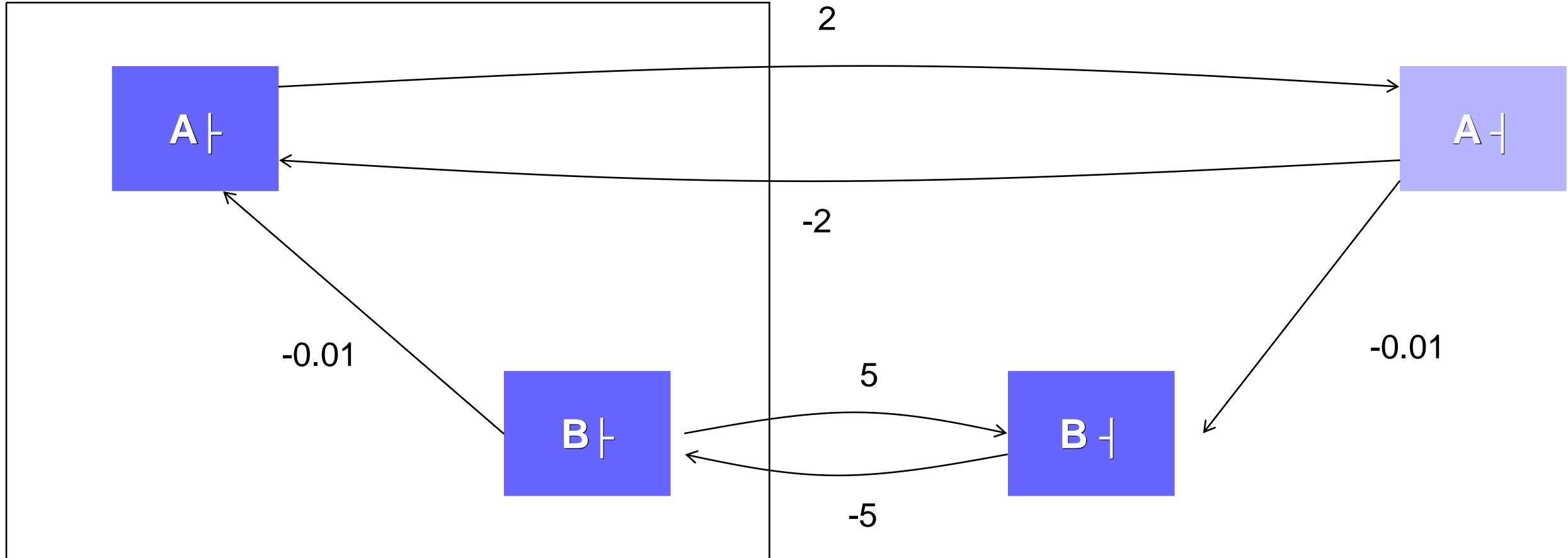




- .Because  $A \vdash$  was added to the plan before  $B \vdash$ , they are ordered as shown (in a total-order).
- .But,  $A \dashv$  will not be applicable until after  $B \dashv$
- .The planner will have to backtrack, over all the intermediate

# Reducing Commitment

- .Record additional information at each state concerning which steps achieve / delete / depend on each fact.
- .Use this information to commit to fewer ordering constraints
- .Still resolve threats based on the intuition of forward-chaining expansion: new actions cannot threaten the preconditions of earlier actions.
  - That is allow promotion (new action comes after link it threatens) but not demotion.



**0.00:**    **(action B)**    **[5.00]**  
**3.01:**    **(action A)**    **[2.00]**

# POPF: Details and Worked Example

# Extending the State: Propositional

- .To capture ordering information we add:
- . $F_+$ ,  $F_-$ , where  $F_+(p)$  ( $F_-(p)$ ) is the index of the step that most recently added (deleted)  $p$
- . $FP$ , where  $FP(p)$  is a set of pairs  $\langle j, d \rangle$ :
  - $\langle j, \varepsilon \rangle$  denotes that step  $j$  has an instantaneous condition on  $p$  (at start or at end)
  - $\langle j, 0 \rangle$  denotes that step  $j$  marks the end of an action with an overall condition on  $p$

Look at the light-match action for start achieve/end delete invariant

# Light Match

:durative-action LIGHT\_MATCH

:parameters (?m - match)

:duration (= ?duration 8)

:condition

(and (at start (unused ?m))

**(over all (light ?m))**

:effect (and

(at start (not (unused ?m)))

**(at start (light ?m))**

# Preconditions after adders

For each `at start` condition  $p$ :

$$t(F^+(p)) + \varepsilon \leq t(i)$$

# Over all conditions after adders

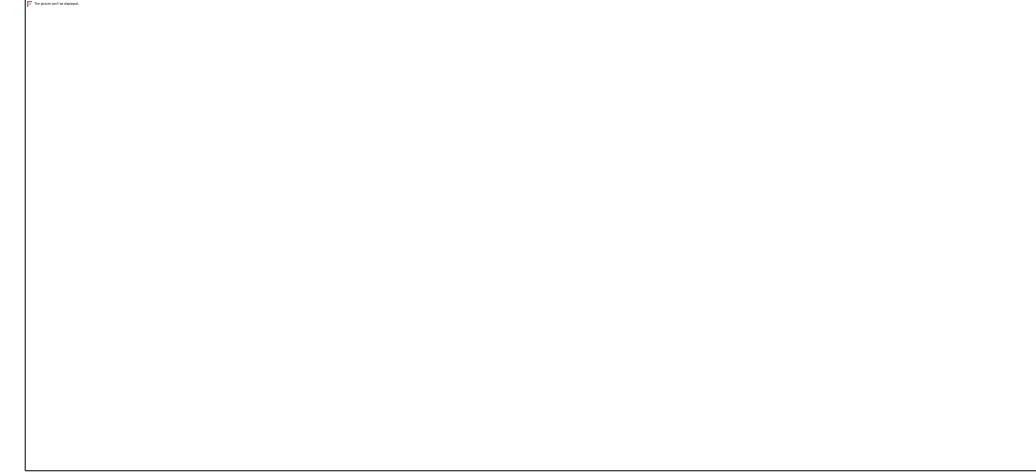
For each over all condition  $p$ :

If  $F^+(p) \neq i$ ,  $t(F^+(p)) \leq t(i)$

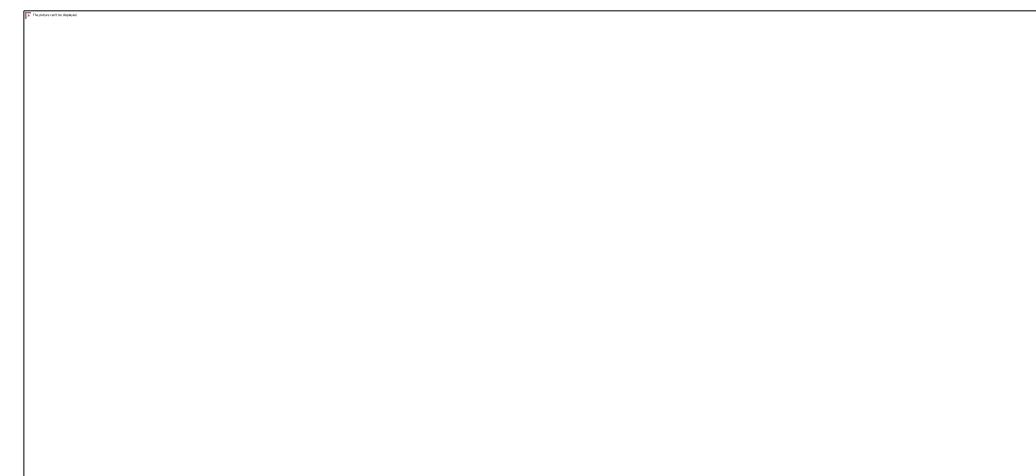
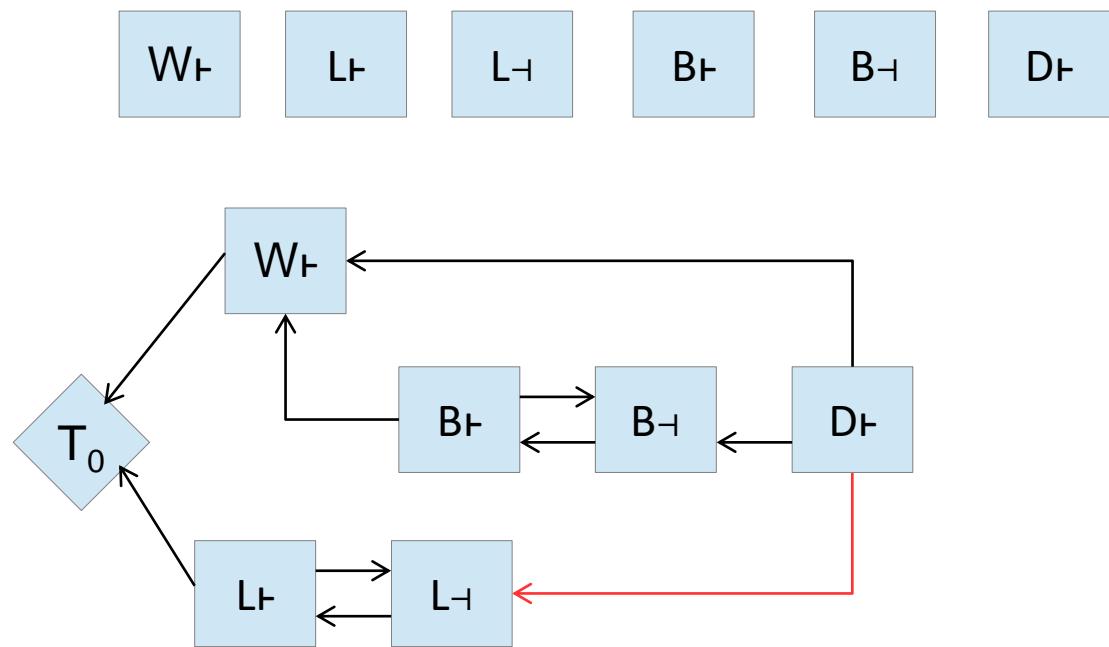
(Actions can achieve their own over all conditions.)

# Running Example

- .Driverlog problem with shifts;
- Driver works a 6 hour shift;
- Durative action work
- .Duration 6;
- .Adds (working driver) at start, deletes it at end.
- Drive, Board and Alight actions have overall condition (working driver);
- Load and Unload do not.
- .Must deliver package from A to E.



# Running through the Example



Disembark renamed to Alight

# Deletes after preconditions

For each `at start` delete effect  $p$ , come after all steps with a precondition on  $p$ :

$$\forall \quad \langle j, d \rangle \text{ in } FP(p), t(j) + d \leq t(i)$$

...and after the last adder of  $p$  (so we know if  $p$  is true or false):

$$t(F^+(p)) + \varepsilon \leq t(i)$$

Then, set assign  $F^-(p) = i$ , and clear  $FP(p)$

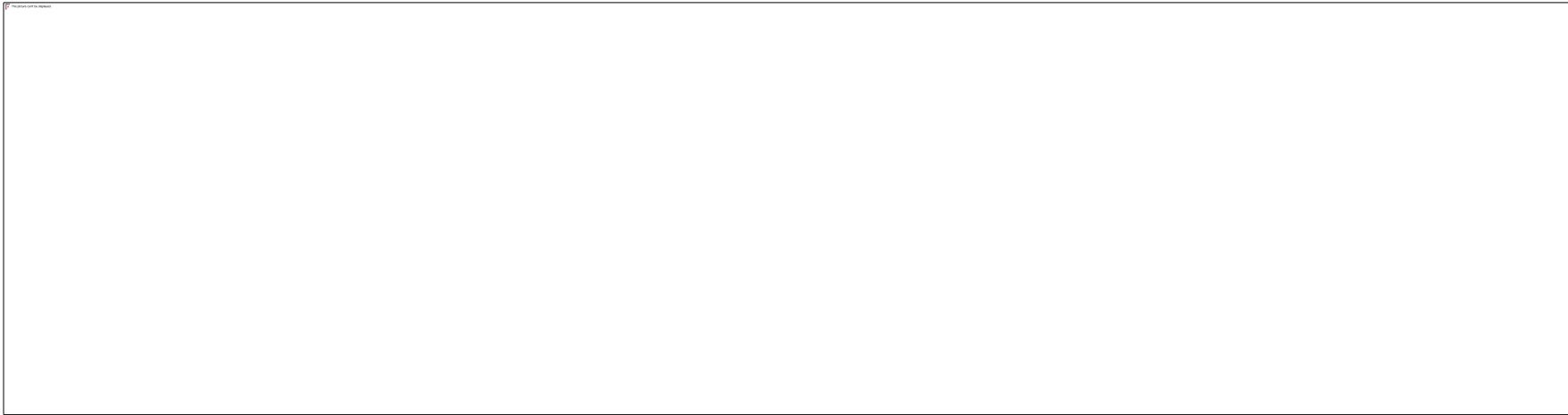
# Adds come after deletes

For each `at start` add effect  $p$

$$\text{if } F^-(p) \neq i, t(F^-(p)) + \varepsilon \leq t(i)$$

Then, set  $F^+(p) = i$ .

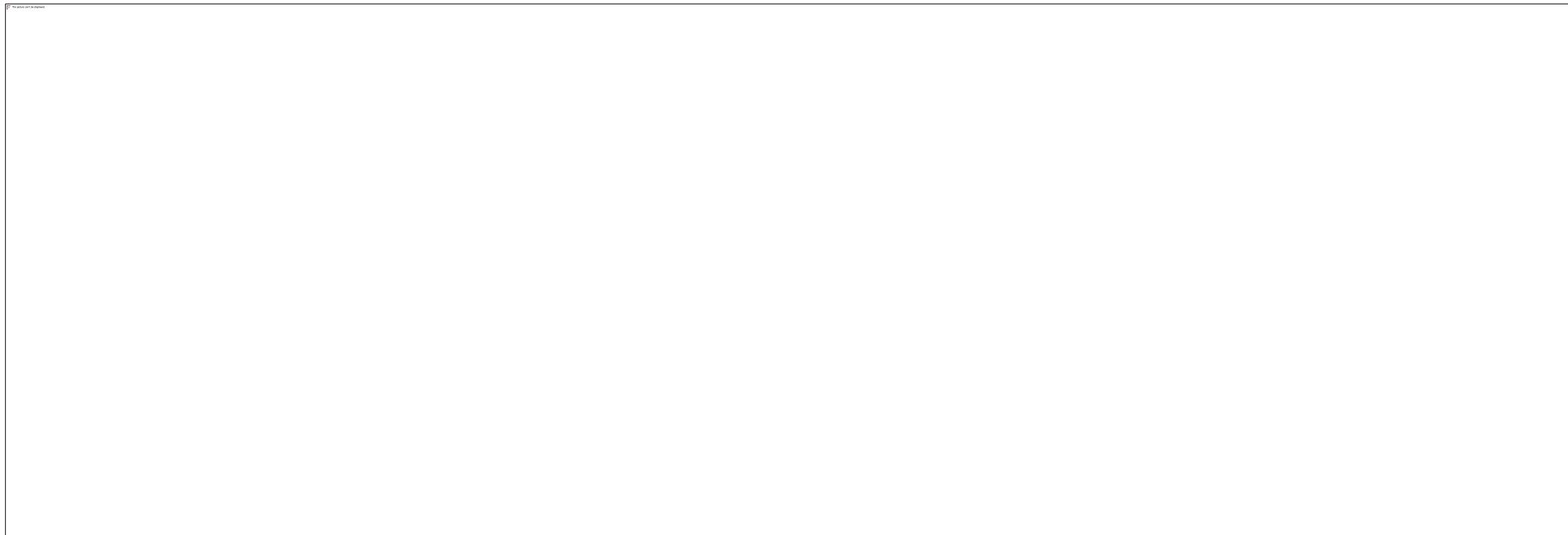
# POPF STN For This Problem



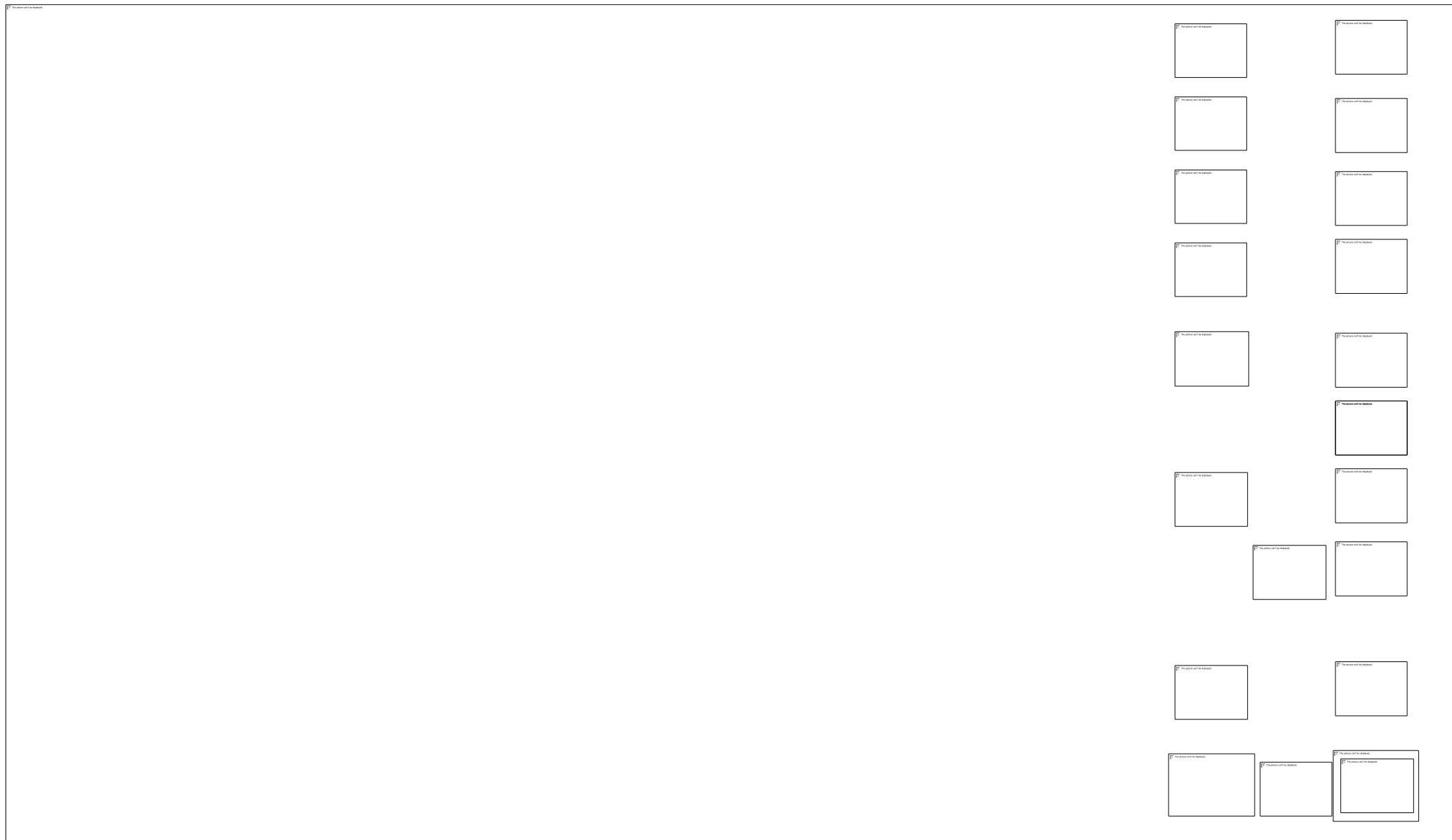
- Transitively implied edges omitted for clarity:
  - e.g. all the drive/board ends before work end;
  - All the drive/board starts after work start.

# Why is this Good?

1 partial order plan with 1 STN represents many totally ordered plans.



# These Plans and More...



# Time Taken



# Makespan (plan duration)

# Test 2: Time Taken (Deadlines)

# Compression Safety

# Search space size

- .Each durative action = two plan steps
- .Search space **at least twice as big**
- .Can we do anything about this?

# Why did we start this action?

```
( :durative-action LOAD-TRUCK
  :parameters
    (?obj - obj ?truck - truck ?loc - location)
  :duration (= ?duration 2)
  :condition
    (and (over all (at ?truck ?loc))
         (at start (at ?obj ?loc)))
  :effect
    (and (at start (not (at ?obj ?loc))))
```

# Why did we start this action?

```
( :durative-action open-barrier
  :parameters
    (?loc - location ?p - person)
  :duration (= ?duration 1)
  :condition
    (and (at start (at ?loc ?p)) )
  :effect
    (and (at start (barrier-open ?loc) )
          (at end (not (barrier-open ?loc)) ) )
```

# Compression Safety

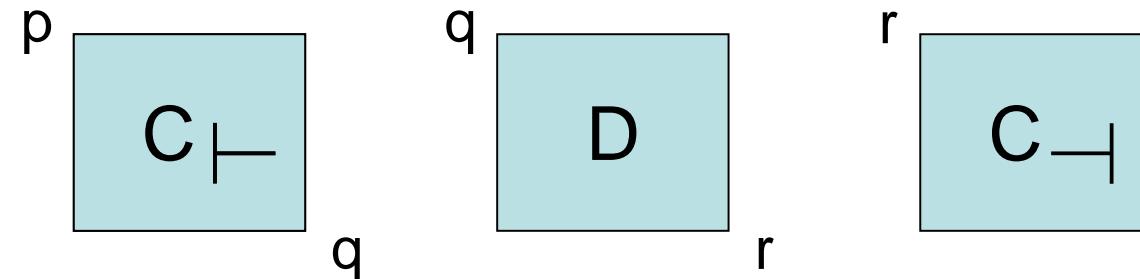
- .In general once the action  $C_+$  has been applied,  $C_-$  will be considered for application in every state in which its preconditions are satisfied.
- .But our intuition is that if the end of the action does nothing bad, we may as well end it straight away.
- .Trucks arriving at locations = yay
- .The barrier coming back down = nope
- .Formally?

# The end-effect rule

- .There must be no end-delete effects
- .Delete effects make preconditions false, which is bad:
  - if we immediately add  $C_{\perp}$  after  $C_F$  in this case, then we might not be able to use a fact that  $C$  added, e.g. the match domain.

# The end-precondition rule

- .If an action has end-preconditions that aren't also over all conditions, we can't just end it – maybe we have to do something first.
- . $C \vdash$  requires p, adds q. D requires q, adds r.  $C \dashv$  requires r.
- pre ( $C \dashv$ ) might not be satisfied.



# Compression Safe End Preconditions

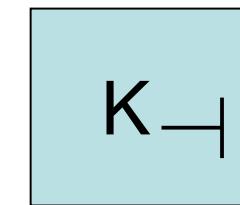
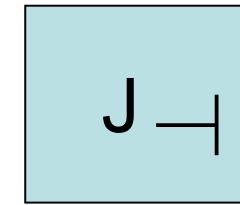
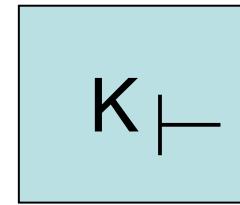
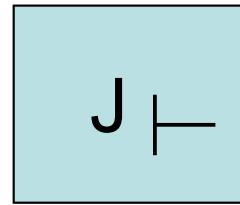
- .In general once the start of an action ( $C_{\vdash}$ ) has been applied Invariants of  $C$  will be maintained until  $C_{\dashv}$  is applied (no action may be applied that violates them).
- .If  $\text{pre}(C_{\dashv}) \subseteq \text{Pre}(C_{\leftrightarrow})$ ,  $C_{\dashv}$  is always applicable so we don't need to worry about something deleting  $\text{pre}(C_{\dashv})$ .

# Putting it together

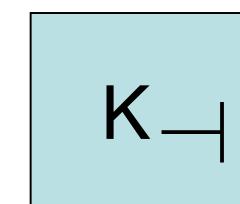
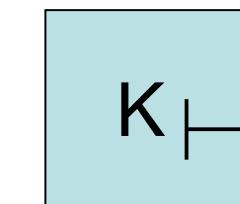
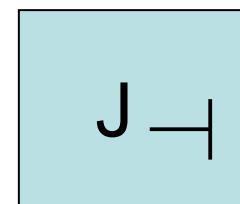
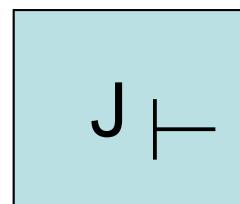
- .An action is compression safe if:
  - It has no end delete effects;
  - and no end preconditions that are not over all conditions.
- .In general once the start of an action ( $C_{\vdash}$ ) has been applied its end ( $C_{\dashv}$ ) will be considered in every possible state (in which its end preconditions are satisfied) until it has been applied.
- .But if the action is compression safe we can just add both the start and end of the action to the plan at the same time.

# What about quality?

- Two actions J and K, totally independent
- Most efficient plan:

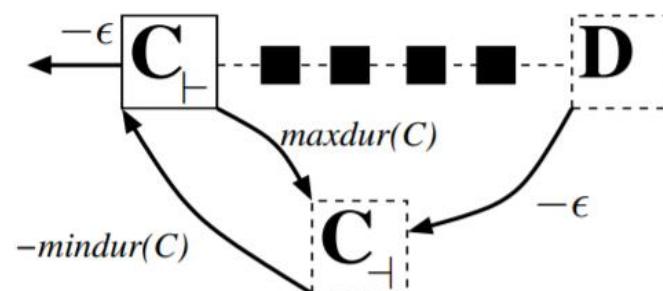


- But if we end them as soon as they start?

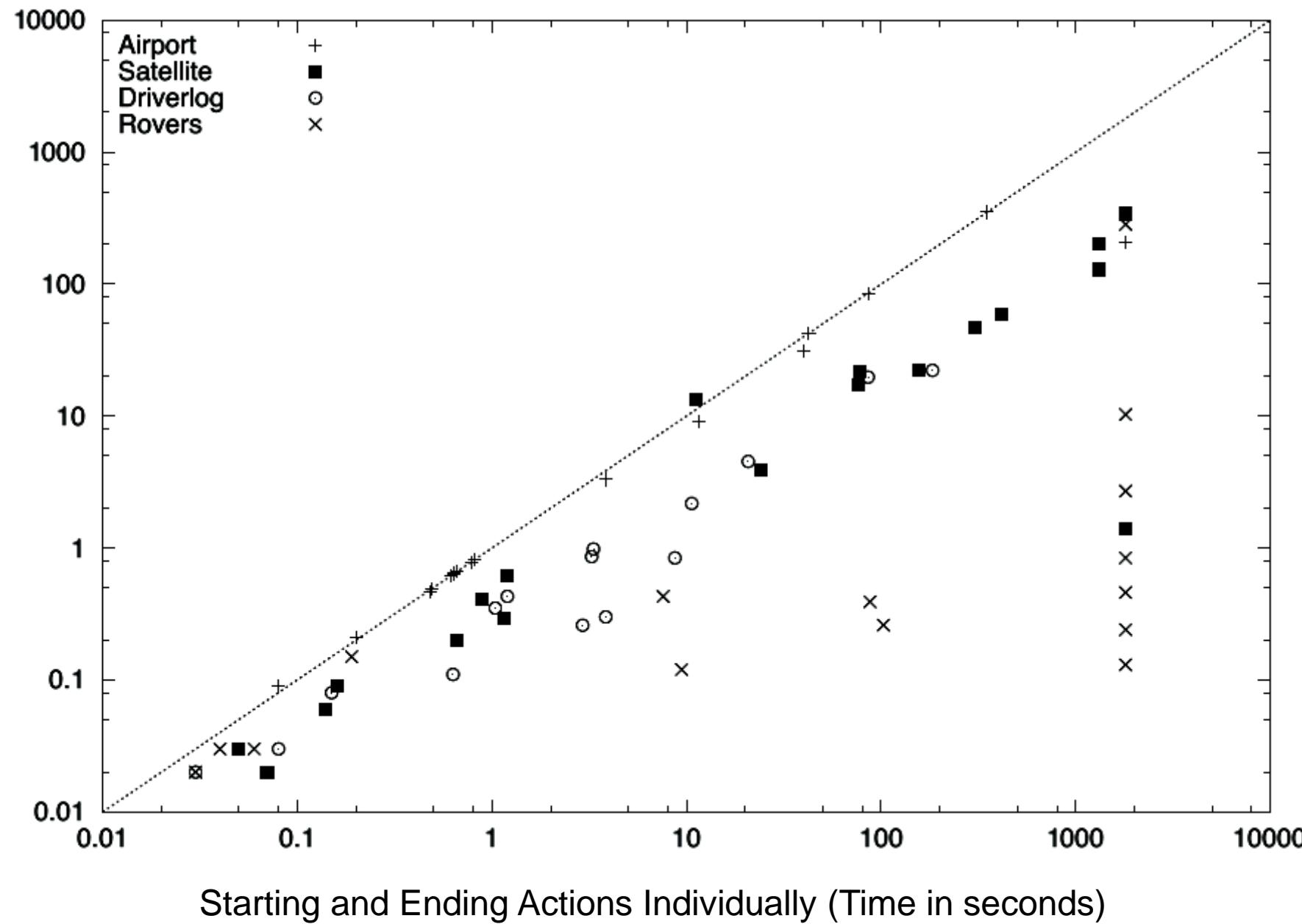


# POPF Does it for us!

- .If we 'automatically' end a compression safe action, that adds a fact p (recall this action does not delete anything)?
  - Only future actions that use or delete p must come after the action.
  - POPF will do this anyway



# So what?

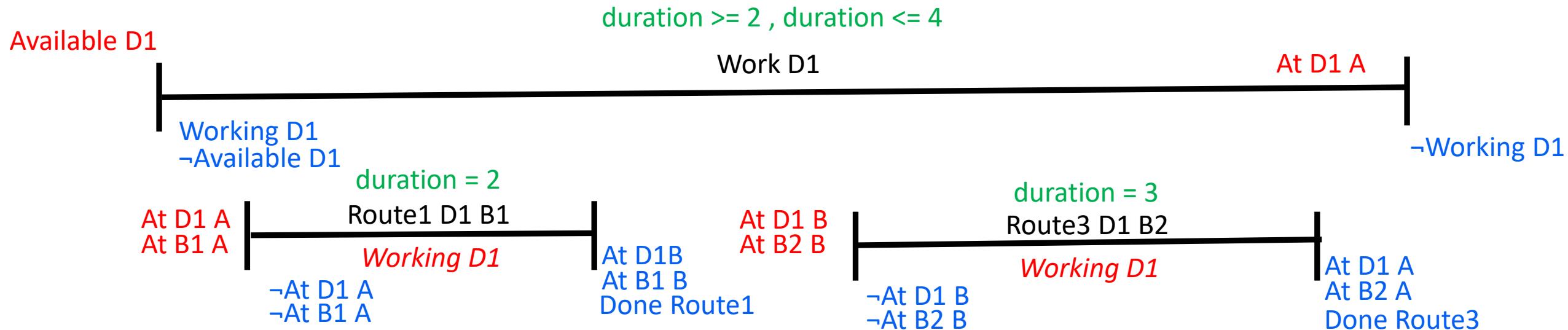


# Planning with Deadlines

# Public Transport Example

- Drivers have working hours;
- Bus routes have fixed durations and start and end locations.
- Goals are that each bus route is done.
- The routes have timetables that they must follow.

# Temporal Planning: Public Transport



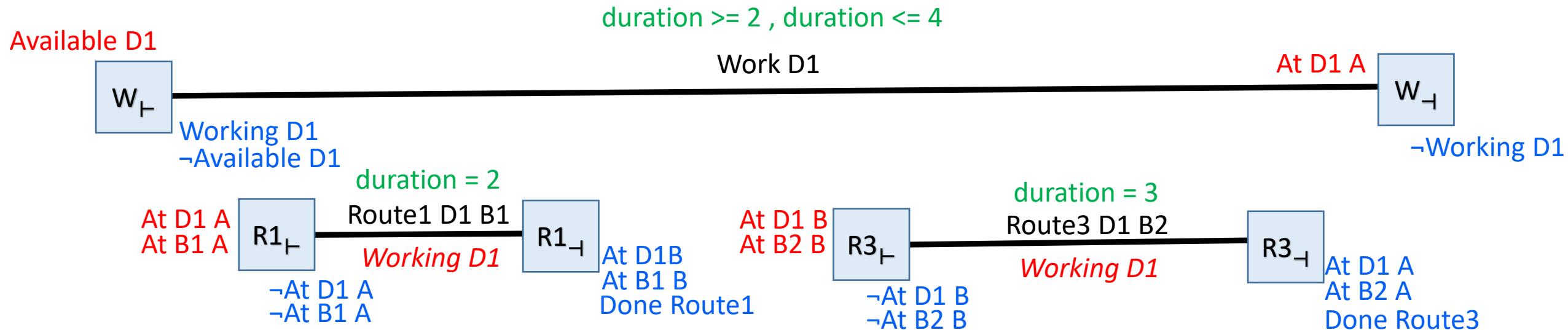
Actions have:

- Conditions and Effects at the start and at the end;
- Invariant/overall conditions;
- Durations constraints:  
 $(= ?\text{duration} 4)$   
 $(\text{and } (\geq ?\text{duration} 2) (\leq ?\text{duration} 4))$

"Planning with Problems Requiring Temporal Coordination." A. I. Coles, M. Fox, D. Long, and A. J. Smith. AAAI 2008.

"Managing concurrency in temporal planning using planner-scheduler interaction." A. I. Coles, M. Fox, K. Halsey, D. Long, and A. J. Smith. Artificial Intelligence. 173 (1) (2009).

# Planning with Snap Actions: Crikey/Crikey3



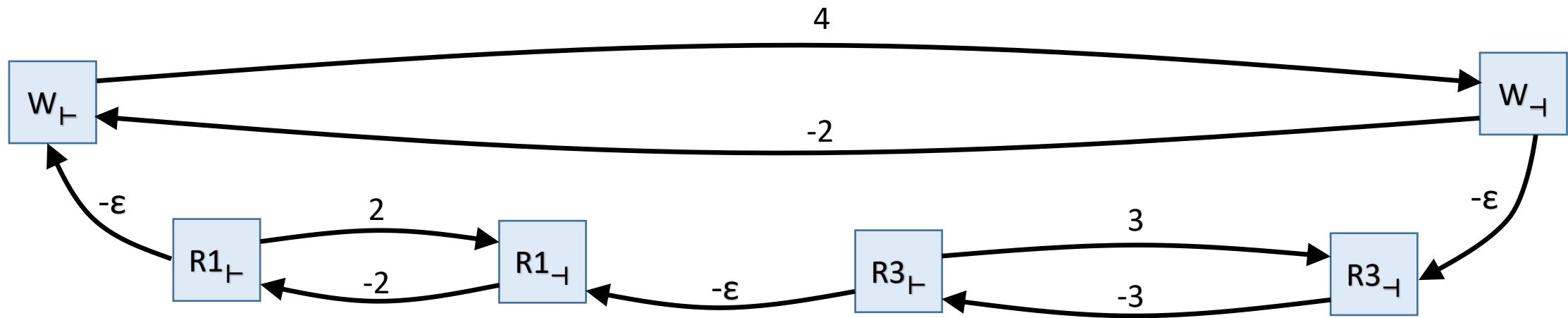
Three Challenges:

- Make sure ends can't be applied unless starts have.
- Overall Conditions.
- Duration constraints.

"Planning with Problems Requiring Temporal Coordination." A. I. Coles, M. Fox, D. Long, and A. J. Smith. AAAI 2008.

"Managing concurrency in temporal planning using planner-scheduler interaction." A. I. Coles, M. Fox, K. Halsey, D. Long, and A. J. Smith. Artificial Intelligence. 173 (1) (2009).

# Planning with Snap Actions: Crikey/Crikey3



Constraints:

$$W_- - W_+ \geq 2$$

$$W_- - W_+ \leq 4$$

$$R1_+ \geq W_+ + \varepsilon$$

$$R1_- - R1_+ = 2$$

$$R3_+ \geq R1_+ + \varepsilon$$

$$R3_- - R3_+ = 3$$

$$W_+ \geq R3_+ + \varepsilon$$

"Planning with Problems Requiring Temporal Coordination." A. I. Coles, M. Fox, D. Long, and A. J. Smith. AAAI 2008.

"Managing concurrency in temporal planning using planner-scheduler interaction." A. I. Coles, M. Fox, K. Halsey, D. Long, and A. J. Smith. Artificial Intelligence. 173 (1) 2009.

# Timed Initial Literals

- Introduced in PDDL 2.2 (IPC 2004);
- Allow us to model facts that become true, or false, at a specific time.
- Can use them to model deadlines or time windows.
- Cannot be done directly, but we can achieve this by adding more facts to the domain.

# Modelling Deadlines using TILs

- Make sure the action achieving the desired fact has a condition that ensures it takes place before the deadline (over all or at start/end).
- Make that fact true in the initial state.
- And a TIL to delete it at the deadline.
- Note that we could have multiple deadlines for different objects.

```
(:durative-action unload-truck  
:parameters (?p - obj ?t- truck ?l- location)  
:duration (= ?duration 2)  
:condition (and (over all (at ?t ?l))  
              (at start (in ?p ?t)))  
             (at end (can-deliver ?p)))  
:effect (and (at start (not (in ?p ?t)))  
           (at end (at ?p ?l))))  
Init:  
(can-deliver package1)  
(at 9 (not (can-deliver package1)))  
(can-deliver package2)  
(at 11 (not (can-deliver package2)))
```

# Modelling Time Windows Using TILs

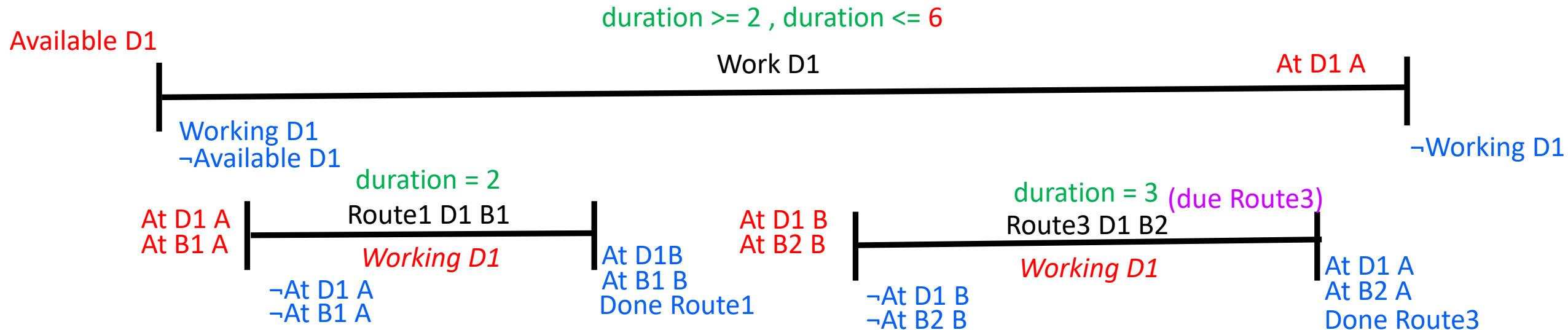
- Make sure the action achieving the desired fact has a condition that ensures it takes place during the window (over all or at start/end). POPF/OPTIC will generally work better if you use over all where possible.
- Have a TIL to add that fact at the starting point for the window.
- And one to delete it when the window ends.
- Note that we could have multiple windows for the same fact by adding further TILs to the initial state.

```
(:durative-action bus-route
:parameters (?d – driver ?r – route ?b – bus
             ?from ?to – loc)
:duration (= ?duration (route-duration ?r))
:condition (and (at start (route ?r ?from ?to))
                  (at start (at ?d ?from))
                  (at start (at ?b ?from))
                  (over all (working ?d))
                  (at end (due ?r)))
:effect (and (at start (not (at ?d ?from)))
               (at start (not (at ?b ?from)))
               (at end (at ?d ?to))
               (at end (at ?b ?to))
               (at end (done ?r)))
)
init:
(at 3.75 (due route3))
(at 4 (not (due route3)))
```

# Reasoning with TILs in Forward Search

- Order the TILs chronologically;
- At each state we have a choice:
  - Apply an action that is applicable in that state;
  - Apply the next Available TIL.
- This allows us to leave the choice to search about whether the TIL will appear before or after a given action.
- POPF has some advantages in this situation:
  - Only necessary orderings are enforced.

# Temporal Planning: Public Transport



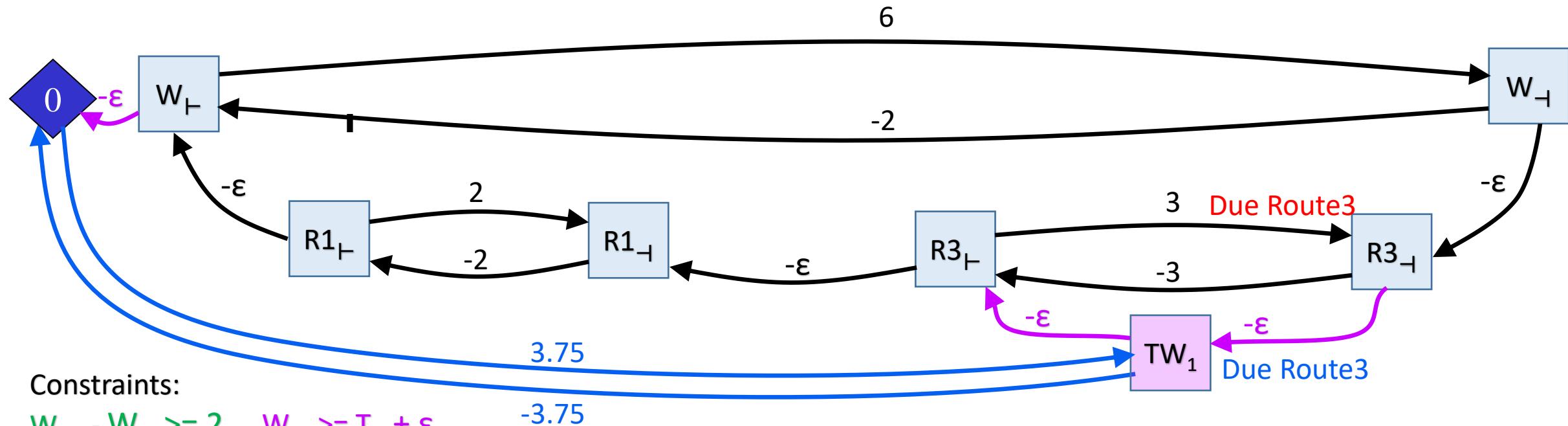
Actions have:

- Conditions and Effects at the start and at the end;
- Invariant/overall conditions;
- Durations constraints:  
 $(= ?\text{duration} 4)$   
 $(\text{and } (>=?\text{duration} 2) (<=?\text{duration} 6))$
- We can also have windows of opportunity: (at 3.75 (due Route3)) (at 4 (not (due Route3)))

"Planning with Problems Requiring Temporal Coordination." A. I. Coles, M. Fox, D. Long, and A. J. Smith. AAAI 2008.

"Managing concurrency in temporal planning using planner-scheduler interaction." A. I. Coles, M. Fox, K. Halsey, D. Long, and A. J. Smith. Artificial Intelligence. 173 (1) (2009).

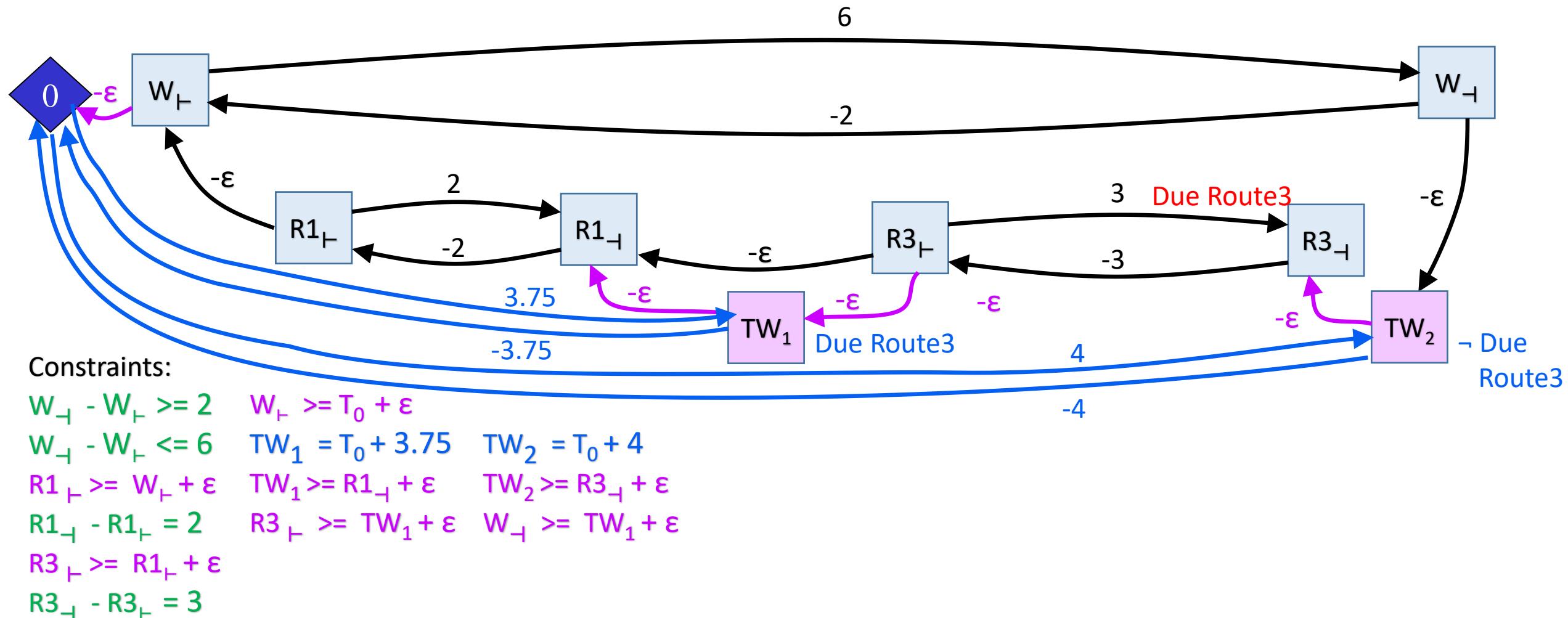
# Planning with Time Windows: Crikey/Crikey3



"Planning with Problems Requiring Temporal Coordination." A. I. Coles, M. Fox, D. Long, and A. J. Smith. AAAI 2008.

"Managing concurrency in temporal planning using planner-scheduler interaction." A. I. Coles, M. Fox, K. Halsey, D. Long, and A. J. Smith. Artificial Intelligence. 173 (1) 2009.

# Planning with Time Windows: Crikey/Crikey3



"Planning with Problems Requiring Temporal Coordination." A. I. Coles, M. Fox, D. Long, and A. J. Smith. AAAI 2008.

"Managing concurrency in temporal planning using planner-scheduler interaction." A. I. Coles, M. Fox, K. Halsey, D. Long, and A. J. Smith. Artificial Intelligence. 173 (1) 2009.



# Classical Planning Improving Heuristic Search

6CCS3AIP – Artificial Intelligence Planning

Dr Tommy Thompson



# Classical Planning: Material Overview

- Classical Planning: The fundamentals.
- Non-Forward Search
  - Graphplan
- Improving Search
  - Heuristic Design (RPG/LAMA)
  - Dual Openlist Search
- SAT Planning
- POP Planning
- HTN Planning
- Optimal Planning
- Pattern Databases
- Planning Under Uncertainty



# Fundamentals

- **Definition of the problem – PDDL**
- **Forward Search**
  - Breadth/Depth First
- **Introducing Heuristics**
  - Best First Search/A\*

## Constraints:

Fuel

Truck capacity

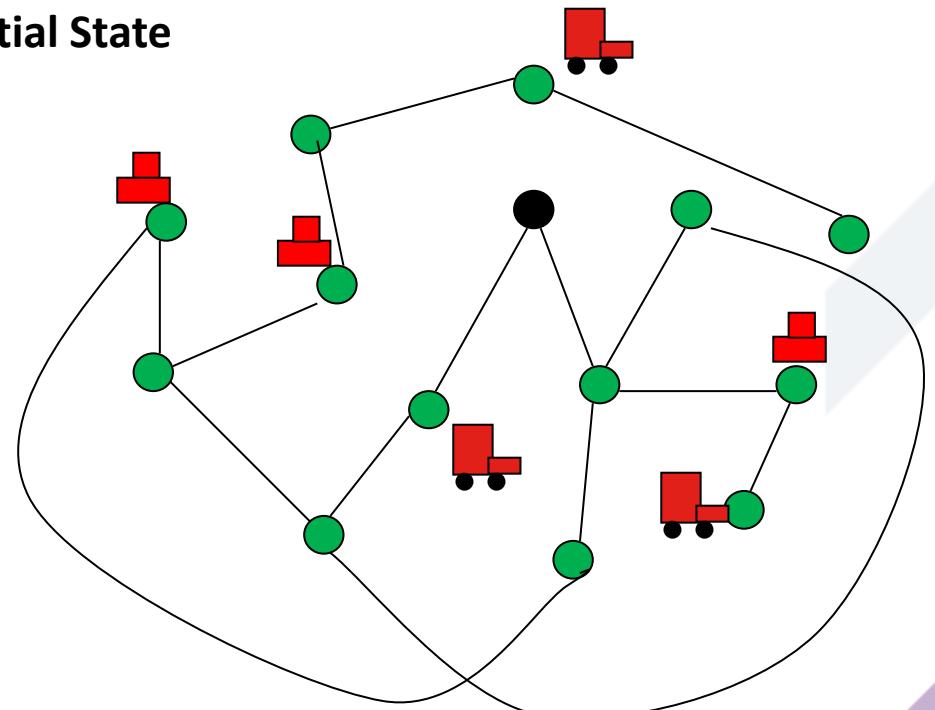
Number of drivers

...

...

-  Cargos
-  Trucks
-  Destination point

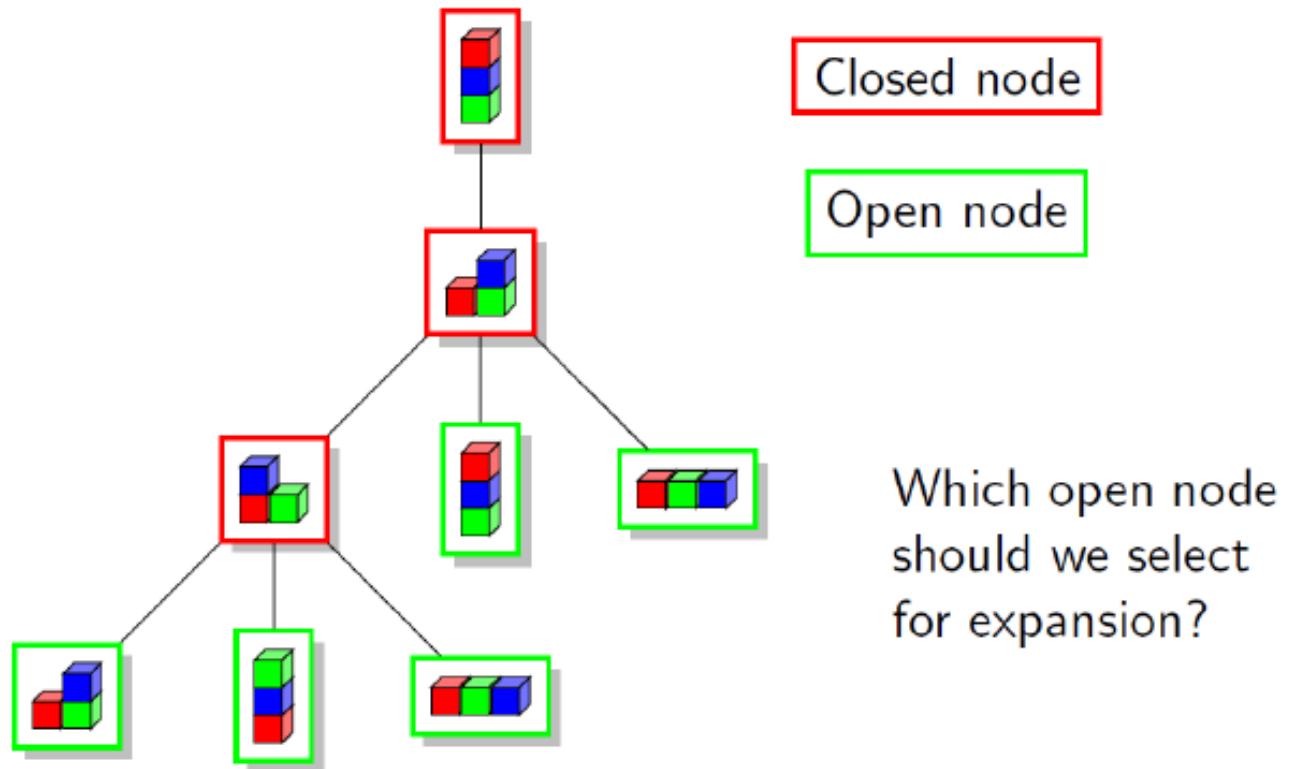
Initial State



**Goal:** all cargos at a destination point

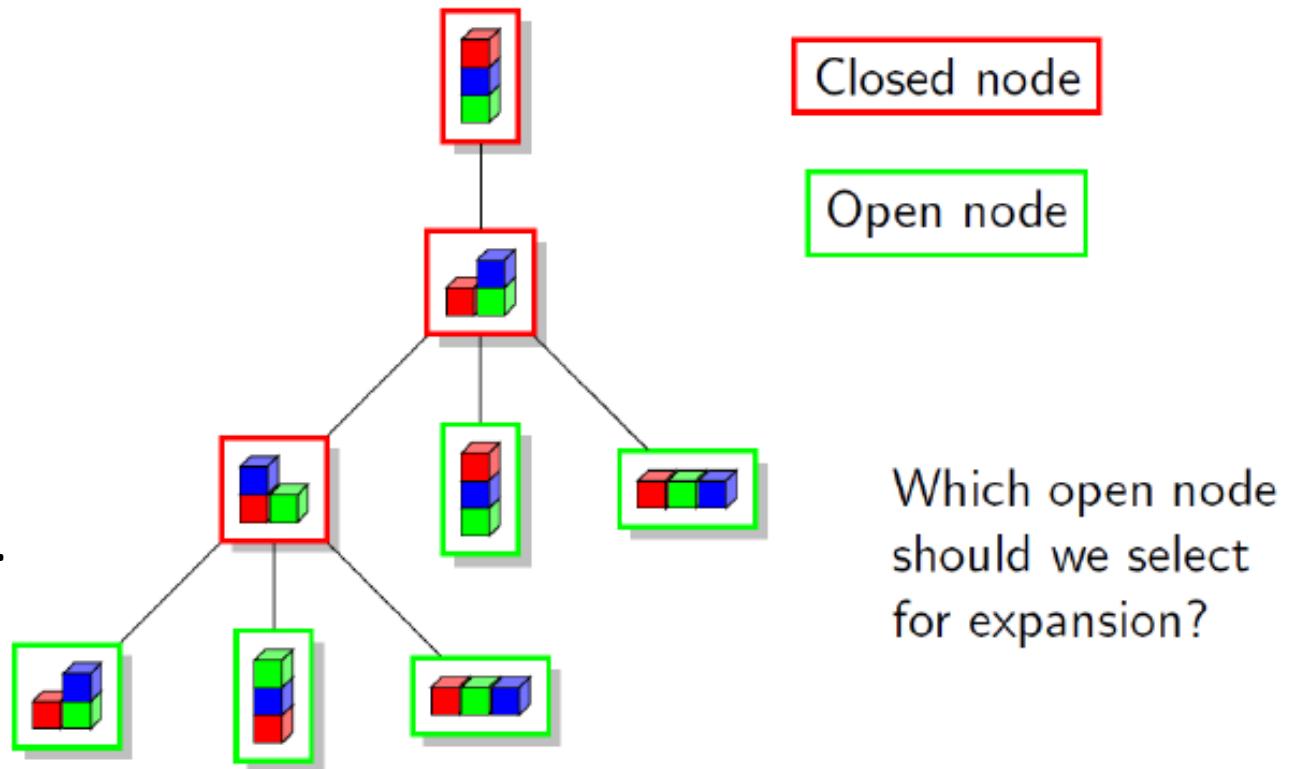
# Fundamentals

- **Definition of the problem – PDDL**
- **Forward Search**
  - Breadth/Depth First
- **Introducing Heuristics**
  - Best First Search/A\*



# Heuristic Search Planning

- **Requires**
  - $h$ : estimate of  $s_{current}$  to goal
  - $g$ : cost of path from  $s_{init}$  to  $s_{current}$
- **Method**
  - Prioritise open nodes with heuristic
  - Different algorithms exploit heuristic and costs.
- **Examples**
  - **Uniform Cost Search**: Expand node with minimum  $g$
  - **Greedy Best-First Search**: Expand node with minimum  $h$ .
  - **A\***: Expand node with minimum  $(g+h)$



# Building Heuristics

- **Admissible**

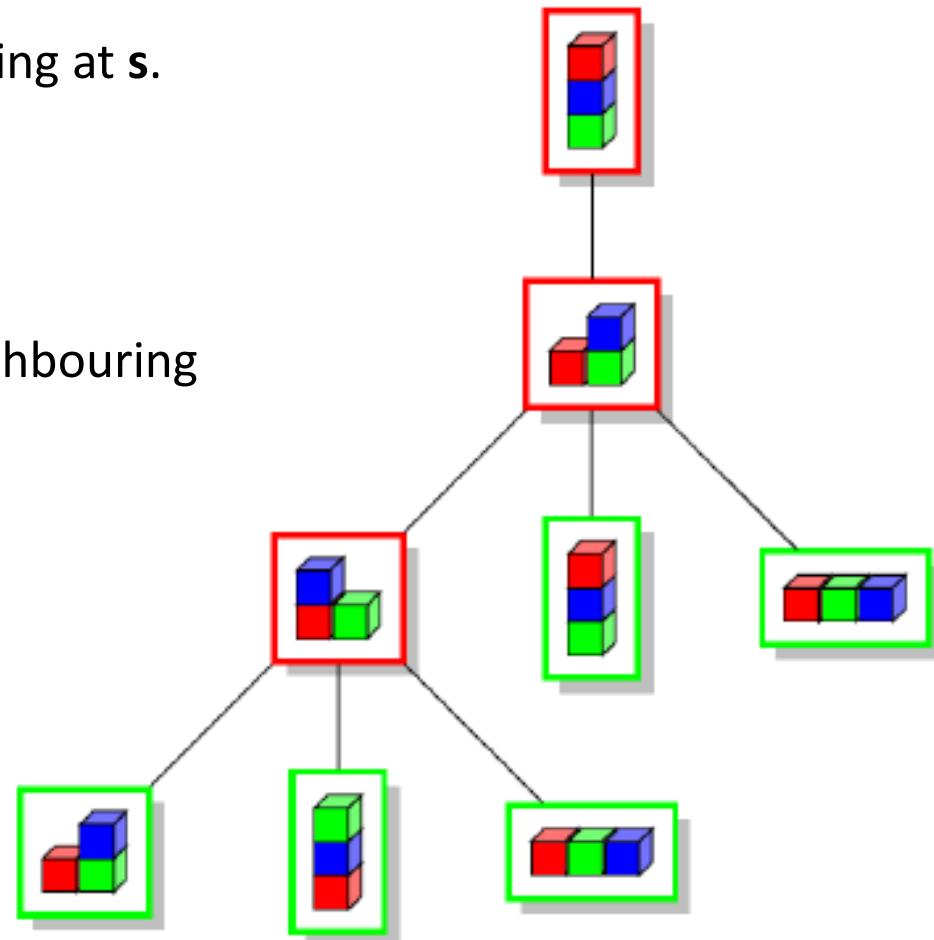
- If  $h(s)$  is a lower bound of the cost of all goal-reaching plans starting at  $s$ .
- i.e. Heuristic value is less than or equal to the actual cost.

- **Consistent**

- $h(s)$  is always less than or equal to estimated value from any neighbouring vertex, plus the cost of reaching it.
- i.e. if we travel from  $s \rightarrow s'$  using action  $a$ 
  - $h(s') - h(s) + \text{cost}(a) = 0$

- **Additive**

- $h(s) = h_1(s) + h_2(s)$  where  $\forall s \in S$  admissible.



# Outstanding Problems

- Building Domain-Independent Heuristics
- Improving Search



# Improving Heuristic Search

- Problem Relaxation and Relaxed Planning (RPG Heuristic)
- Planning Landmarks
- Landmark Counting (and LAMA)



# Classical Planning Improving Heuristic Search

6CCS3AIP – Artificial Intelligence Planning

Dr Tommy Thompson



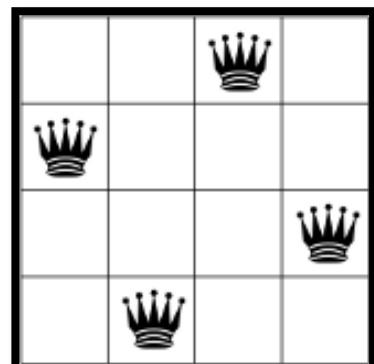
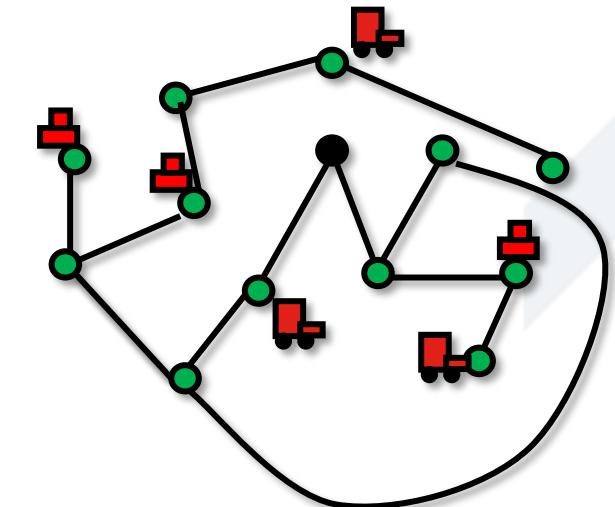
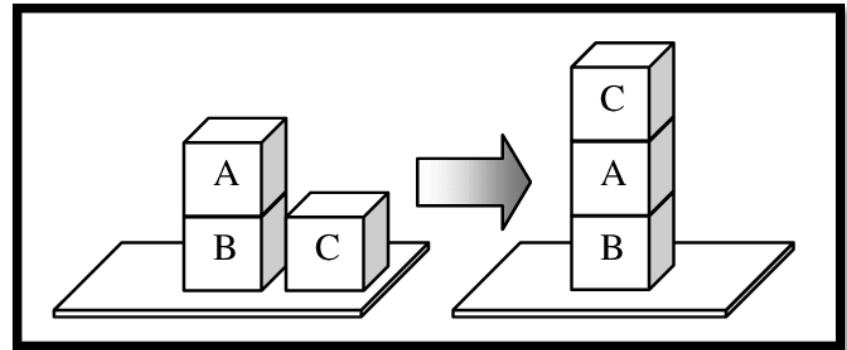
# Classical Planning Relaxed Planning & RPG Heuristic

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



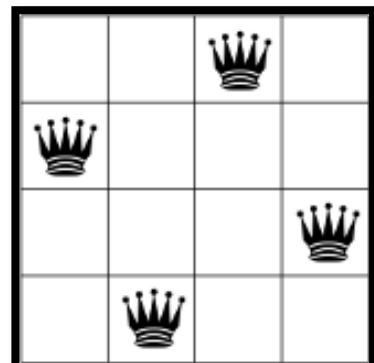
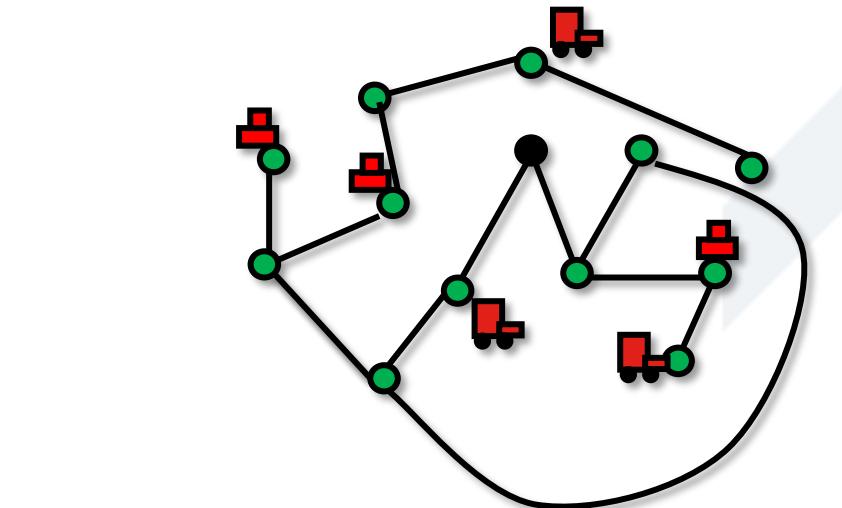
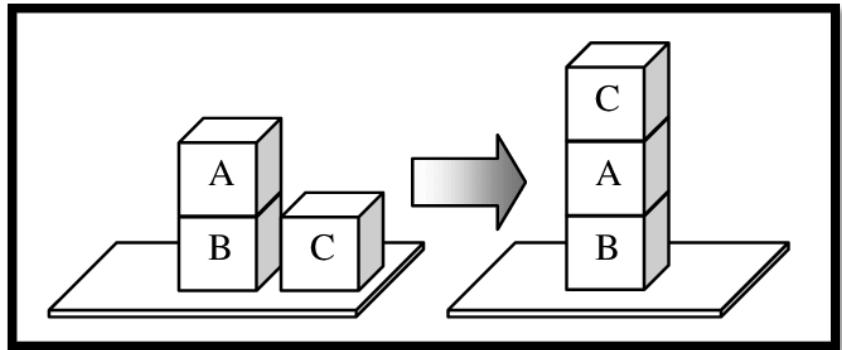
# Domain Independent Heuristics

- Building heuristics that can adapt to different domains/problems.
- Not reliant on specific information about the problem.
- We analyse aspects of the search and planning process to find potential heuristics.



# Domain Independent Heuristics

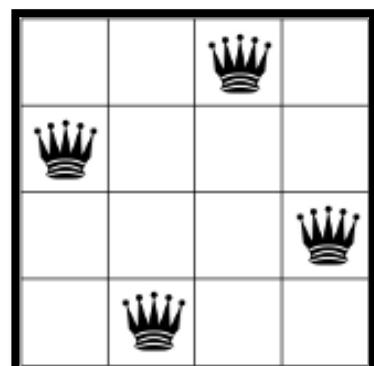
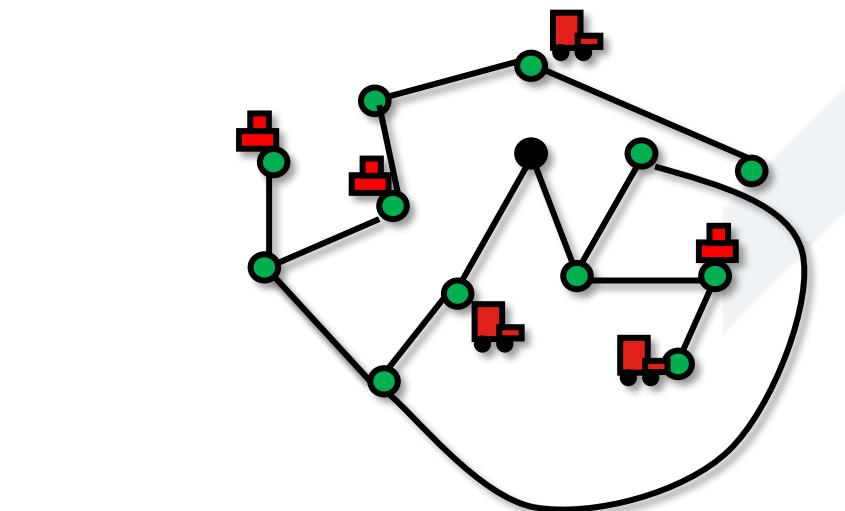
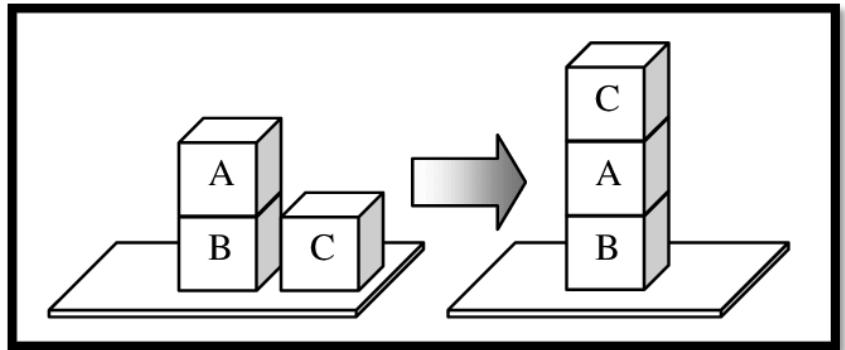
- STRIPS planning (Fikes & Nilsson, 1971) uses number of goals satisfied.
  - Number of variables from  $s_{goal}$  satisfied in  $s_{current}$
- Goal-Counting Heuristic: More satisfied goals, then closer to the goal?
- Problems
  - Uninformative about the problem.
  - Ignores a lot of the problem structure.



Fikes, R.E. and Nilsson, N.J., 1971. STRIPS: A new approach to the application of theorem proving to problem solving.  
Artificial intelligence, 2(3-4), pp.189-208.

# Relaxed Planning

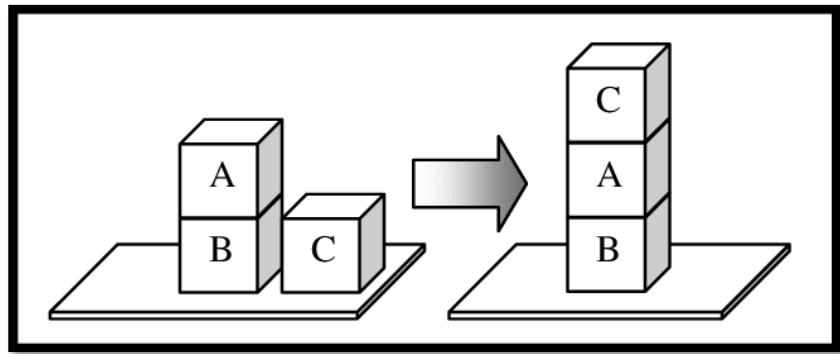
- Establishing valid solutions on planning problems is hard.
  - Calculating whether a planning instance has any valid solutions exists within PSPACE-complete and optimal planning is NP-Hard to compute (Bylander, 1994).
  - Need a polynomial time heuristic – given we're going to use it a lot!
- Relaxed Plan Heuristics reliant on solving the 'relaxed problem'.
  - Estimate cost using 'simpler' version of existing problem.
  - Solving the relaxed problem provides a lower-bound on optimal plan length.



Bylander, T., 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2), pp.165-204.

# Delete Relaxation

- For any given action, we have two types of effects:
- Add Effects
  - New information **added** to the world state.
- Delete Effects
  - Information **removed** from the world state.
- Delete relaxation heuristics ignore delete effects when the state-transition model is applied.



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

$$s \in S, a \in A$$

$$\gamma(s, a) \rightarrow s'$$

# Delete Relaxation

- **Delete Relaxation**

- Estimate cost to the goal by **removing negative effects** of actions.
- i.e. any PDDL effect that removes a fact is no longer considered.

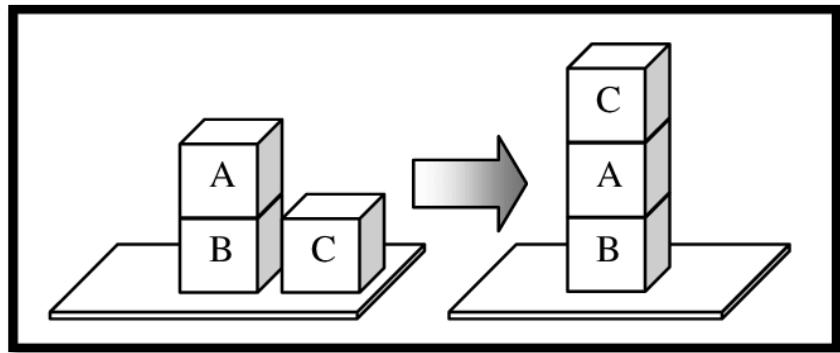
- **Example: FreeCell Solitaire**

- Free cells, tableau positions **remain available after moving cards onto them**.
- Cards **remain movable** and **remain valid targets for other cards** after moving cards on top of them.



# Relaxed Planning

- Relaxed planning used to denote planning using delete relaxation.
  - Resulting in relaxed plans.
- How does this work as a heuristic?
  - For evaluating current state  $s_{curr}$ , calculate relaxed plan.
  - $h(S_{curr}) = \text{cost } (\# \text{actions}) \text{ of the relaxed plan.}$



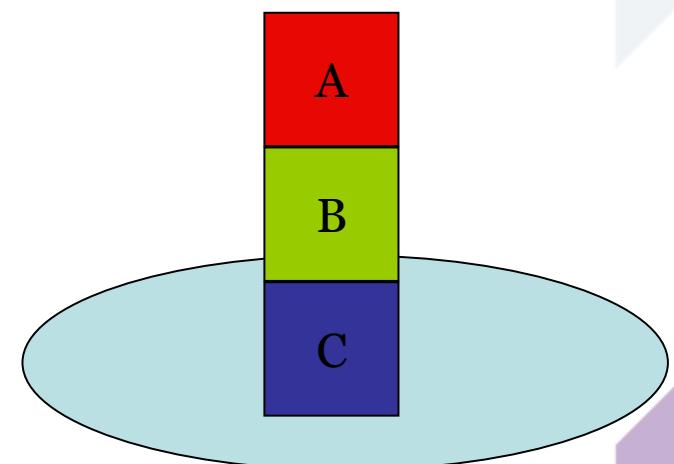
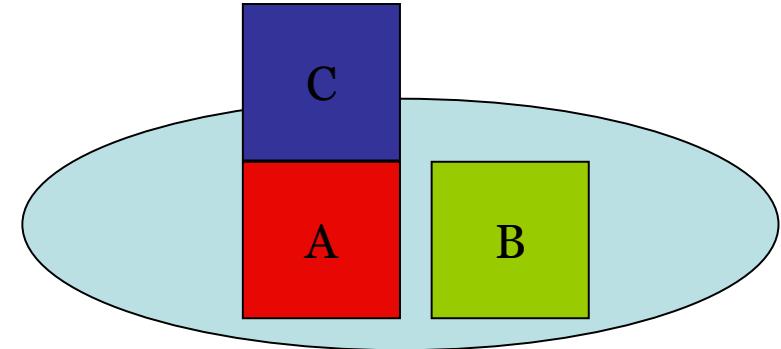
$$\Sigma = (S, A, \gamma)$$

$$s \in S, a \in A$$

$$\gamma(s, a) \rightarrow s'$$

# Relaxed Planning Graph Heuristic

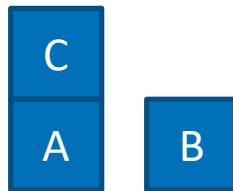
- Relaxed Planning Graph (RPG) heuristic creates a simple layered approach to exploring the state space.
- Inspired by the GraphPlan system
  - GraphPlan covered later in this module.
  - For reference, see (Blum & Furst, 1995)
- The relaxed solution plan  $P' = \langle O_0, O_1, \dots, O_{m-1} \rangle$ 
  - Where each  $O_i$  is the set of actions selected in parallel at time step  $i$ , and  $m$  is the number of the first fact layer containing all goals
  - $h(s) := \sum |O_i|$



Blum, A. L., & Furst, M. L. (1995). Fast planning through planning graph analysis.

In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI95), pp. 1636

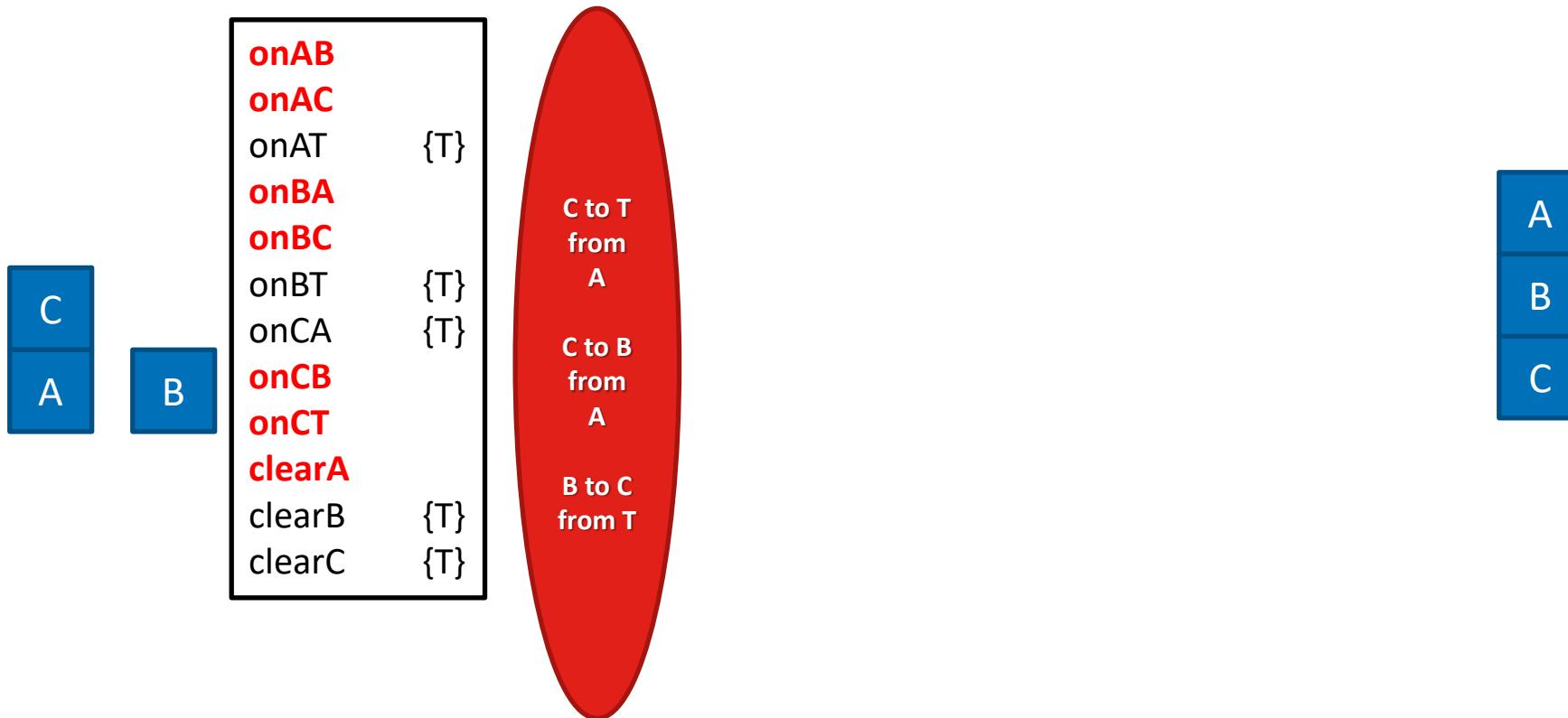
# Building a Relaxed Planning Graph



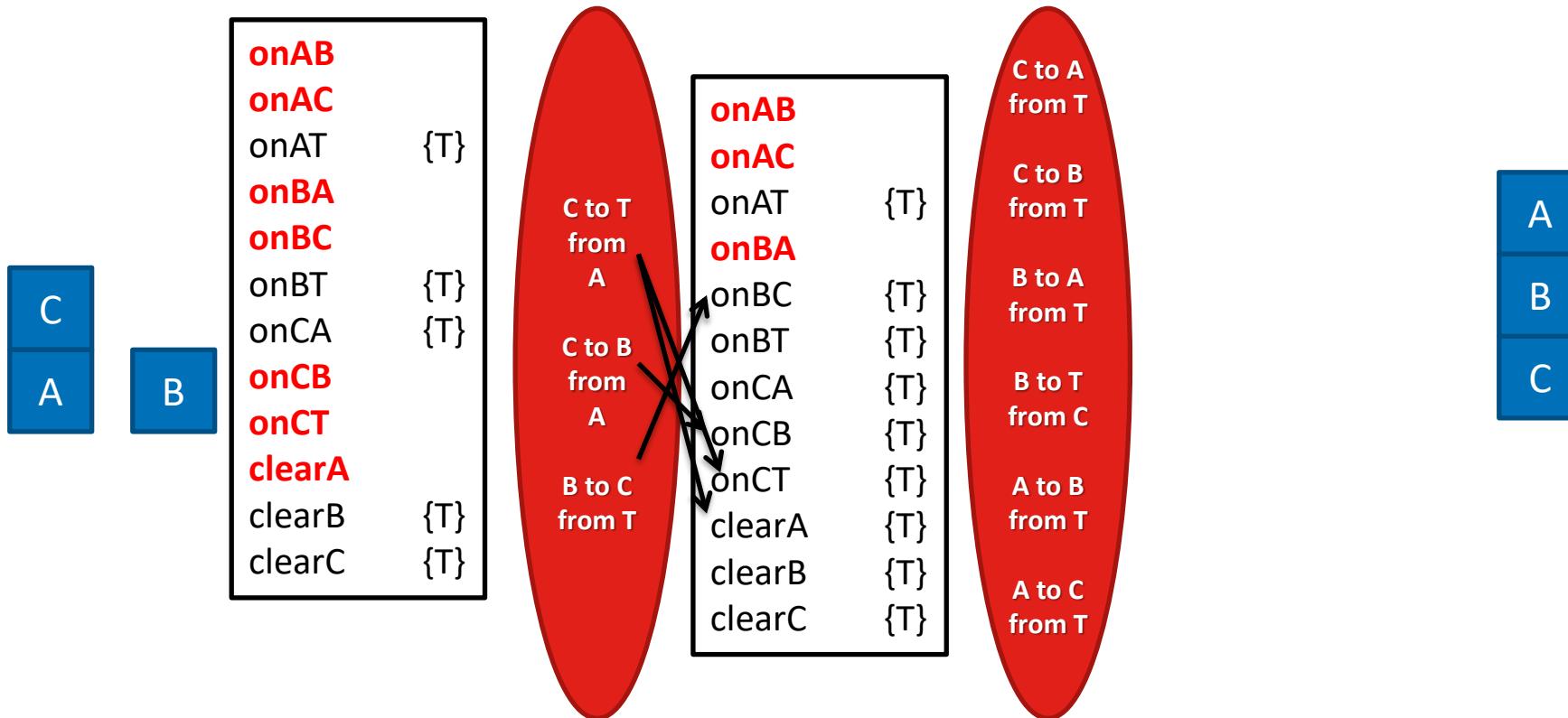
# Building a Relaxed Planning Graph



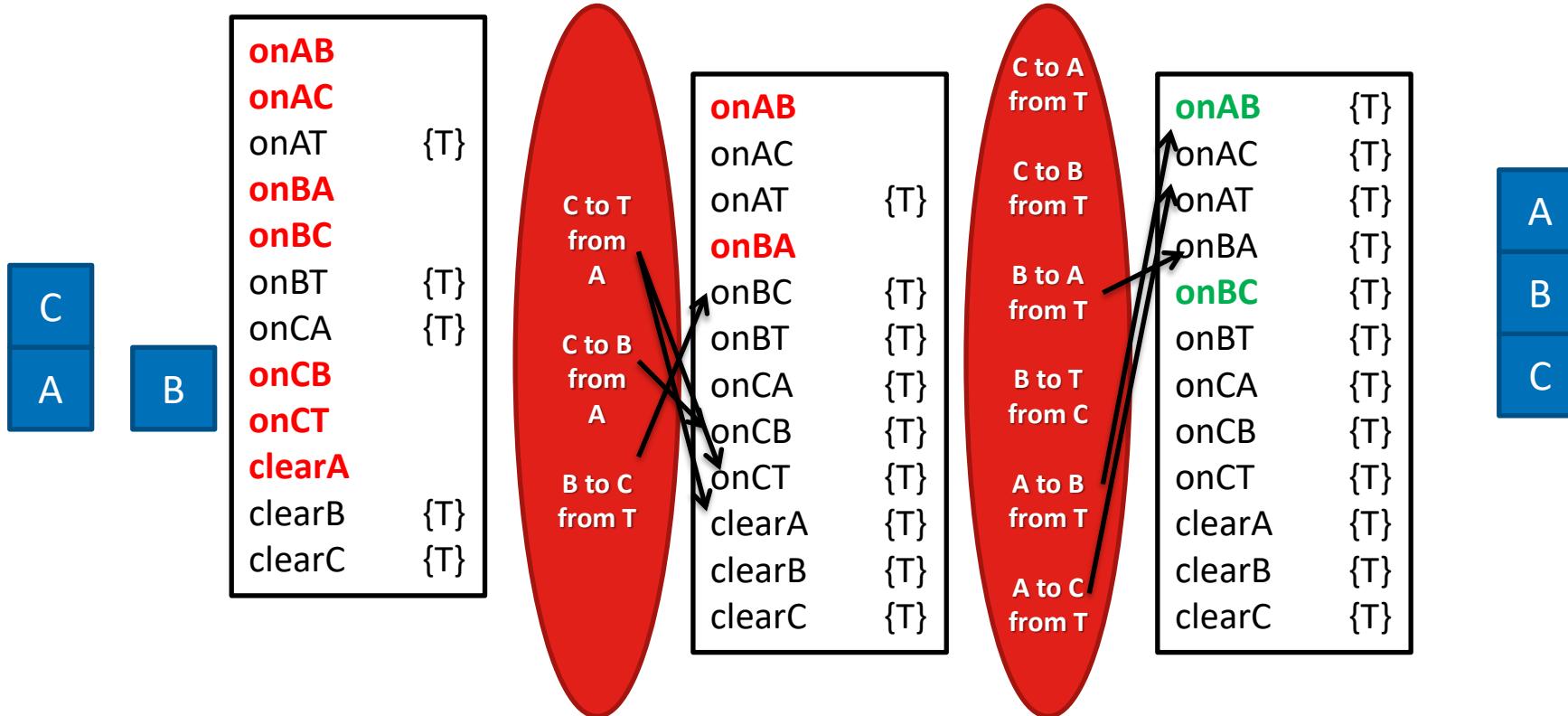
# Building a Relaxed Planning Graph



# Building a Relaxed Planning Graph



# Building a Relaxed Planning Graph



# Extracting a Solution

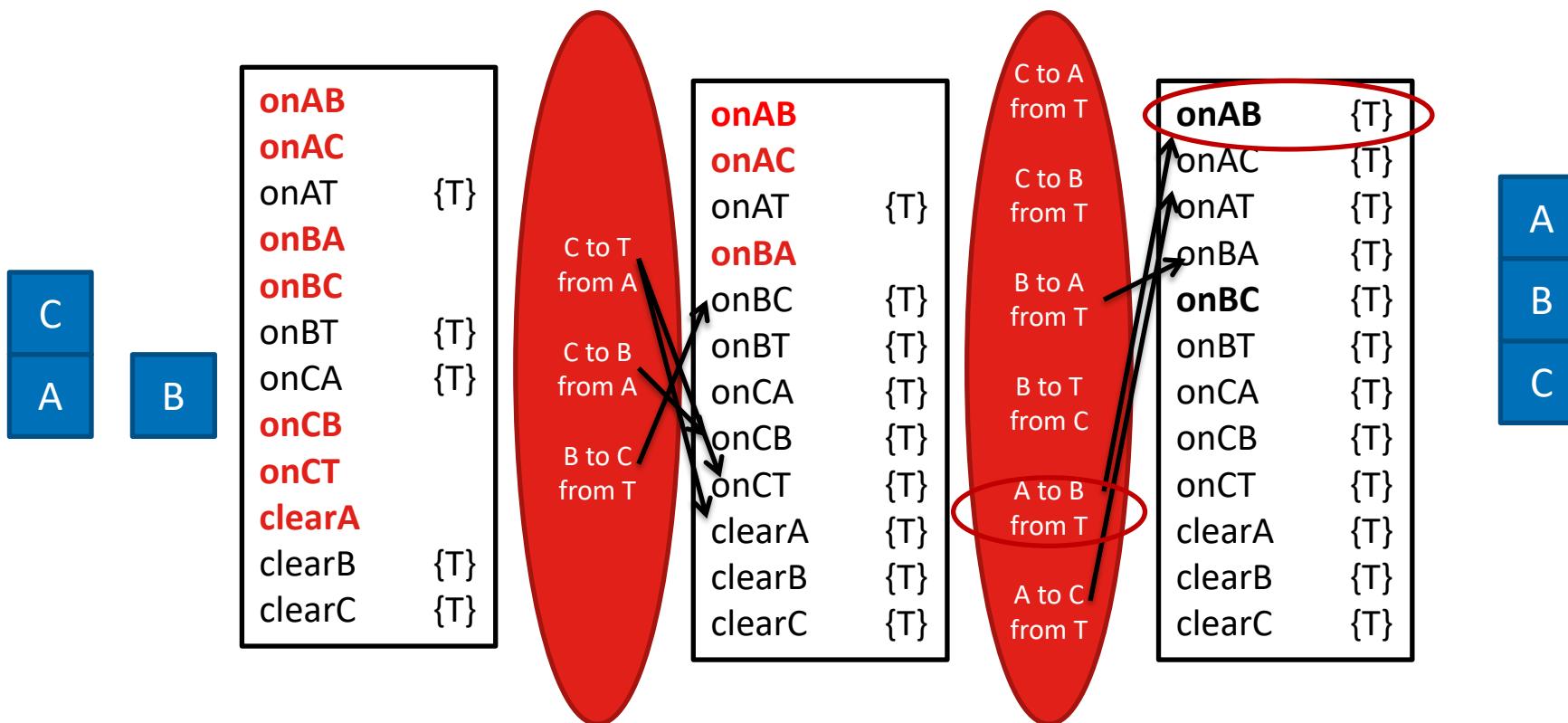
- To get a solution, work backwards through the RPG.
  - At each fact layer  $f(n)$ , we have goals to achieve  $g(n)$ .
- We start with  $g(n)$  containing the problem goals.
  - For each fact in  $g(n)$ :
    - If it was in  $f(n-1)$ , add it to  $g(n-1)$
    - Otherwise, choose an action from  $a(n)$ , and add its preconditions to  $g(n-1)$
- Stop when at  $g(0)$ .

onAB	{T}
onAC	{T}
onAT	{T}
onBA	{T}
onBC	{T}
onBT	{T}
onCA	{T}
onCB	{T}
onCT	{T}
clearA	{T}
clearB	{T}
clearC	{T}



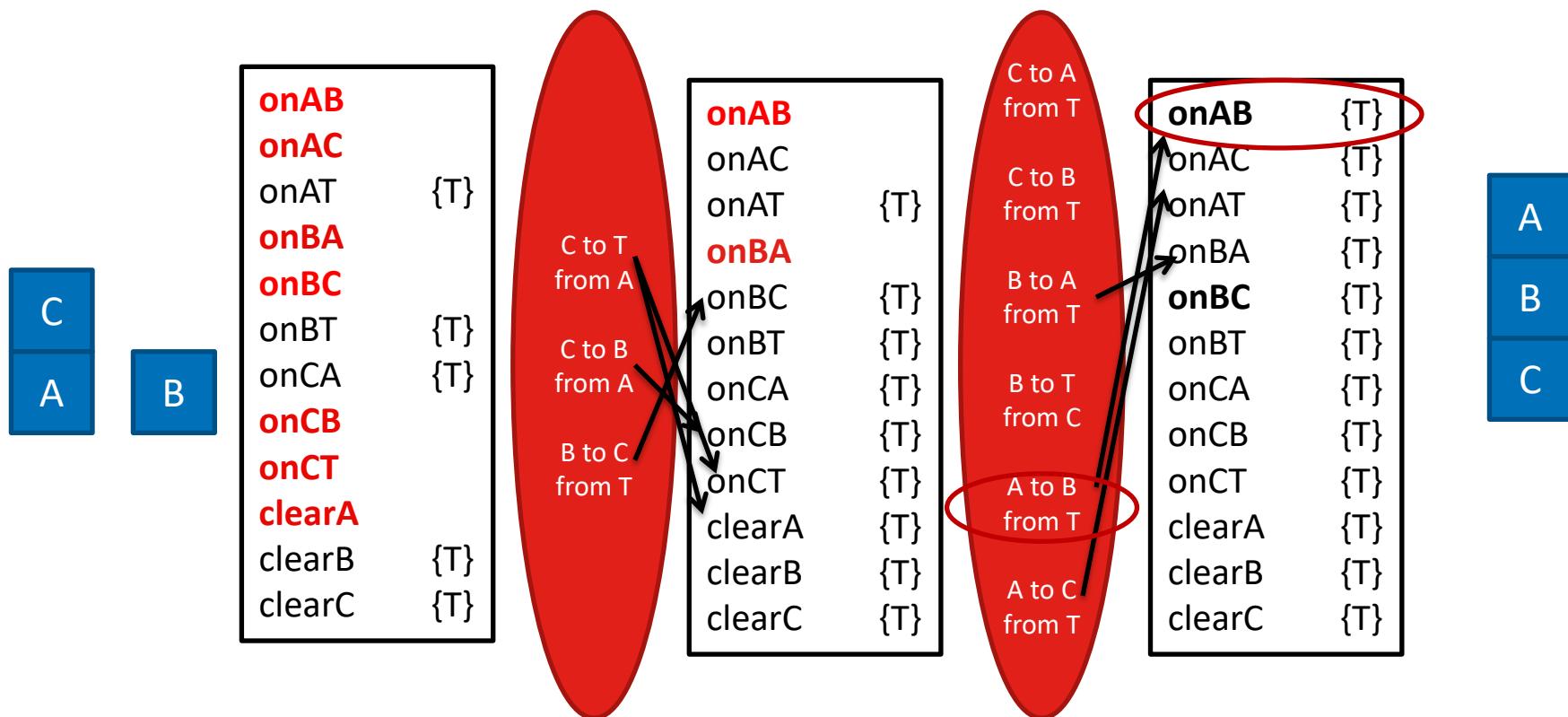
# Extracting a Solution

- $G(n) = \text{onAB}, \text{onBC}$
- $G(n-1) = \text{onBC}$



# Extracting a Solution

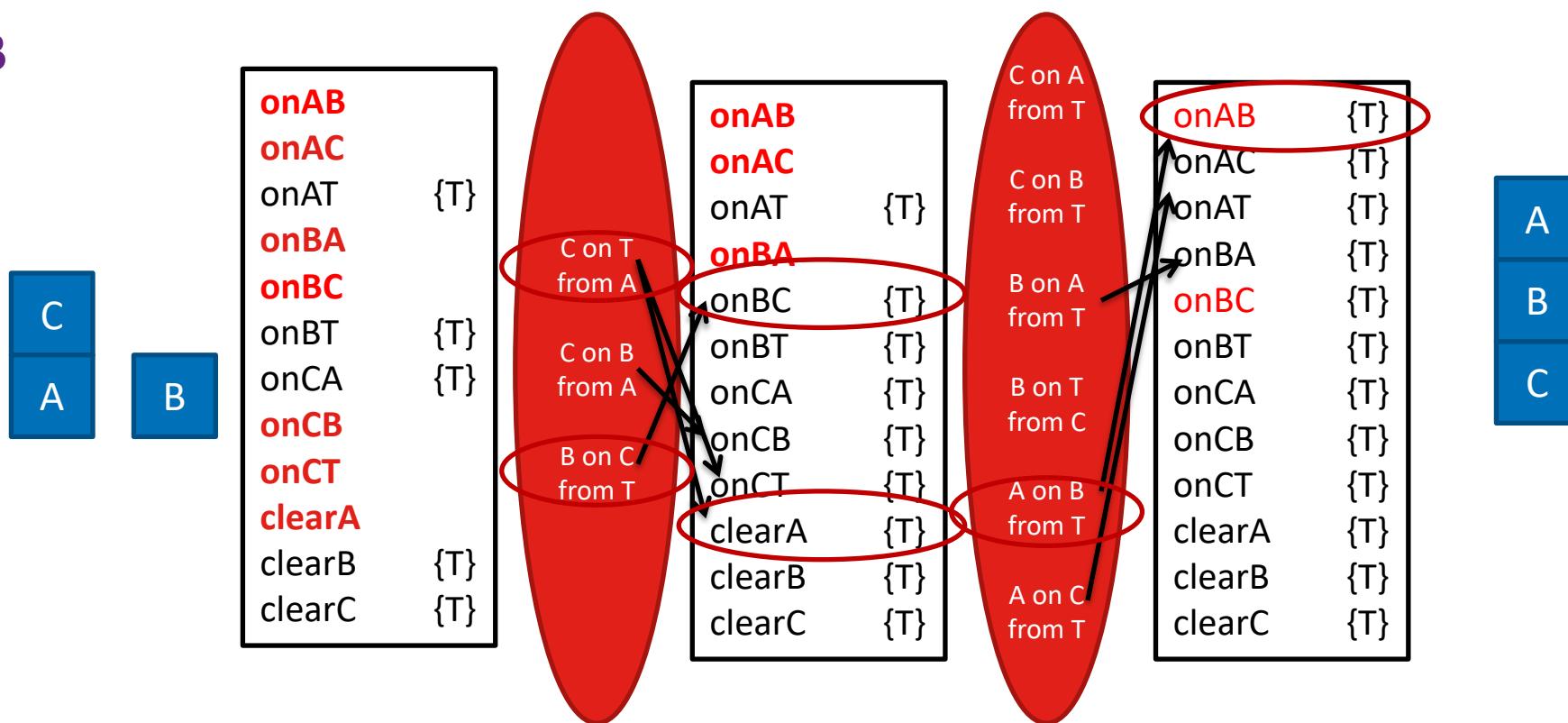
- $G(n-1) = \text{onBC, clearA}$
- $G(n-2) = ???$



# Extracting a Solution

- $G(n-1) = \text{onBC, clearA}$
- $G(n-2) = \text{clearB}$

Final relaxed solution:  
Put C on T from A, Put B to C from T, Put A to B from T



# Summary

- A good way to come up with heuristics: solve the simplified version of the problem.
- Delete Relaxation – discarding all delete effects – provides simplification that can be explored using greedy search in polynomial time.
- Delete Relaxation always creates simplifications: problems always easier to solve than the original.
- RPG layers provide admissible heuristic to relaxed problems.
  - But the relaxed plan length is not.

# Classical Planning Relaxed Planning & RPG Heuristic

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Classical Planning RPG Heuristic in the FF Planner

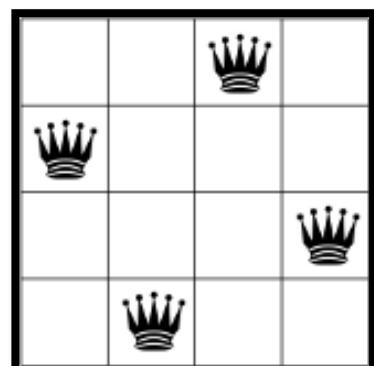
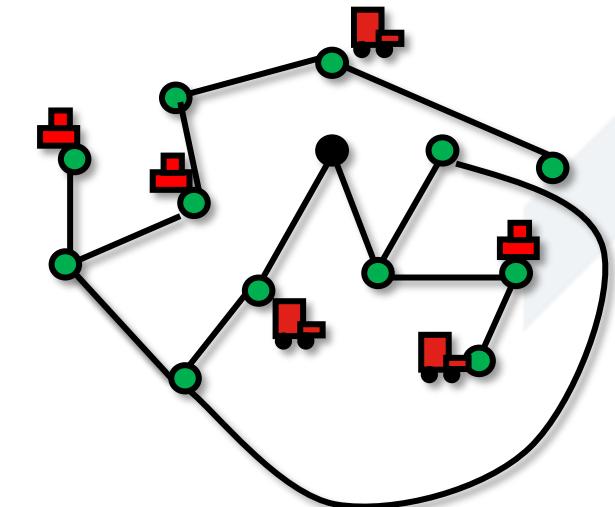
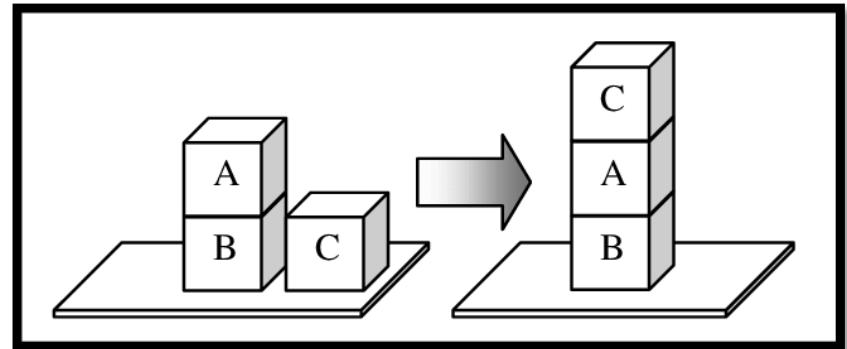
6CCS3AIP – Artificial Intelligence Planning

Dr Tommy Thompson



# Domain Independent Heuristics

- Building heuristics that can adapt to different domains/problems.
- Not reliant on specific information about the problem.
- We analyse aspects of the search and planning process to find potential heuristics.



# Delete Relaxation

- **Delete Relaxation**

- Estimate cost to the goal by **removing negative effects** of actions.
- i.e. any PDDL effect that removes a fact is no longer considered.

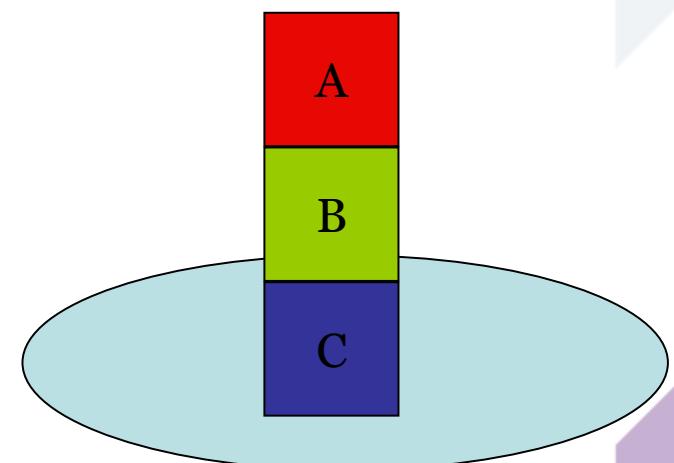
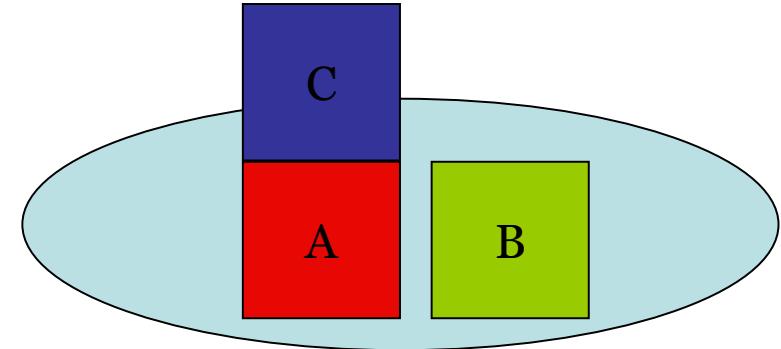
- **Example: FreeCell Solitaire**

- Free cells, tableau positions **remain available after moving cards onto them**.
- Cards **remain movable** and **remain valid targets for other cards** after moving cards on top of them.



# Relaxed Planning Graph Heuristic

- Relaxed Planning Graph (RPG) heuristic creates a simple layered approach to exploring the state space.
- Inspired by the GraphPlan system
  - GraphPlan covered later in this module.
  - For reference, see (Blum & Furst, 1995)
- The relaxed solution plan  $P' = \langle O_0, O_1, \dots, O_{m-1} \rangle$ 
  - Where each  $O_i$  is the set of actions selected in parallel at time step  $i$ , and  $m$  is the number of the first fact layer containing all goals
  - $h(s) := \sum |O_i|$

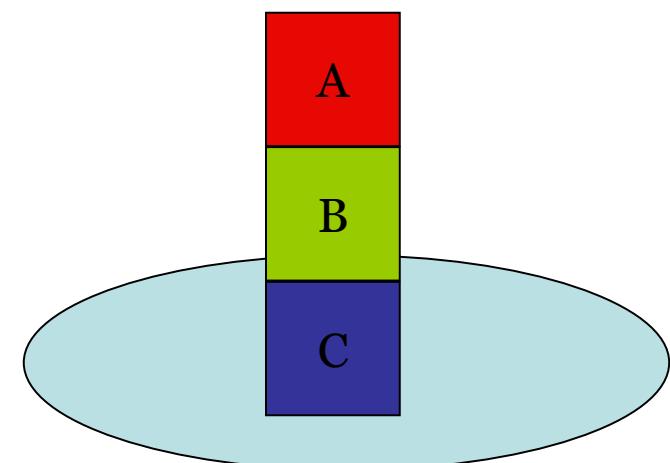
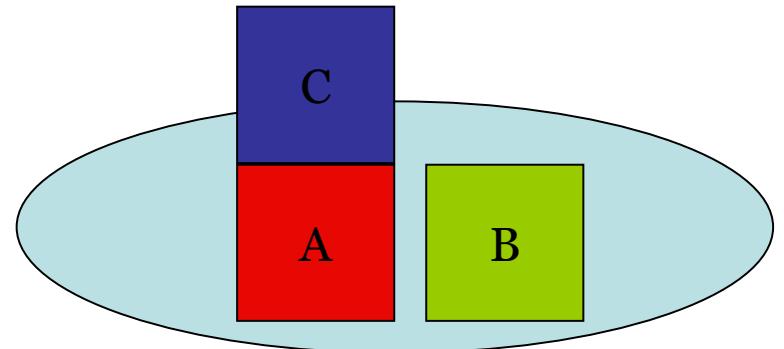


Blum, A. L., & Furst, M. L. (1995). Fast planning through planning graph analysis.

In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI95), pp. 1636

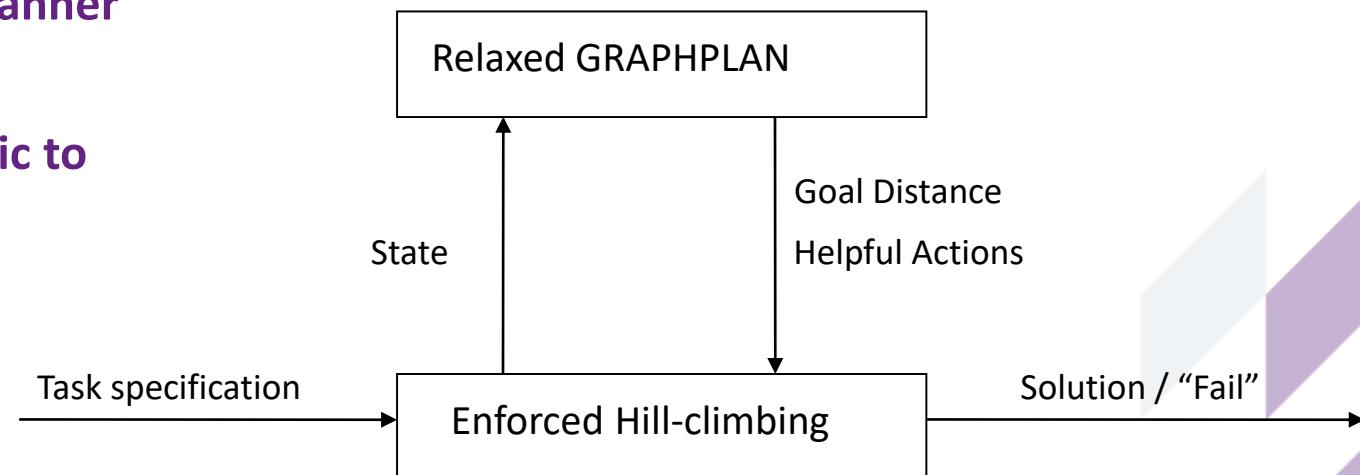
# Domain-Independent Planning: FF

- FF is a forward-chaining heuristic search-based planner
- FF uses the Relaxed Planning Graph (RPG) heuristic to guide search
  - This involves finding a plan from the current state S which achieves the goals G but ignores the delete effects of each action
  - The length of this plan is used as a heuristic value for the state S
- J. Hoffmann and B. Nebel 2001 “The FF Planning System: Fast Plan Generation through Heuristic Search” In Journal of AI Research vol 14.



# Domain-Independent Planning: FF

- FF is a forward-chaining heuristic search-based planner
- FF uses the Relaxed Planning Graph (RPG) heuristic to guide search.
- FF uses both local and systematic search
  - Enforced hill-climbing (EHC)
    - Upon termination, enforced hill-climbing either outputs a solution plan or reports that it has failed
    - If enforced hill-climbing fails, best-first search is invoked
- J. Hoffmann and B. Nebel 2001 “The FF Planning System: Fast Plan Generation through Heuristic Search” In Journal of AI Research vol 14.

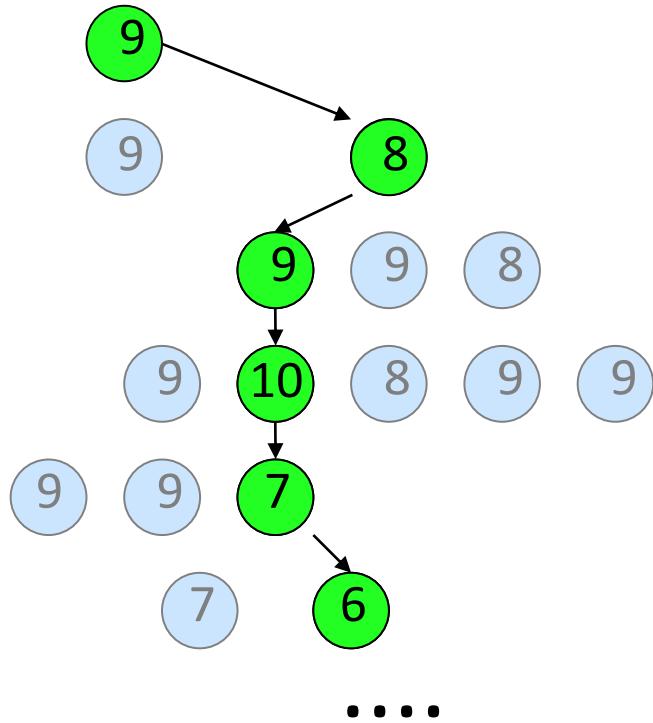
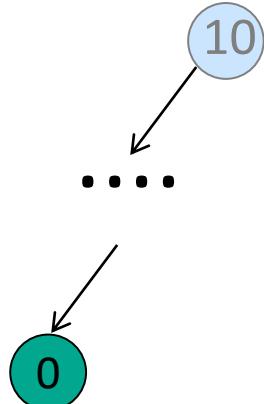


# Enforced Hill Climbing

- Basic idea: lower heuristic = better.
  - Always try and find states with the best heuristic value seen so far:
    - Start with  $\text{best} = h(S_{init})$ , aim to get to  $\text{best} = 0$ .
1. Expand a state  $S$
  2. If we have a successor state  $S'$  with  $h(S') < \text{best}$ :
    - $S = S'$
    - return to Step 1;
  3. If no state is found
    - Breadth-first search until one is found;
    - return to Step 1;

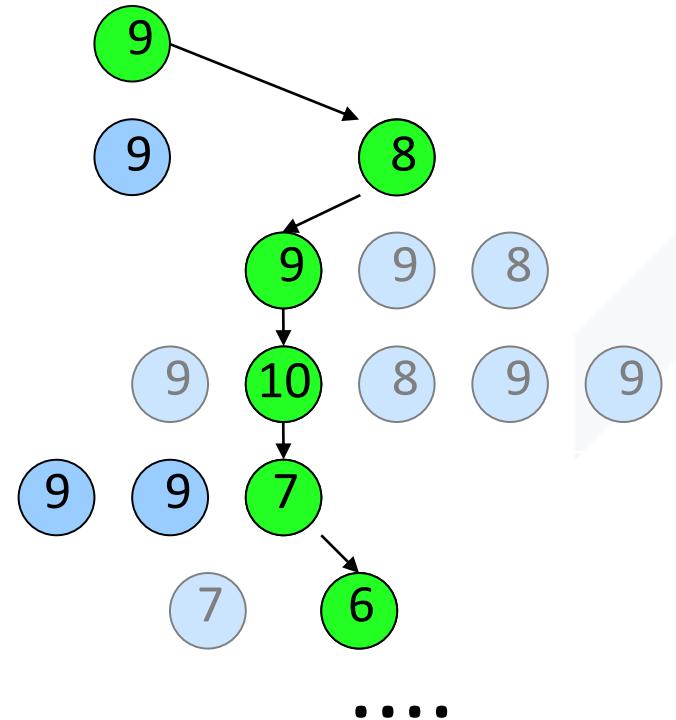
# FF – EHC in Practice

?



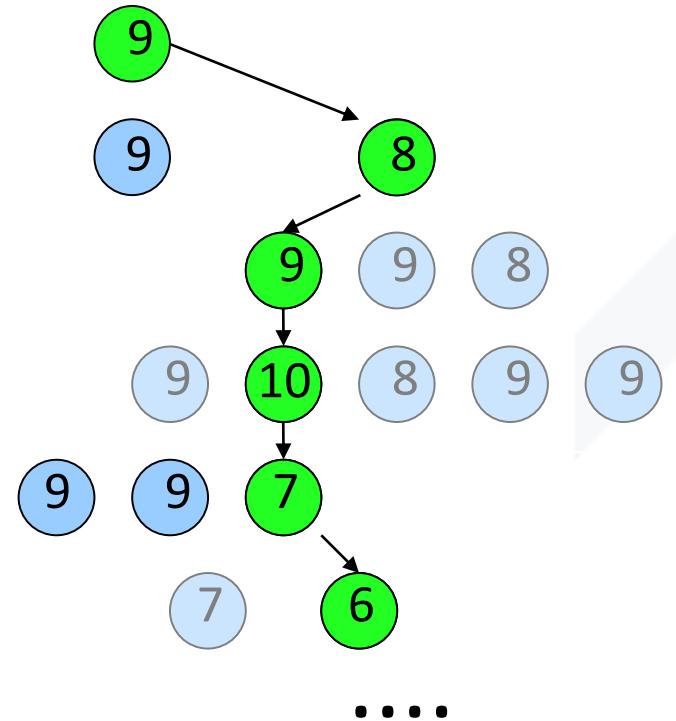
# Problems with EHC

- **EHC can sometimes fail to find a solution**
  - The RPG heuristic can lead it down dead ends
- **When EHC fails, FF resorts to systematic best-first search from the initial state**
  - FF maintains a priority queue of states (the state with the lowest  $h(S)$  value is stored first)
  - The front state in the queue is expanded each time
  - The successors are added to the queue, and so on (loop)
- **Searching on a plateau is expensive**
  - Systematic search is carried out on the part where the heuristic is worst



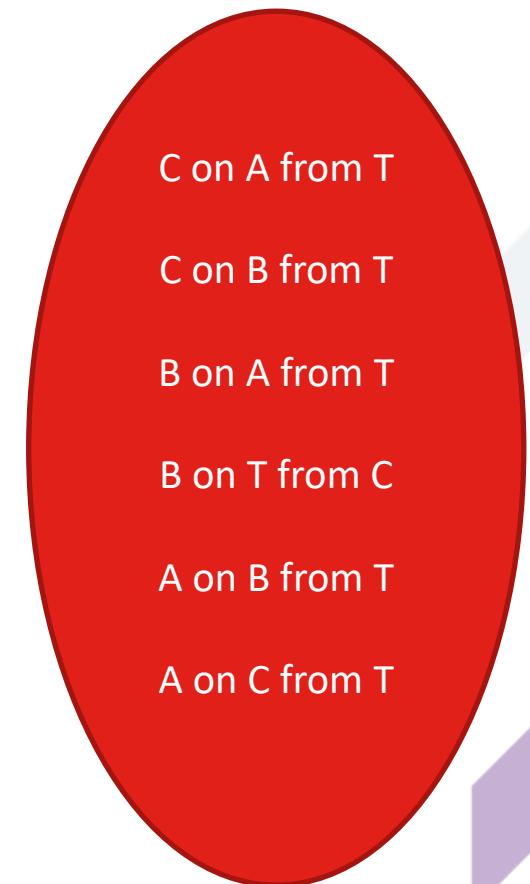
# EHC: Pros and Cons

- **EHC is fast:**
  - Greedy algorithm takes better states when found;
  - Can find solutions quickly.
- **EHC is incomplete:**
  - Suppose the solution was actually down one of those paths that we discarded (or never even generated) because another state looked better.
- **FF is complete:**
  - If EHC fails then start again from the initial state using best-first search.
  - Slower but complete.



# Extra State Pruning: Helpful Actions

- So far, when expanding a state  $S$ , have considered all applicable actions when making successors;
  - However, not all of them are interesting
  - e.g. unloading a package that has just been loaded
- In FF, extra search guidance is obtained from the RPG heuristic:
  - Anything that could achieve a goal in  $g(1)$  is a helpful action
  - Or, in other words, the first actions in the relaxed plan, and others like them.



# Classical Planning RPG Heuristic in the FF Planner

6CCS3AIP – Artificial Intelligence Planning

Dr Tommy Thompson



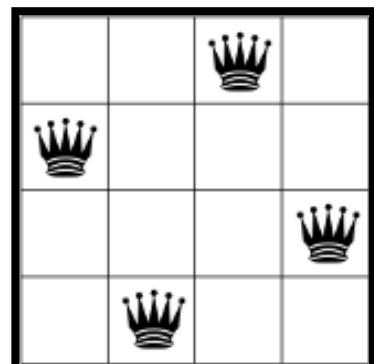
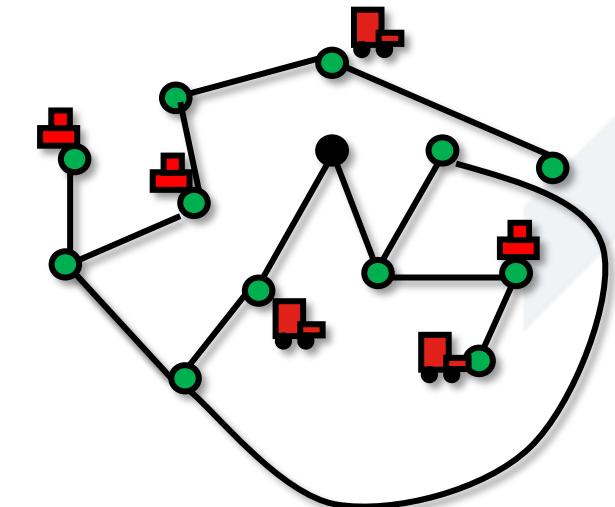
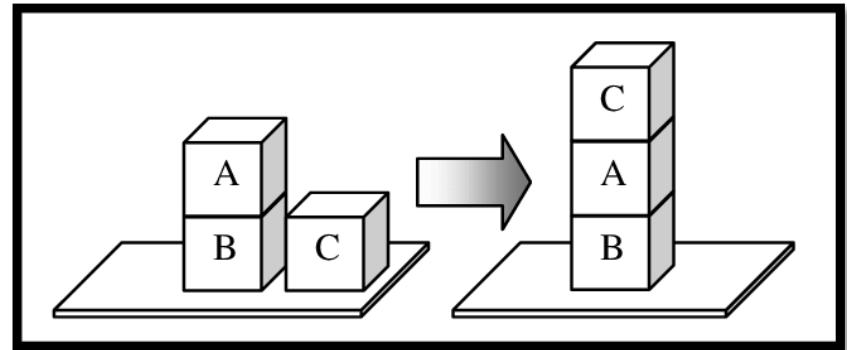
# Classical Planning Landmarks for Planning

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



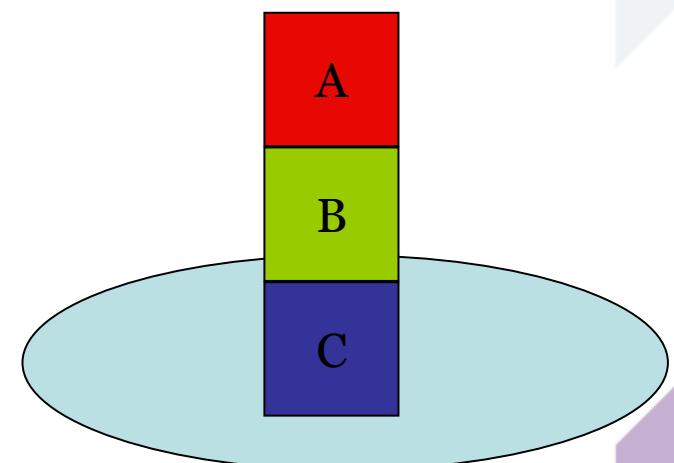
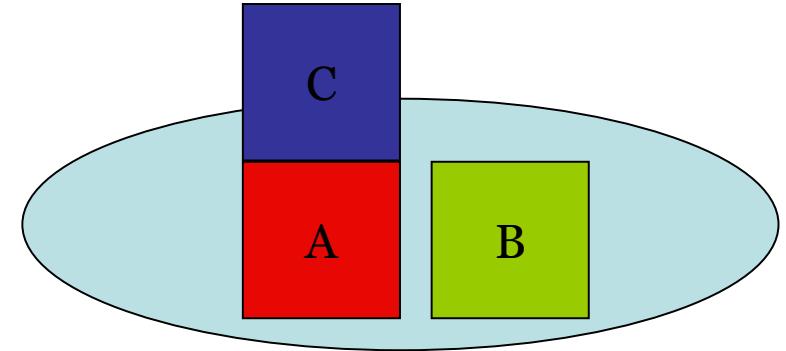
# Domain Independent Heuristics

- Building heuristics that can adapt to different domains/problems.
- Not reliant on specific information about the problem.
- We analyse aspects of the search and planning process to find potential heuristics.



# Relaxed Planning Graph Heuristic

- Relaxed Planning Graph (RPG) heuristic creates a simple layered approach to exploring the state space.
- Inspired by the GraphPlan system
  - GraphPlan covered later in this module.
  - For reference, see (Blum & Furst, 1995)
- The relaxed solution plan  $P' = \langle O_0, O_1, \dots, O_{m-1} \rangle$ 
  - Where each  $O_i$  is the set of actions selected in parallel at time step  $i$ , and  $m$  is the number of the first fact layer containing all goals
  - $h(s) := \sum |O_i|$



Blum, A. L., & Furst, M. L. (1995). Fast planning through planning graph analysis.

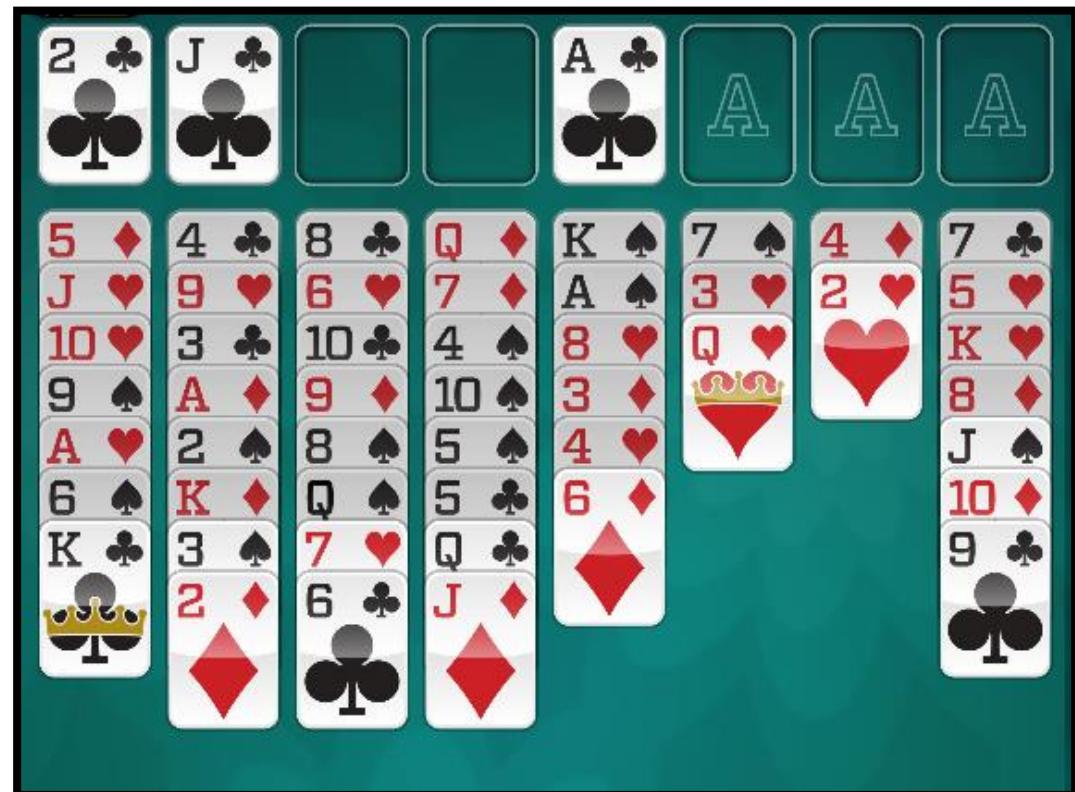
In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI95), pp. 1636

# Landmarks: Constraint-Based Heuristics

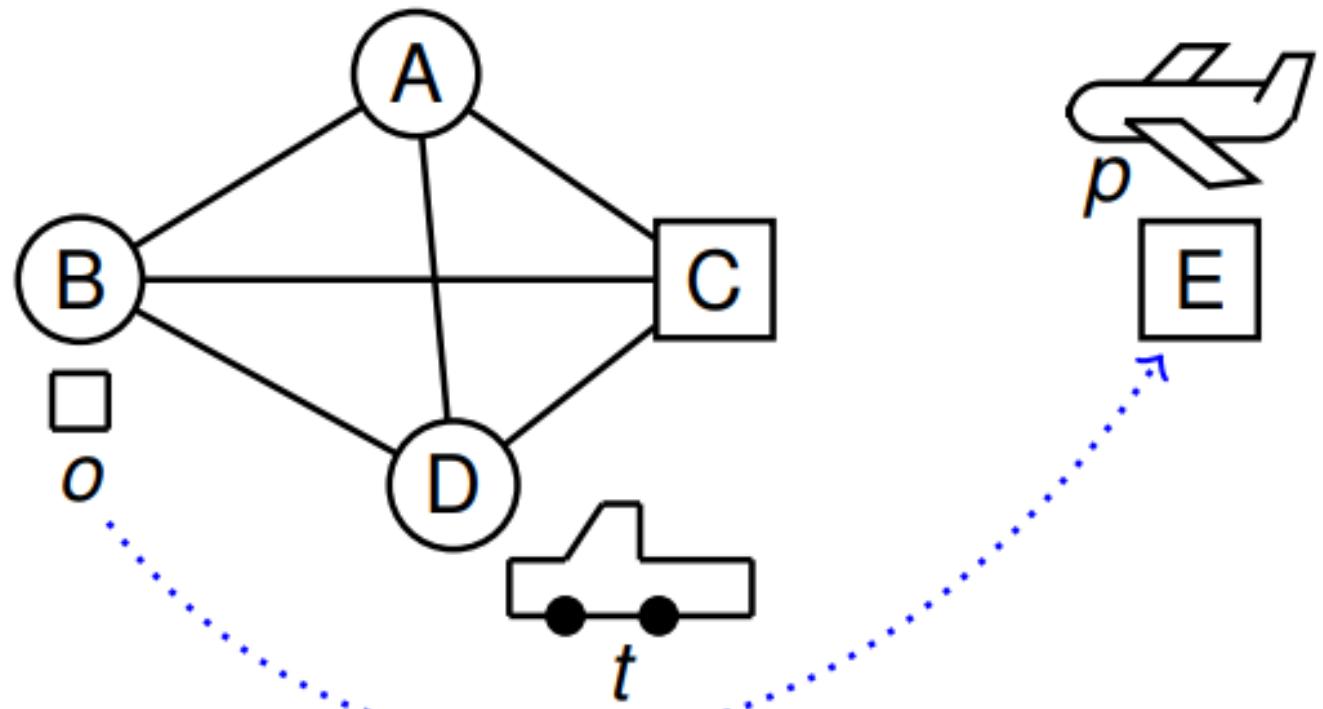
- Understanding and exploiting constraints that encapsulate facets of every possible solution of a problem.
- Some of these constraints can be denoted as landmarks of the solution.
  - Landmarks must be true at some point in every plan generated.
- Landmarks can be ordered to dictate the order in which they should be achieved.
- We aim to discover landmarks and their ordering automatically from the problem.

# Landmarks

- **Fact Landmark**
  - A **variable** takes a particular value **in at least one state**.
- **Action Landmark**
  - An **action must be applied** in the solution.
- **Disjunction Action Landmark**
  - One of a **set of possible actions** must be applied.



# Landmarks Example



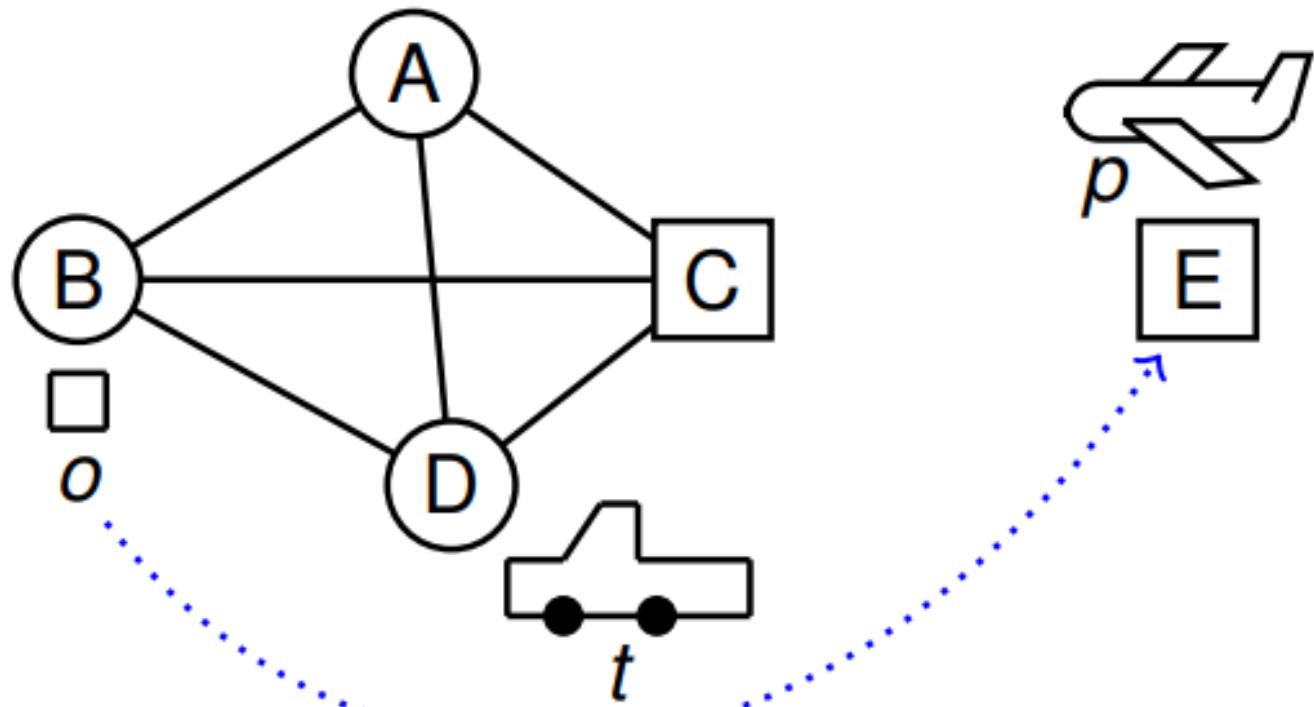
Examples c/o

Silvia Richter, Erez Karpas

Landmarks in Heuristic-Search Planning

ICAPS Tutorial, 2010

# Landmarks Example

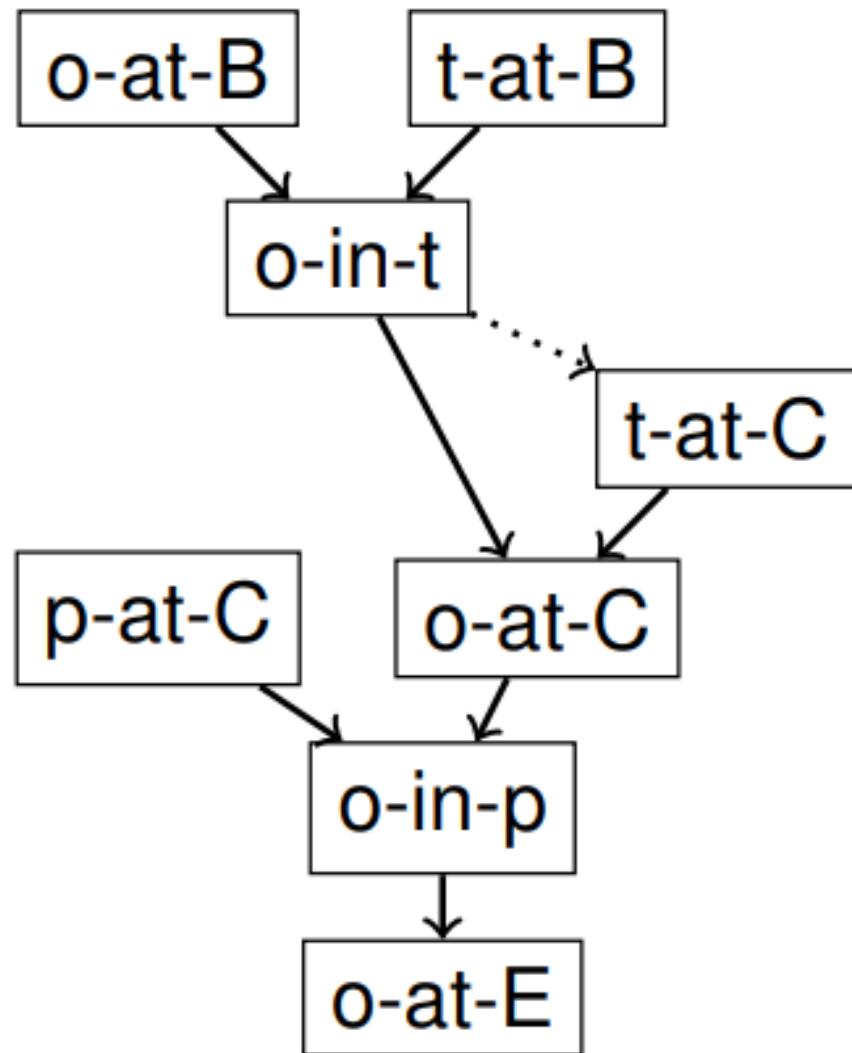


Examples c/o

Silvia Richter, Erez Karpas

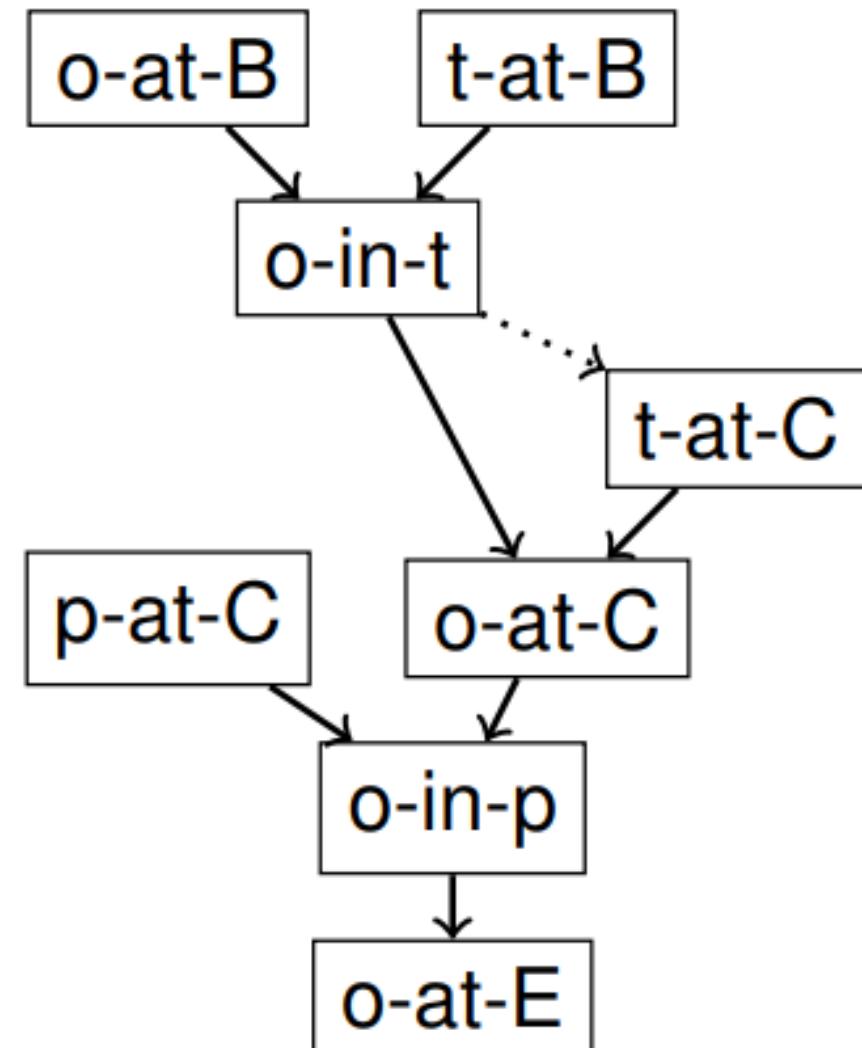
Landmarks in Heuristic-Search Planning

ICAPS Tutorial, 2010



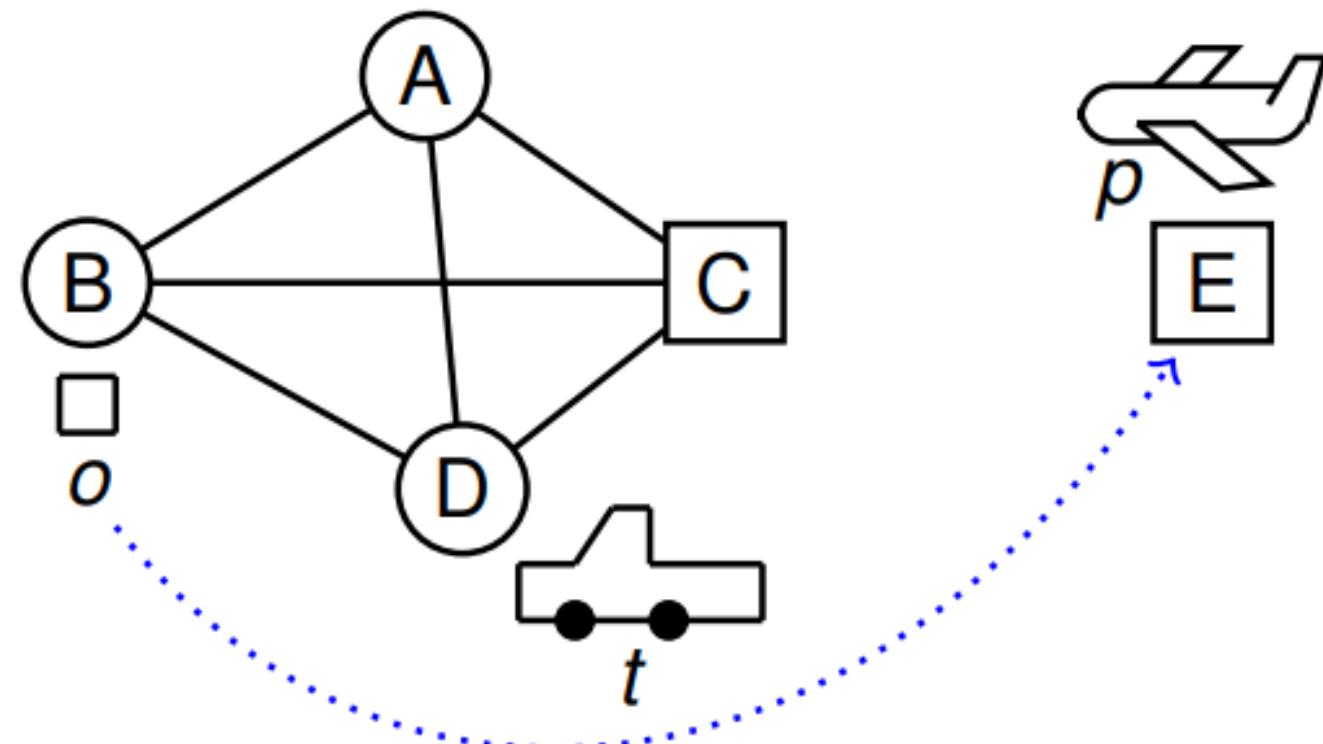
# How Do We Find Landmarks?

- We also need to find good landmarks, given some will not be informative.
  - The set of all possible actions is technically a disjunctive action landmark.
  - Every variable that is true in the initial state is a fact landmark.
- But finding landmarks can be as hard as planning itself (PSPACE-complete).
- Can exploit the relaxed planning graph technique to generate landmarks.



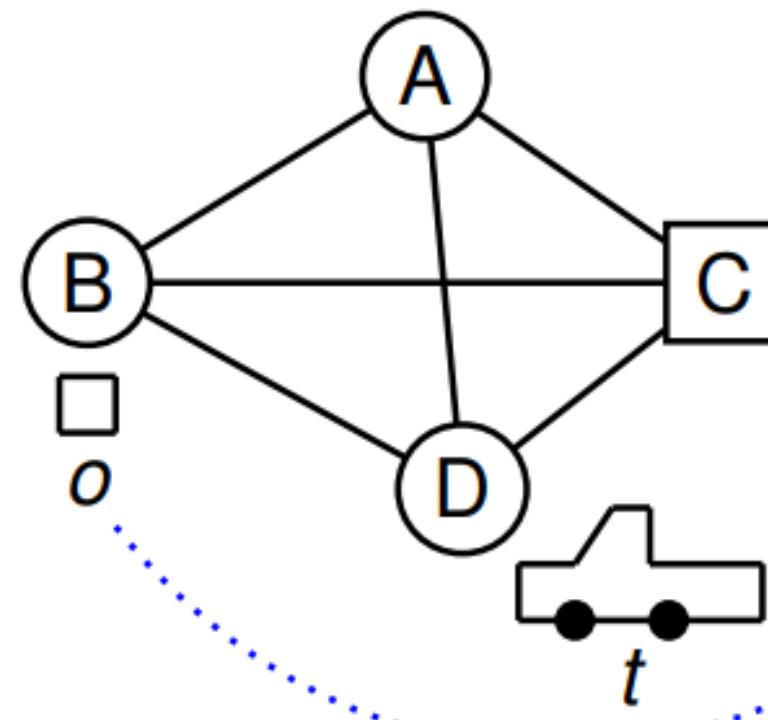
# Finding Landmarks # 1: Deletion Relaxation w/ RPG

- Delete Relaxation Landmarks
- Iterating through all actions...
  - Remove an action if it adds a fact to the planning problem.
  - Build the relaxed planning graph.
  - If the goal no longer appears in the RPG, then the action was a landmark of the problem.
- It works, but has issues.
  - Slow to execute..



# Finding Landmarks #2: Backchaining

- Treat each goal (B) as a landmark, for each goal:
  - Take intersection of preconditions of all actions that achieve goal (A)
  - If numerous actions share A and achieve B, **A is also a landmark**;
  - A is also ordered before (must be achieved before) the goal.
- Repeat for all landmarks found.
- Can do a bit better (find more landmarks) by looking whether A is needed to achieve B for the first time (build RPG without A, see if B appears).
  - e.g. suppose there is another location F joined to E, we can drive from F->E to achieve the goal but not unless we already went through E...

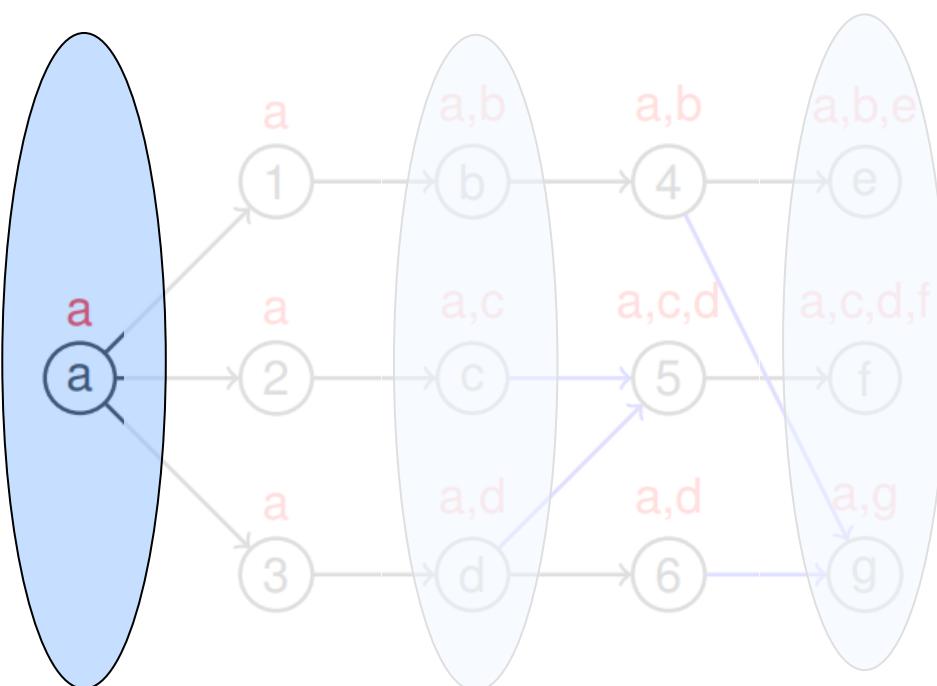


“Ordered Landmarks in Planning“.

J. Hoffmann, J. Porteous, S. Creswell JAIR, Volume 22, pages 215-278.

# Finding Landmarks #3: RPG Propogation

- Actions (numbers) take union over preconditions' labels (all are necessary).
- Facts (letters) take intersection of achievers' labels (at least one is necessary).



“Landmark extraction via planning graph propagation”.

Zhu, L., & Givan, R. ICAPS 2003 Doctoral Consortium. (Example Sylvia Richter)

# RPG Propagation: Extracting Landmarks

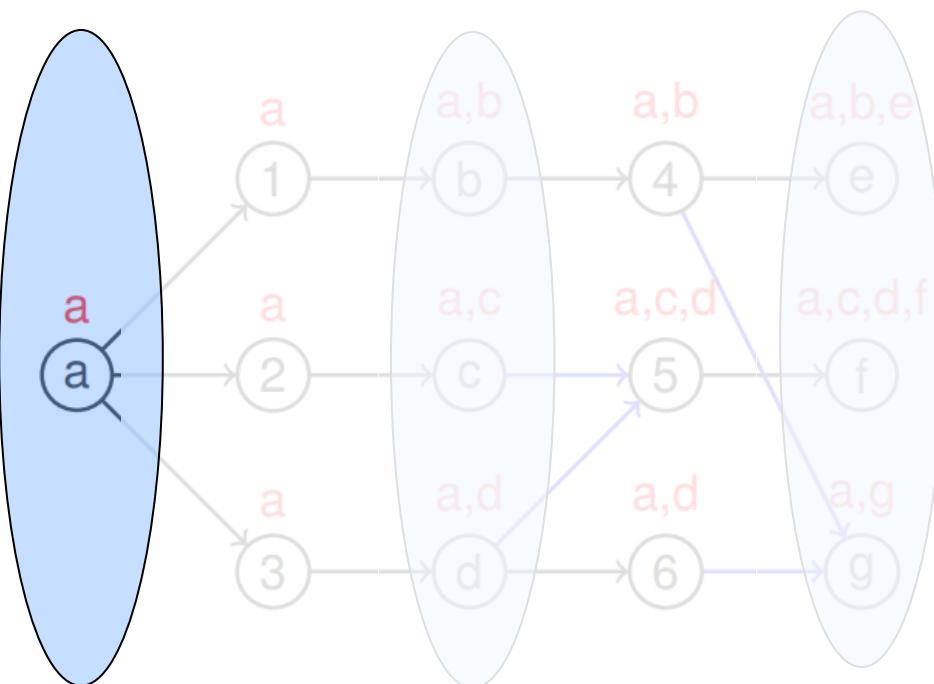
- Finds all causal delete relaxed landmarks in polynomial time
- Build the RPG not just until the goals appear but until all goals appear and we reach a layer with identical labels to the previous one (labels stop changing).
- The landmarks are the labels on the goal nodes in final layer.
- Possible first achievers of B are achievers that do not have B in their label.

# Landmark Orderings

- It can be useful to know the *order* in which landmarks must be achieved.
- Several types of *sound* ordering:
  - *Necessary Ordering*  $A \rightarrow_n B$  : A is always true one step before B;
  - *Greedy-necessary Ordering*,  $A \rightarrow_{gn} B$ : A is true one step before B is true for the first time;
  - *Natural Ordering*  $A \rightarrow B$ : A is true some time before B.
  - $A \rightarrow_n B \Rightarrow A \rightarrow_{gn} B \Rightarrow A \rightarrow B$
- And *unsound* orderings (used in heuristics):
  - *Reasonable Ordering*  $A \rightarrow_r B$ : if B was achieved before A then the plan should delete B to achieve A and reacieve B after (or at the same time as A).
  - *Obedient Reasonable Ordering*  $A \rightarrow_{or} B$ : if B was achieved before A then any plan that obeys reasonable orderings must delete B to achieve A and reacieve B after (or at the same time as A).

# Landmark Orderings

- $A \rightarrow B$  if A forms part of the label for B in the final layer
- $A \rightarrow_{gn} B$  if A is precondition for all possible first achievers of B (achievers not labelled with B).



“Landmark extraction via planning graph propagation”

Zhu, L., & Givan, R. ICAPS 2003 Doctoral Consortium. (Example Sylvia Richter)

# Classical Planning Landmarks for Planning

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Classical Planning Searching with Landmarks

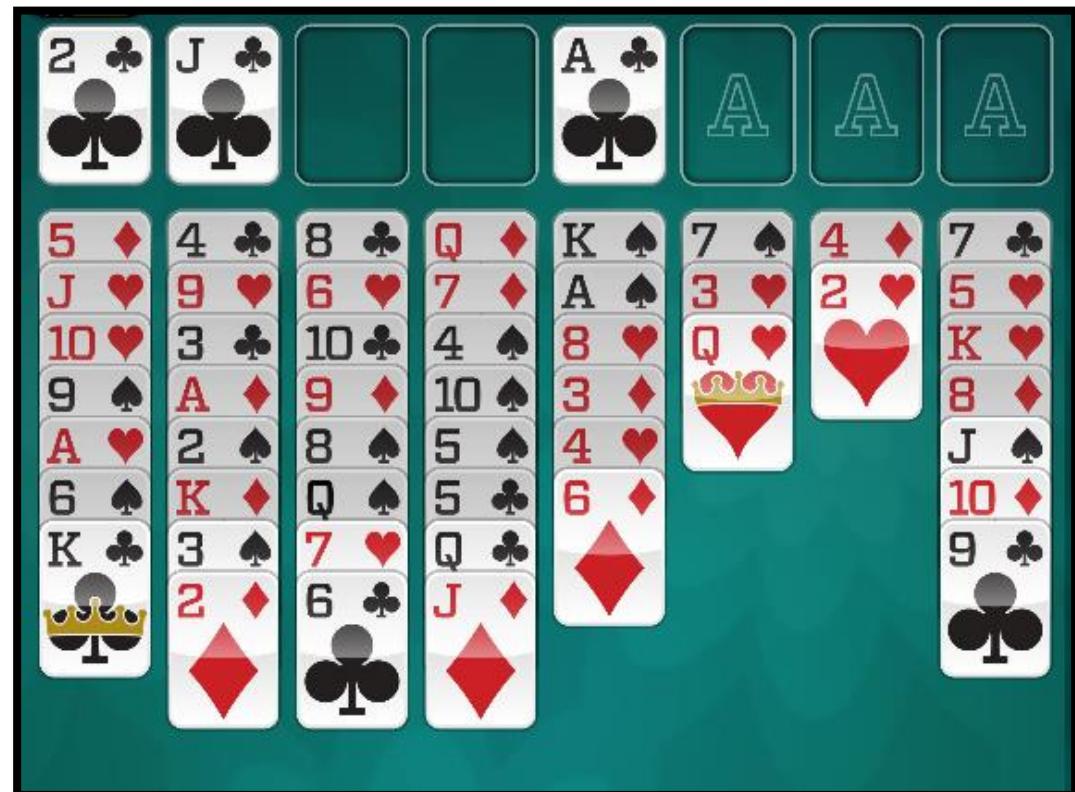
6CCS3AIP – Artificial Intelligence Planning

Dr Tommy Thompson

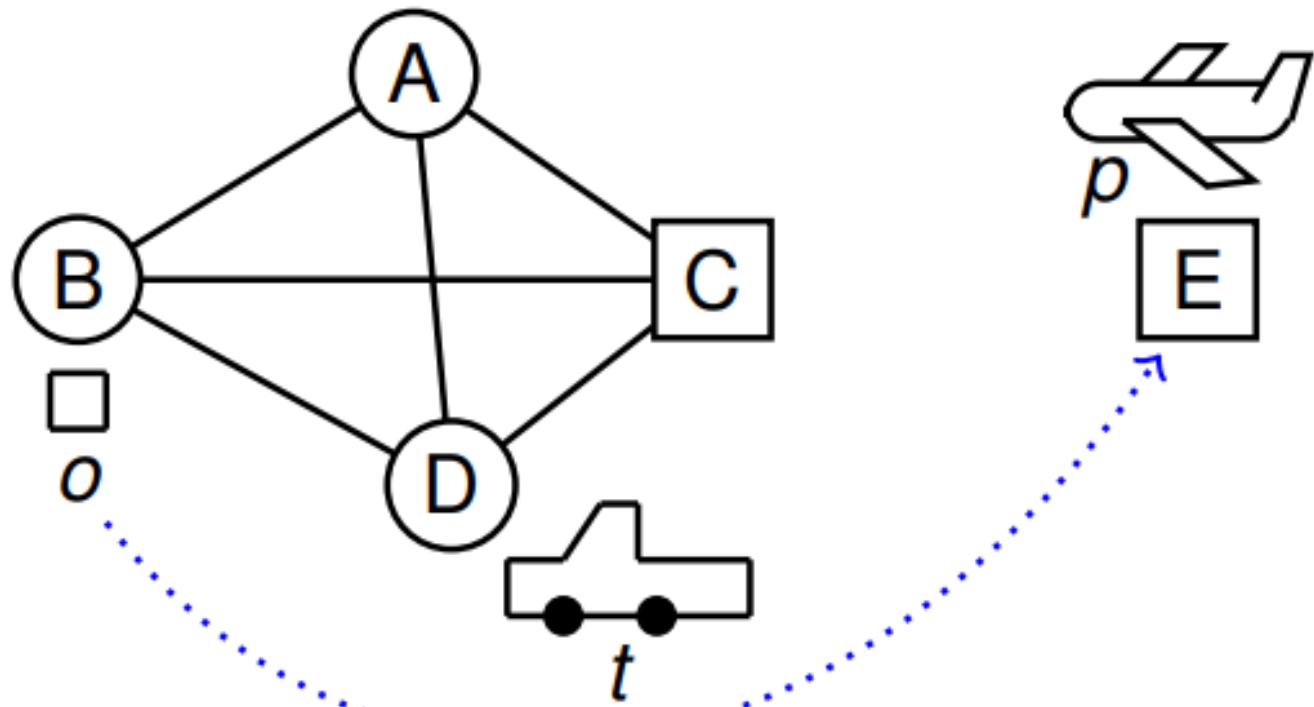


# Landmarks

- **Fact Landmark**
  - A **variable** takes a particular value **in at least one state**.
- **Action Landmark**
  - An **action must be applied** in the solution.
- **Disjunction Action Landmark**
  - One of a **set of possible actions** must be applied.



# Landmarks Example

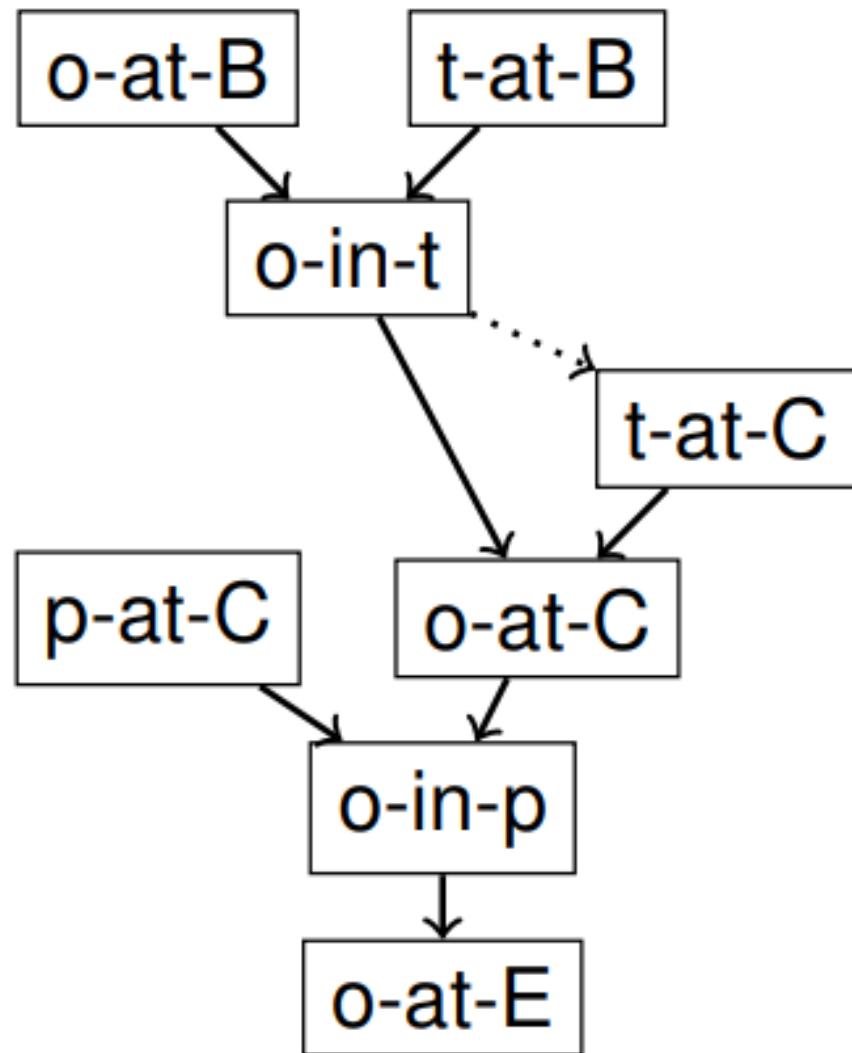


Examples c/o

Silvia Richter, Erez Karpas

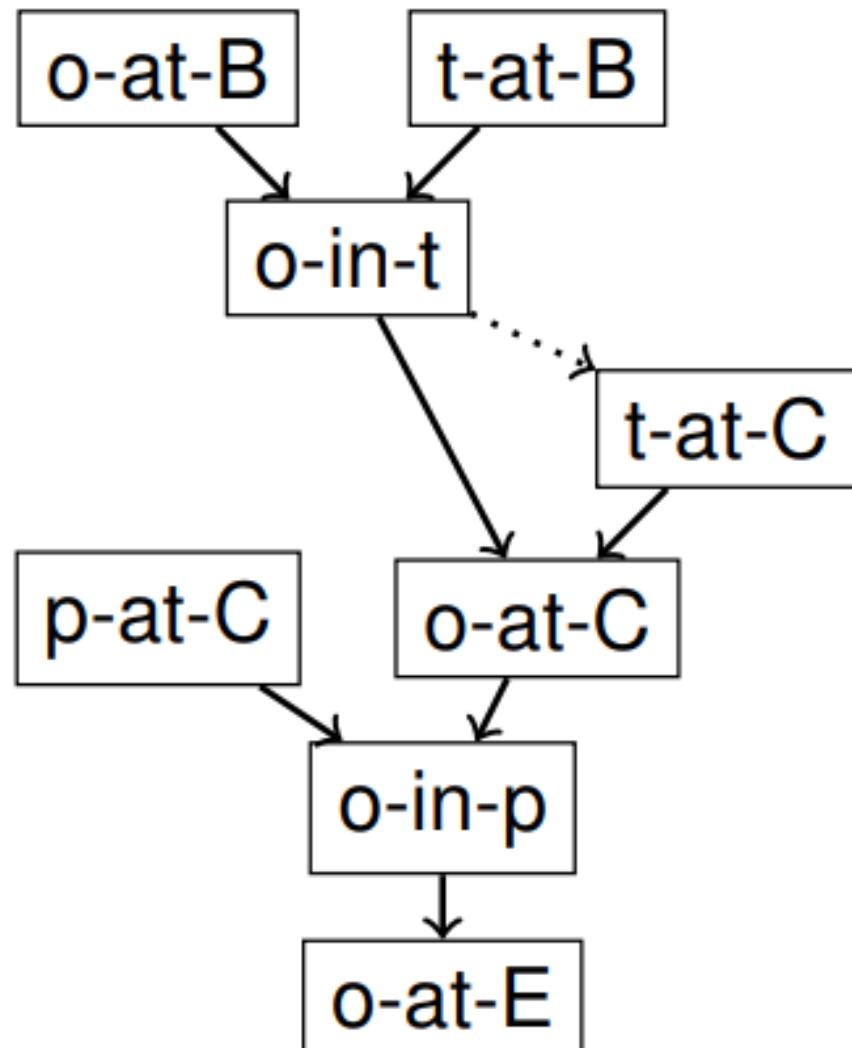
Landmarks in Heuristic-Search Planning

ICAPS Tutorial, 2010



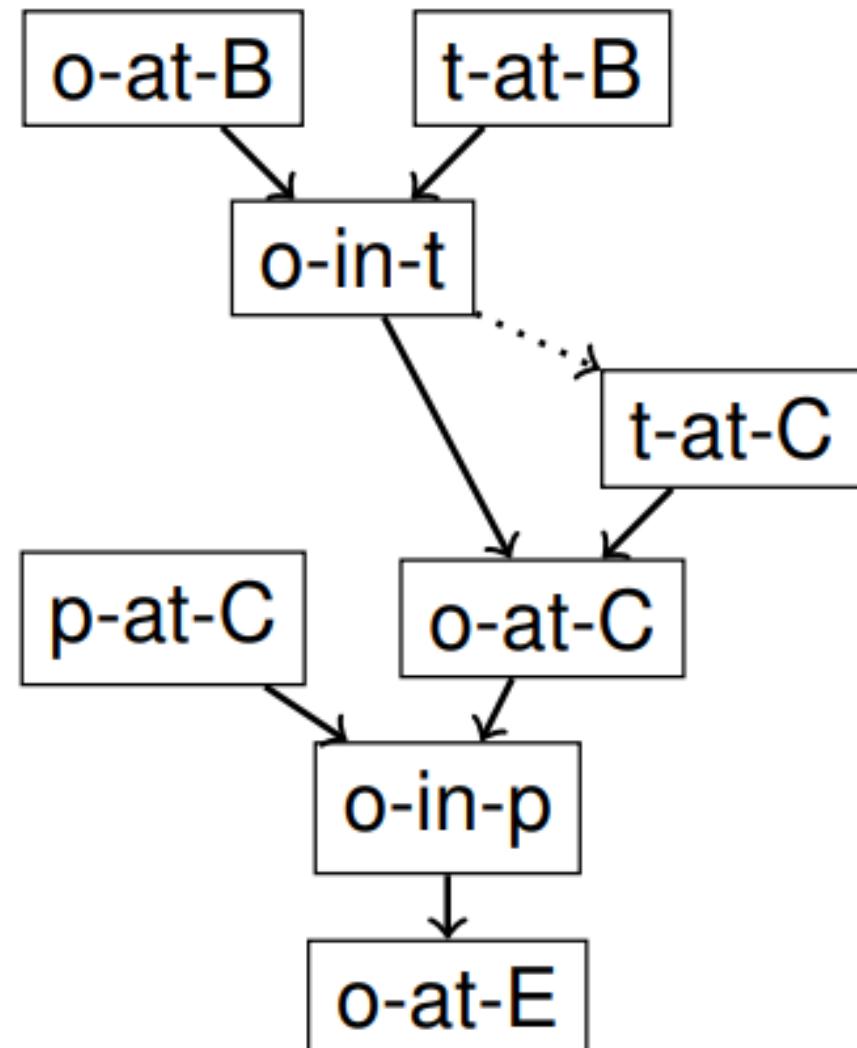
# Making Landmarks Useful

- So we can establish landmarks, but how can we use them
- Provided we generate the landmarks (and their orderings in a pre-processing phase – prior to planning – we can then use them during search in different ways...
  - Landmarks as Planning Subgoals
  - Landmarks as Heuristic Estimates
  - Admissible Landmark Heuristics



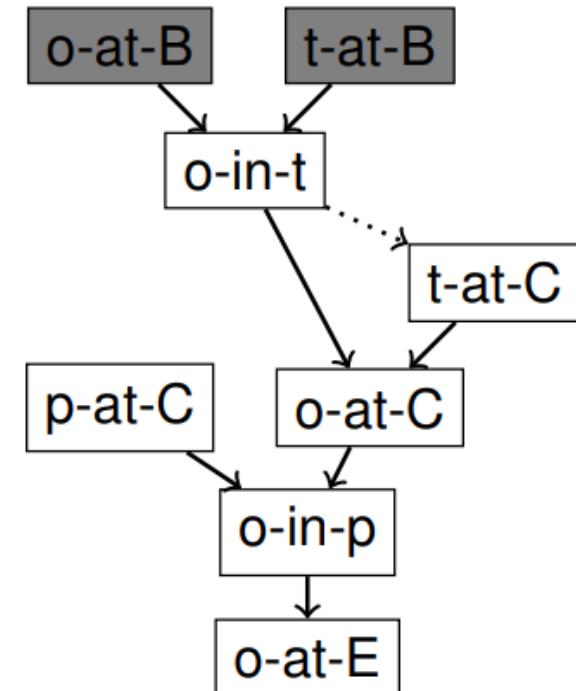
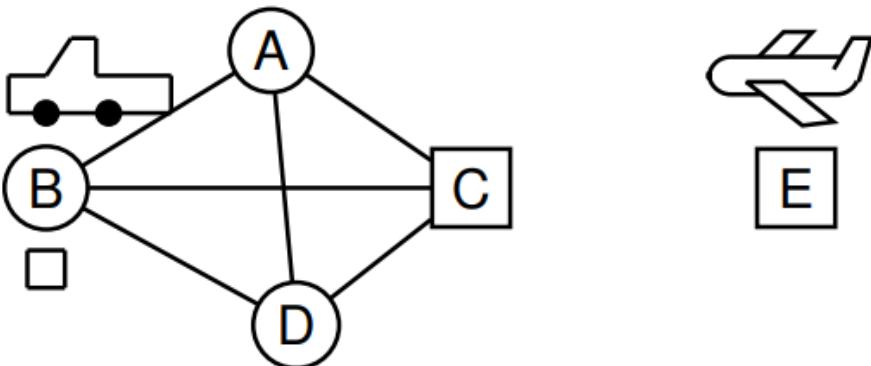
# Landmarks as Subgoals

- Simple approach: plan to one set of landmarks, repeat until all landmarks satisfied.
- Method:
  - Given some landmarks and a (partial) ordering on them.
  - Set the goal to (a disjunction over) the first landmark (**s**), according to the landmark graph, find a plan;
  - Now update the initial state, plan to the next landmark(s) etc



# Planning Between Landmarks

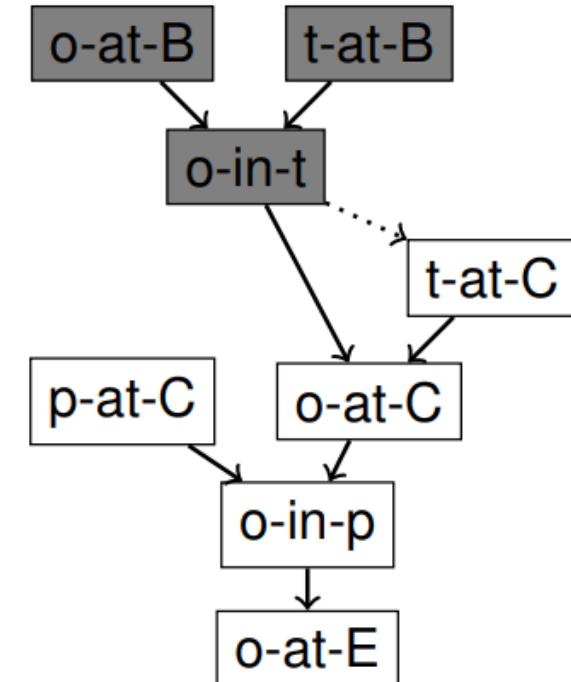
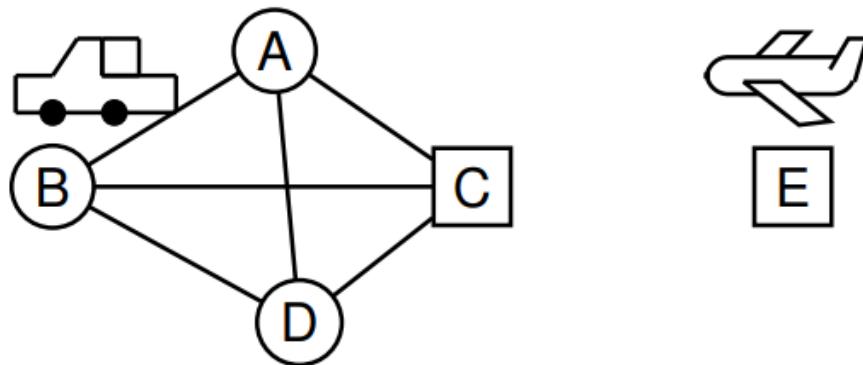
Examples c/o Erez Karpas



- Partial plan: Drive-t-B
- Goal: o-in-t ∨ p-at-C

# Planning Between Landmarks

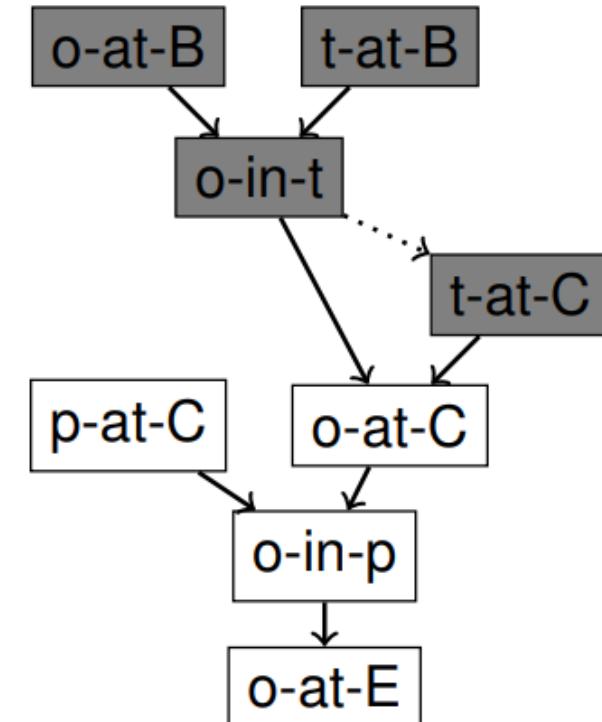
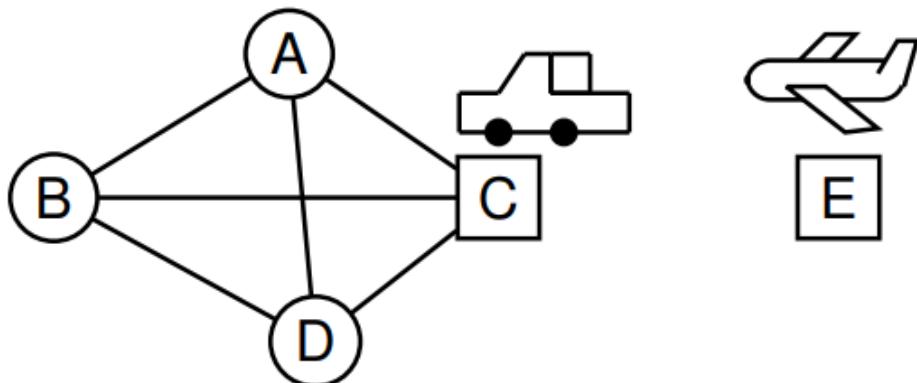
*Examples c/o Erez Karpas*



- Partial plan: Drive-t-B, Load-o-B
- Goal: t-at-C  $\vee$  p-at-C

# Planning Between Landmarks

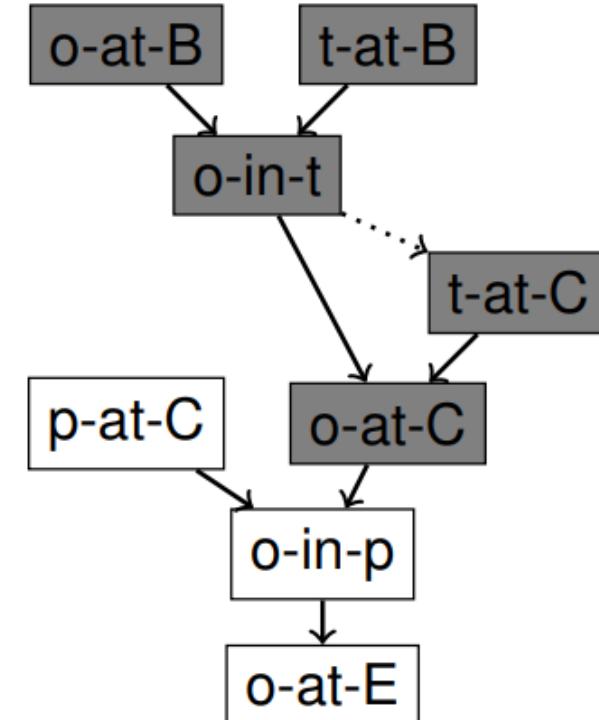
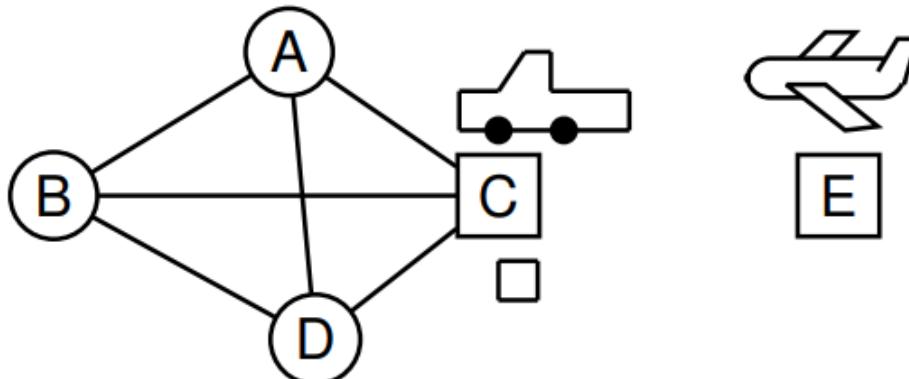
Examples c/o Erez Karpas



- Partial plan: Drive-t-B, Load-o-B, Drive-t-C
- Goal: o-at-C  $\vee$  p-at-C

# Planning Between Landmarks

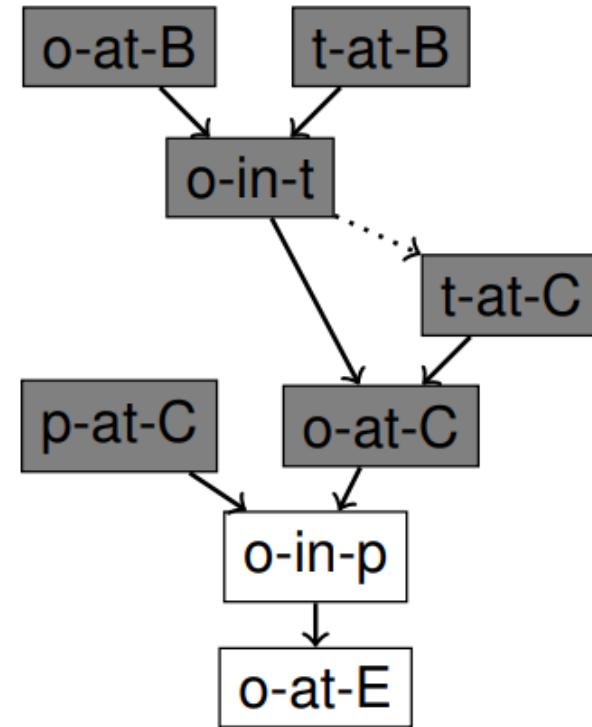
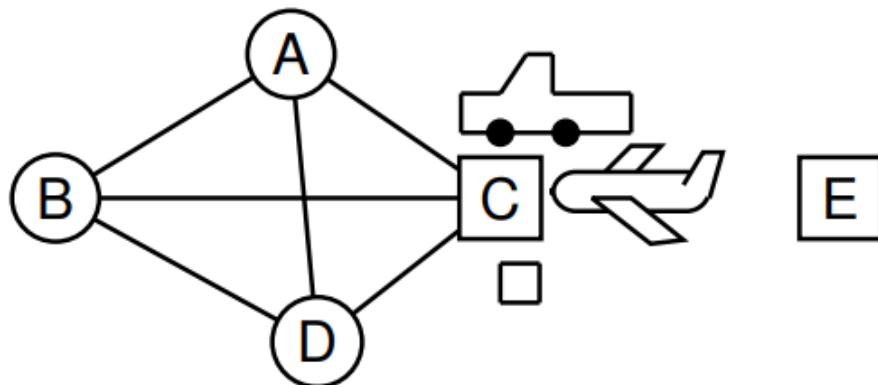
Examples c/o Erez Karpas



- Partial plan: Drive-t-B, Load-o-B, Drive-t-C, Unload-o-C
- Goal: p-at-C

# Planning Between Landmarks

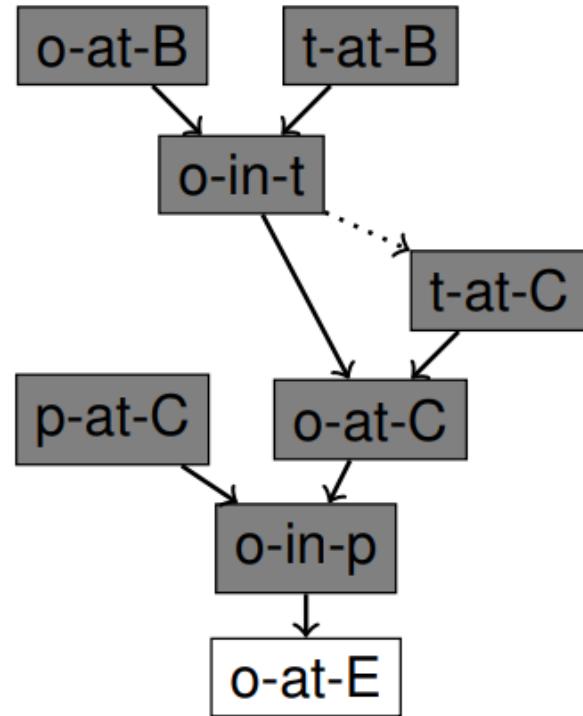
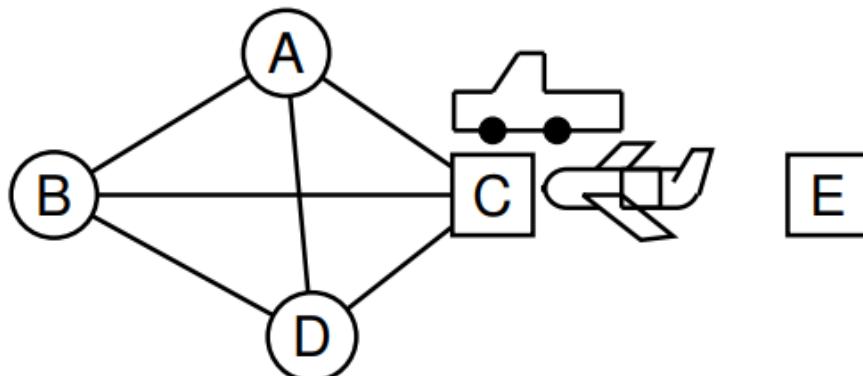
Examples c/o Erez Karpas



- Partial plan: Drive-t-B, Load-o-B, Drive-t-C, Unload-o-C, Fly-p-C
- Goal: o-in-p

# Planning Between Landmarks

Examples c/o Erez Karpas

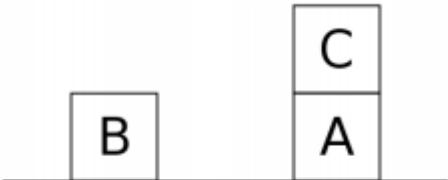


- Partial plan: Drive-t-B, Load-o-B, Drive-t-C, Unload-o-C,  
Fly-p-C, Load-o-p
- Goal: o-at-E

# That was a good one, consider another

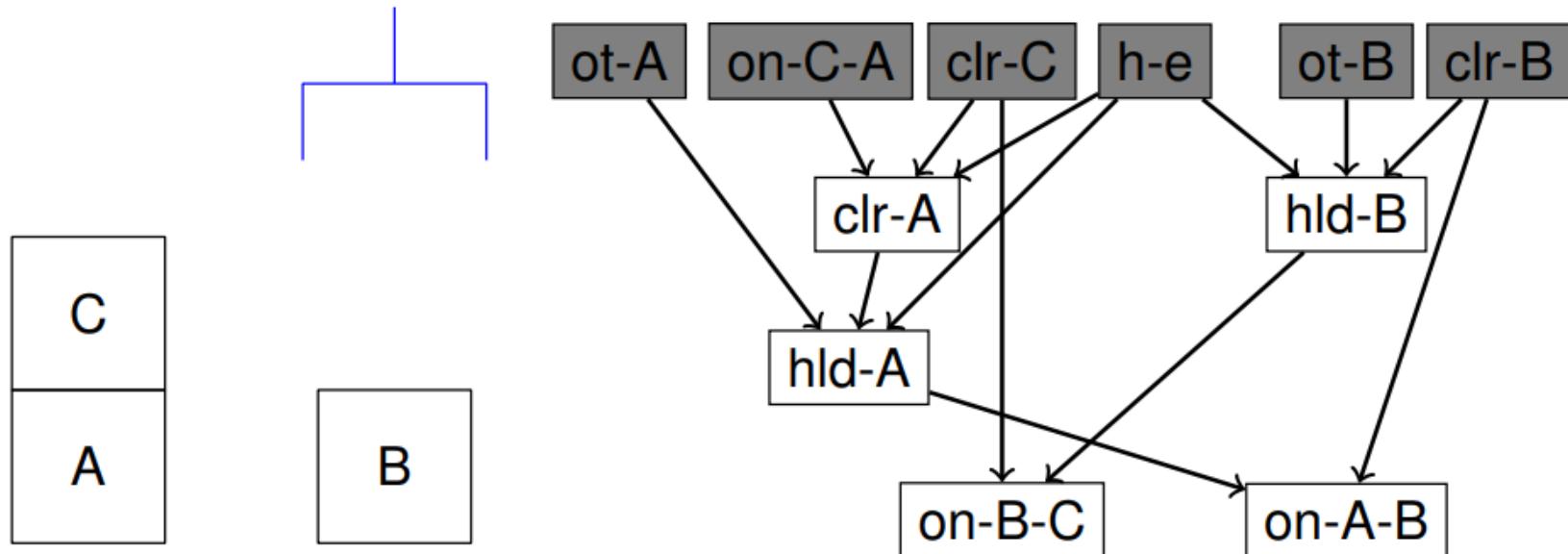
*c/o Erez Karpas*

- The Sussman Anomaly
- Goal: (on A B) (on B C)



# That was a good one, consider another

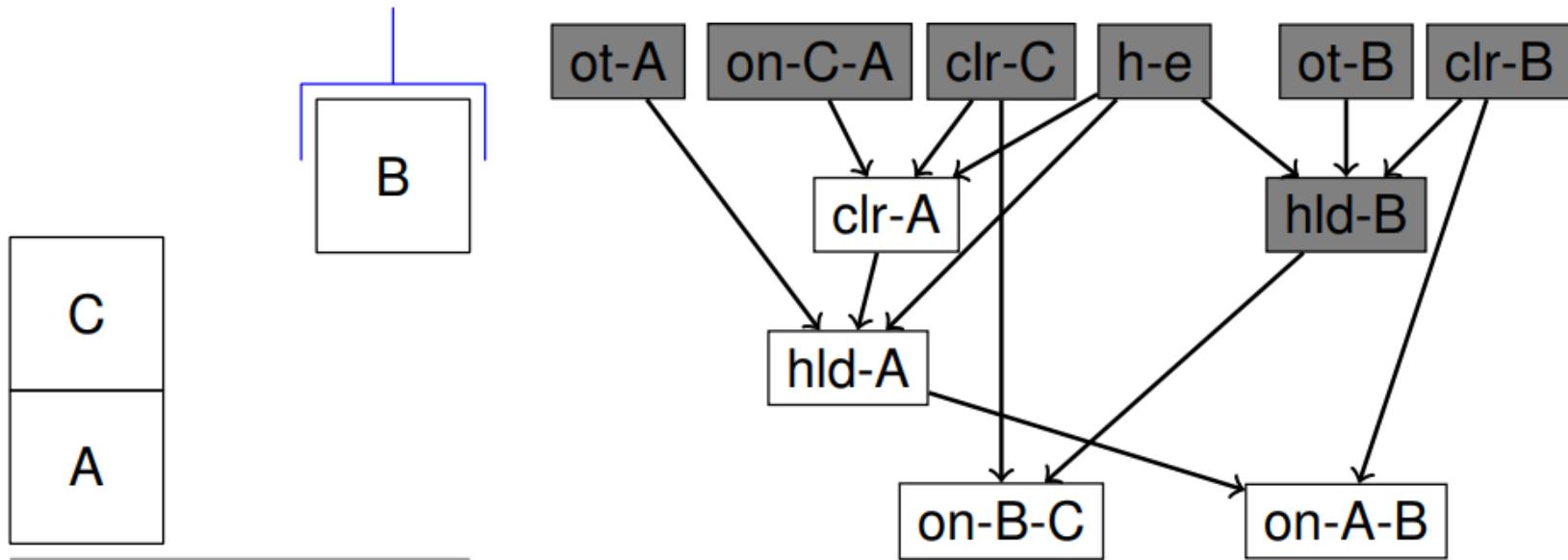
c/o Erez Karpas



- Partial plan:  $\emptyset$
- Goal: clear-A  $\vee$  holding-B

# That was a good one, consider another

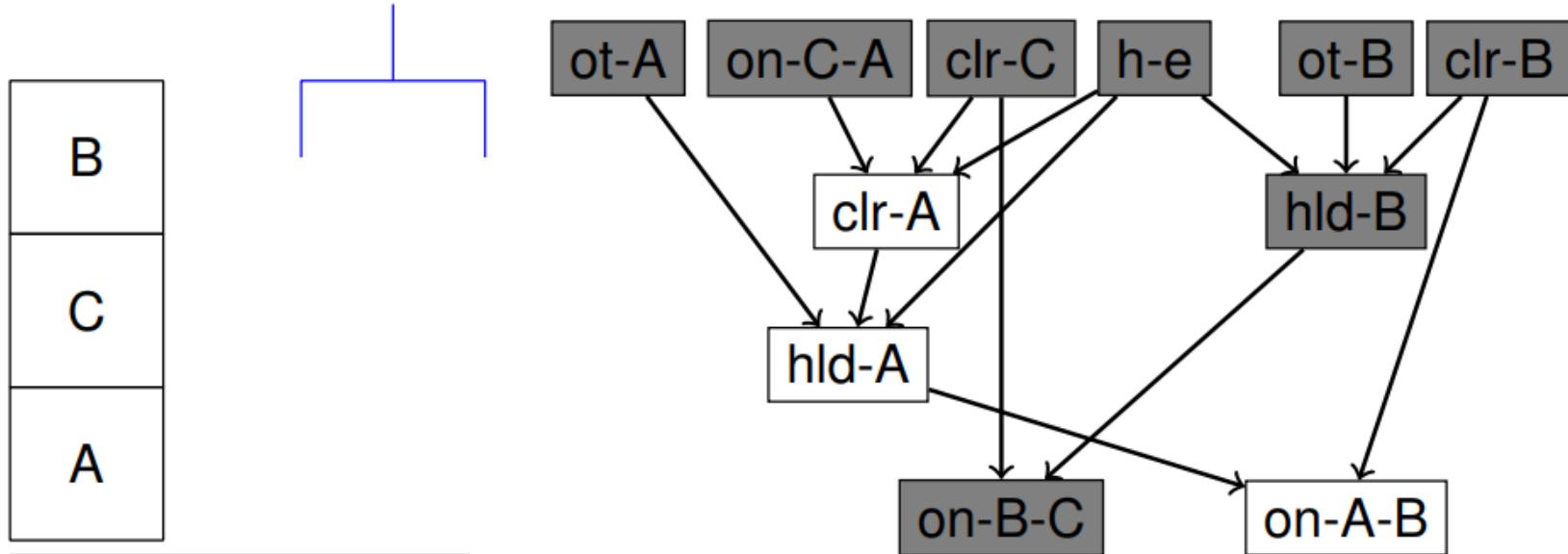
c/o Erez Karpas



- Partial plan: Pickup-B
- Goal: clear-A  $\vee$  on-B-C

# That was a good one, consider another

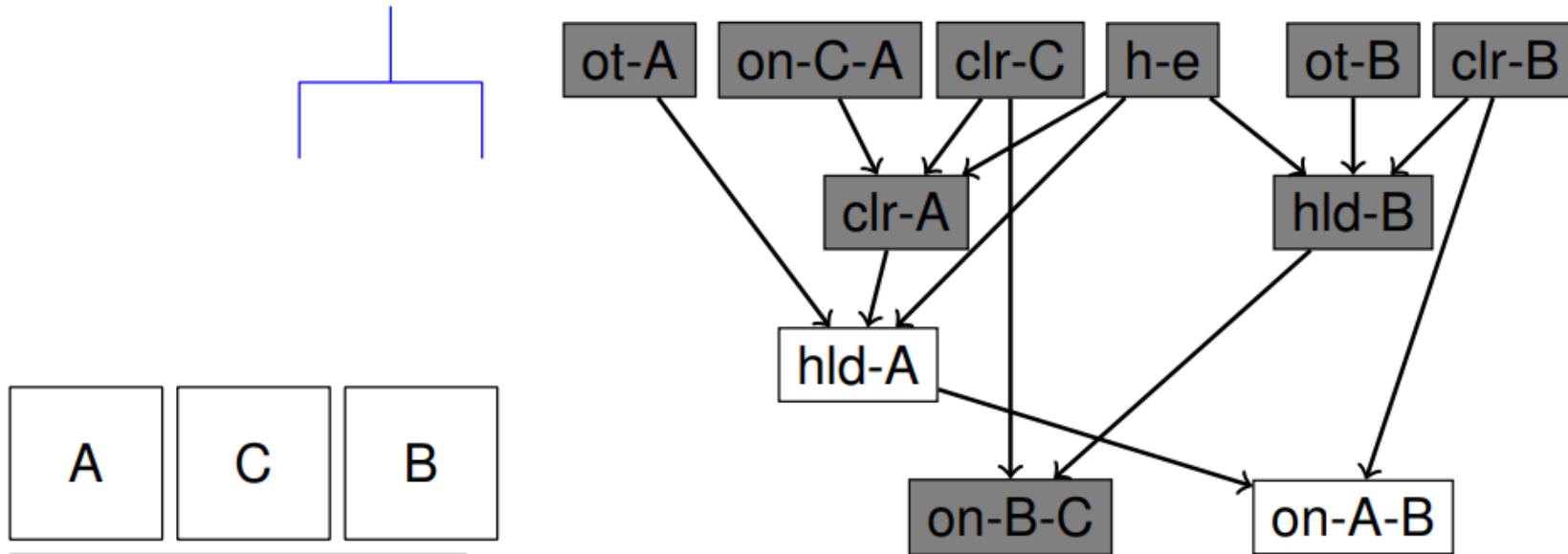
c/o Erez Karpas



- Partial plan: Pickup-B, Stack-B-C
- Goal: clear-A

# That was a good one, consider another

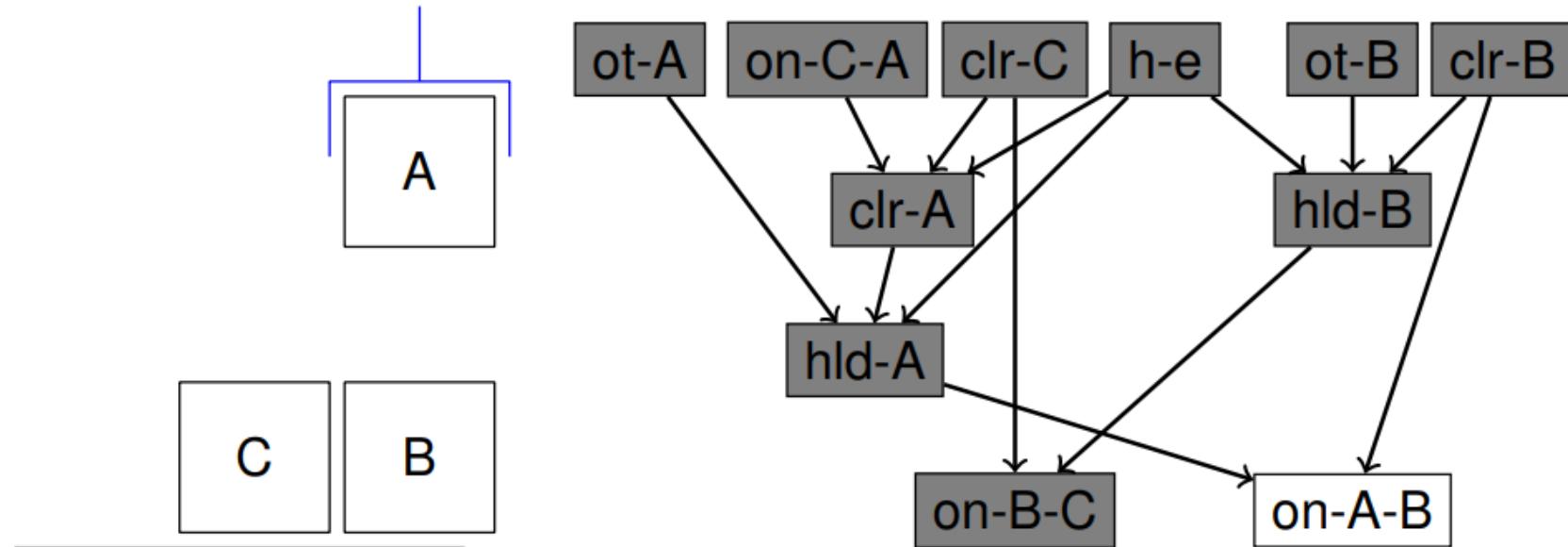
c/o Erez Karpas



- Partial plan: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C
- Goal: holding-A

# That was a good one, consider another

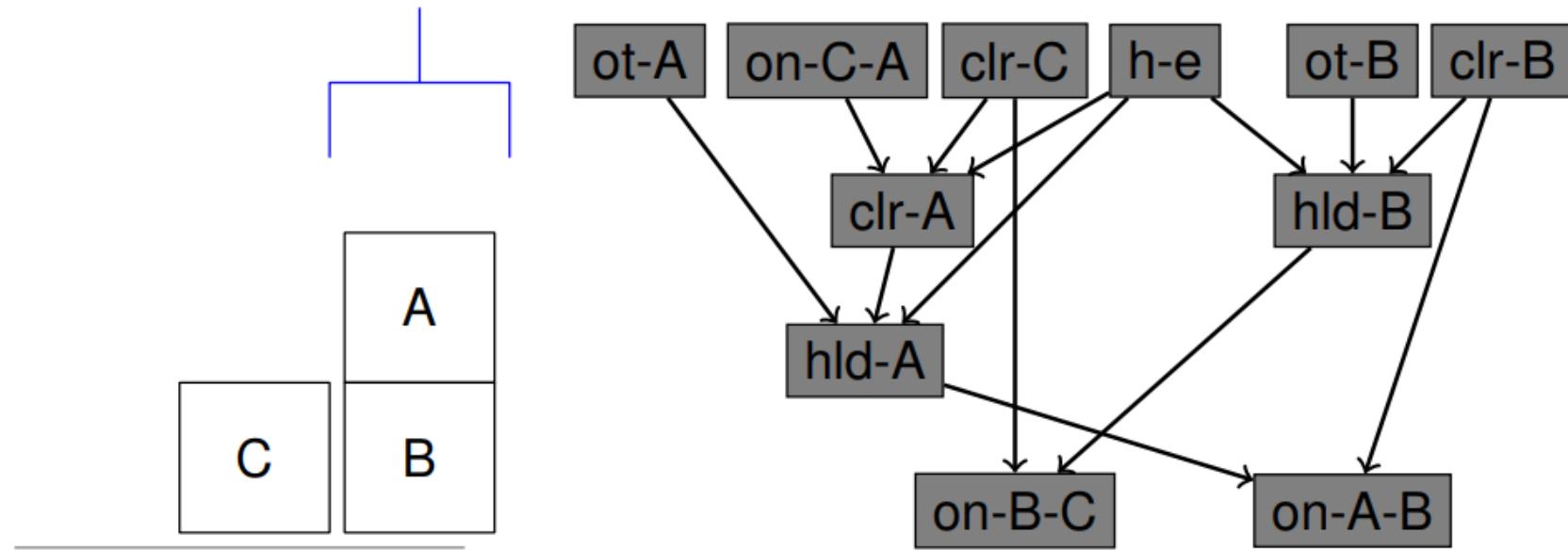
c/o Erez Karpas



- Partial plan: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C, Pickup-A
- Goal: on-A-B

# That was a good one, consider another

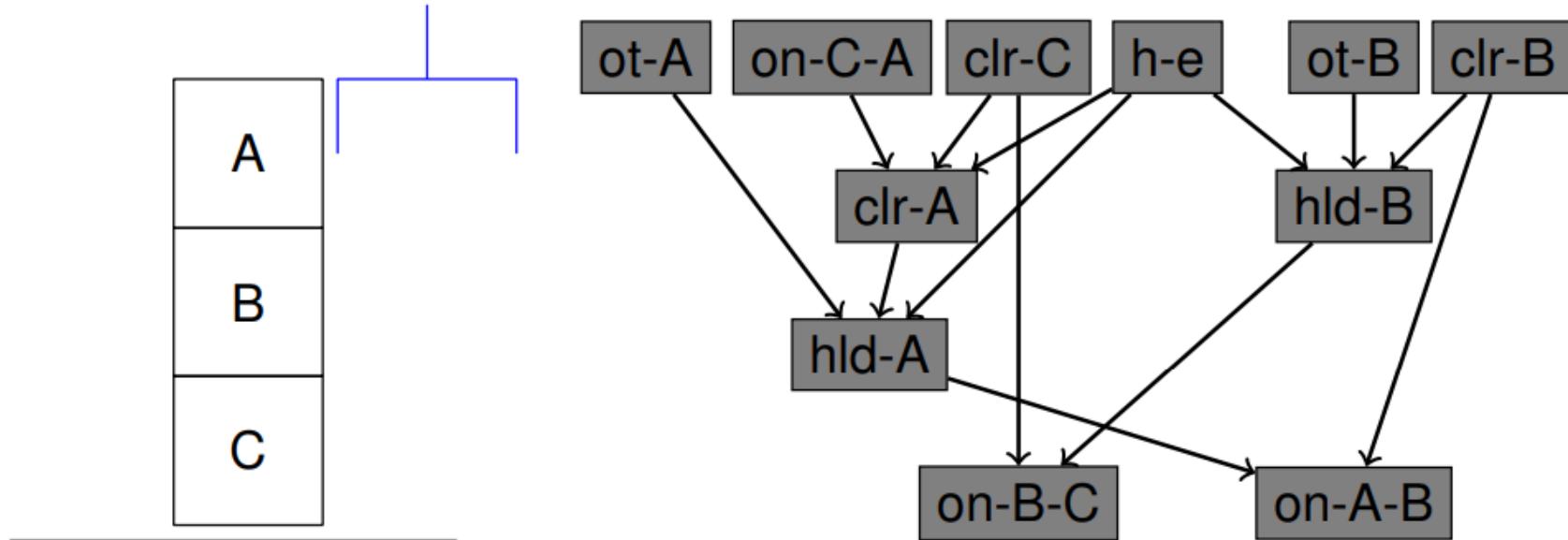
c/o Erez Karpas



- Partial plan: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C, Pickup-A, Stack-A-B
- Goal: Still need to achieve on-B-C

# That was a good one, consider another

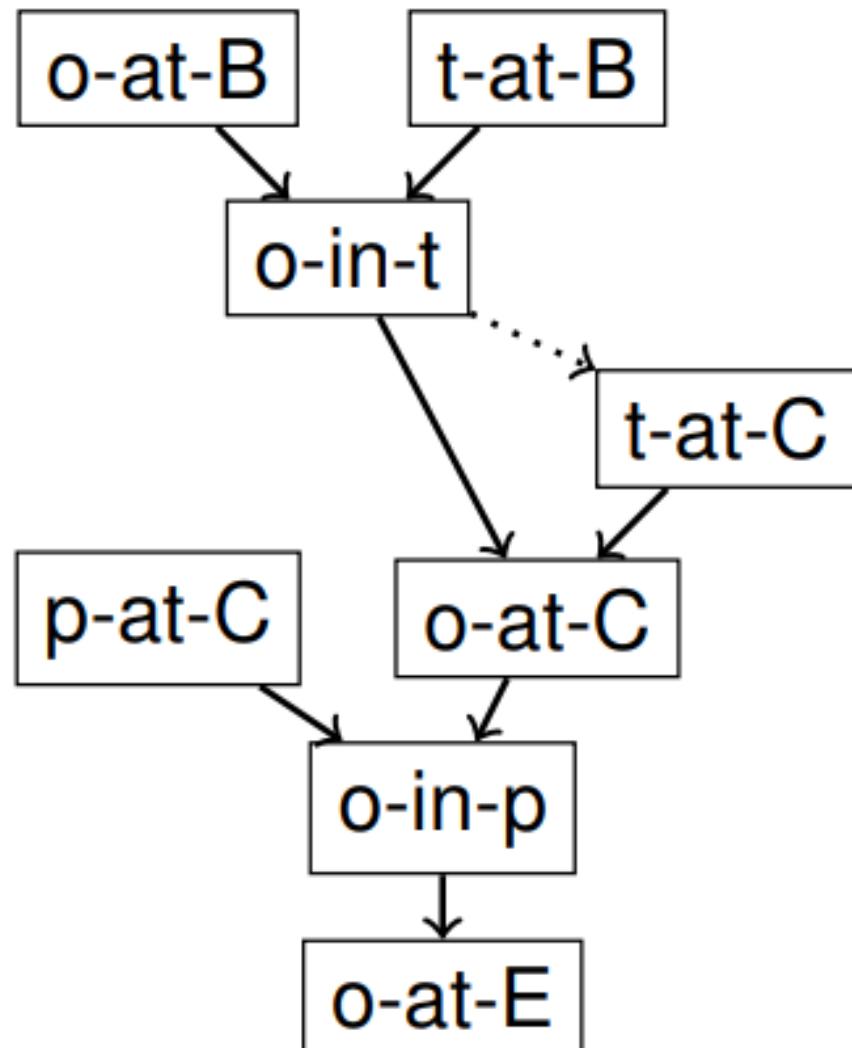
c/o Erez Karpas



- Partial plan: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C, Pickup-A, Stack-A-B, Unstack-A-B, Putdown-A, Pickup-B, Stack-B-C, Pickup-A, Stack-A-B
- Goal:  $\emptyset$

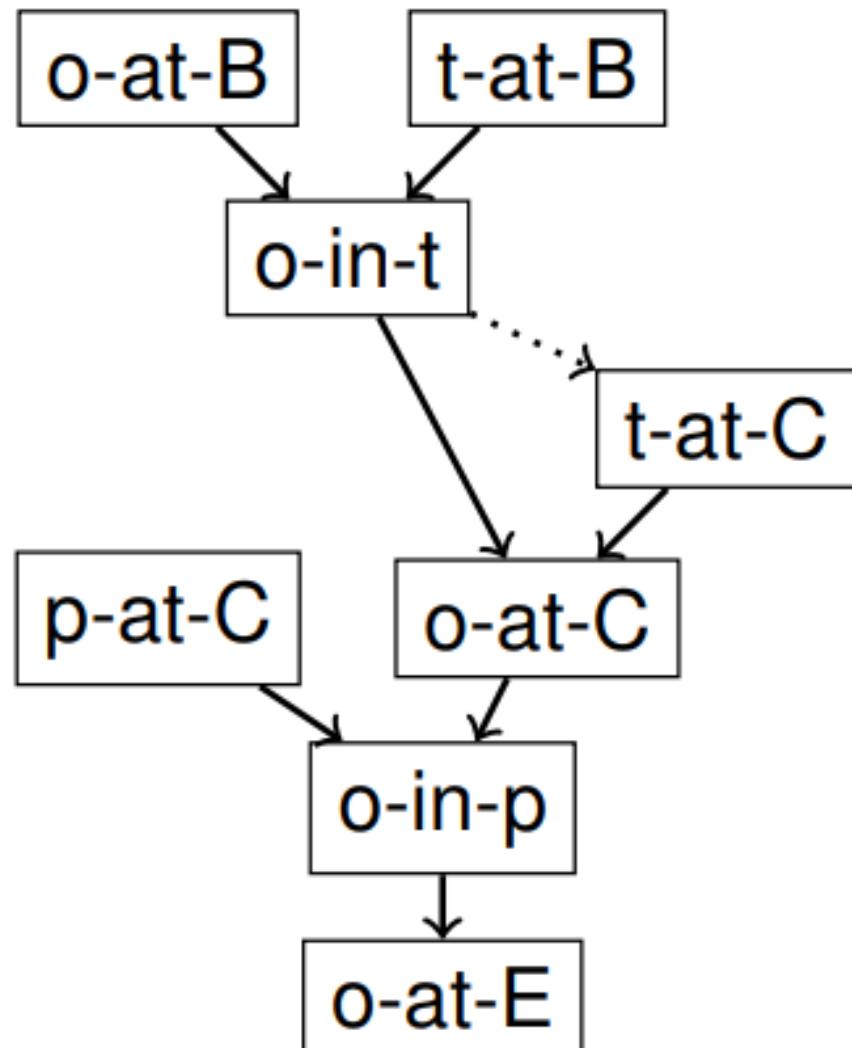
# The Good and The Bad

- Faster planning as the planner needs to explore the search space to a lower depth.
- Can lead to poor-quality plans, sometimes landmarks need to be undone and achieved again if tackled in the wrong order.
- Incomplete in problems with dead ends.



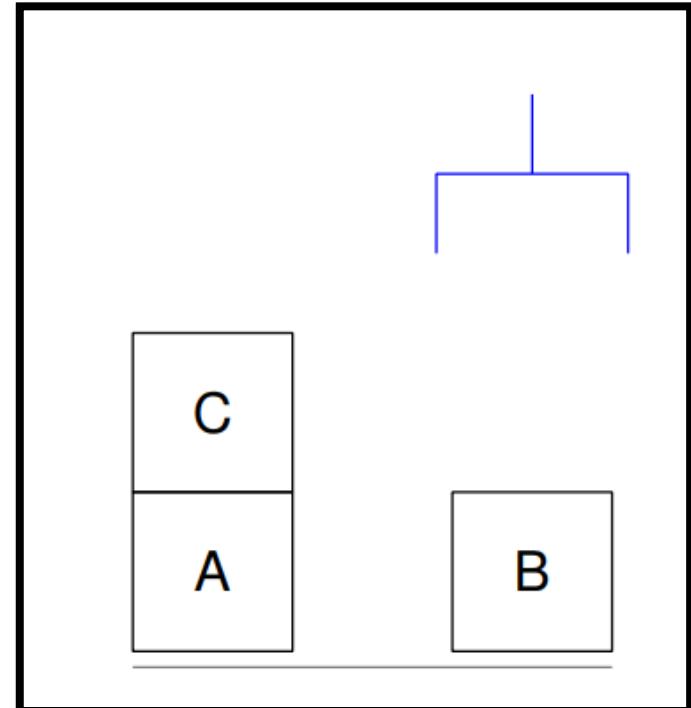
# Landmarks as Heuristics

- Number of landmarks still to be achieved is a heuristic estimate.
- LM-Count: A *path-dependent* heuristic.
  - We can't tell which landmarks have been achieved just from what's true in the state.
- Adopted by LAMA (winner of the IPC 2008 Winner Satisficing Track)



# Path-Dependent Heuristic

- Consider the situation shown here...
  - Did we achieve the ‘Holding B’ landmark?
- Achieved landmarks are a function of the path taken, not an observation of the state.
- Hence LM-Count is an inadmissible heuristic.
  - One action can achieve more than one landmark.

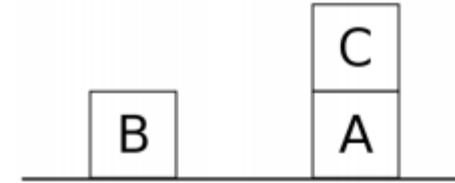


# Computing LM-Count

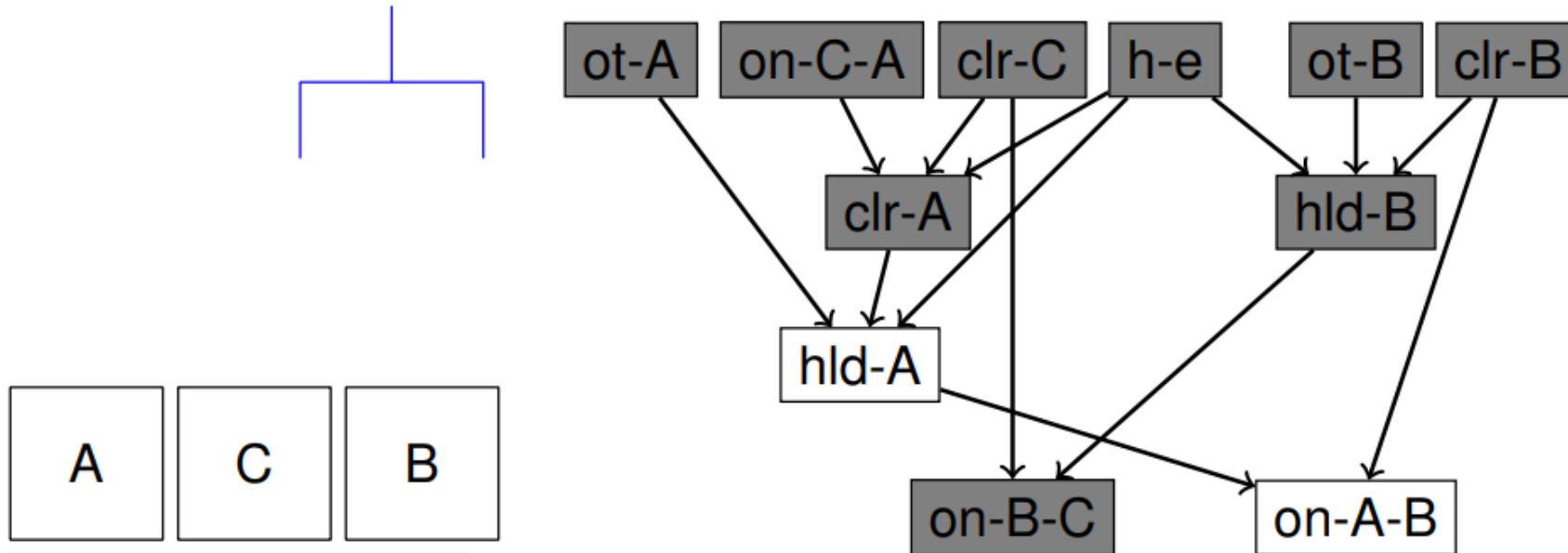
- $H(s,p) = (L \setminus \text{Accepted}(s,p)) \cup \text{ReqAgain}(s,p)$ 
  - For state  $s$ , path  $p$
- **L: set of all landmarks discovered for the problem;**
- **Accepted(s,p): Accepted landmarks:**
  - Landmark is accepted if it's true in  $s$  and all its predecessors in the landmark graph are accepted.
- **ReqAgain(s,p): Landmarks that we've seen but know we need to see again. L is required again if:**
  - Landmarks is false and it is a goal;
  - Landmarks is false in  $s$  and is a greedy-necessary predecessor of landmark  $m$  (must be true the step before  $m$ ), which is not accepted.

# Example

C/O Erez Karpas



- In the Sussman anomaly, after performing: Pickup-B, Stack-B-C, Unstack-B-C, Putdown-B, Unstack-C-A, Putdown-C



- on-B-C is a *false-goal*, and so it is required again

# Admissible Landmark Heuristics

- Many of the most successful modern heuristics for optimal planning are landmark based.
- Not covered here but if you're interested this tutorial is a good start:
  - [http://videolectures.net/icaps2010\\_sanner\\_lhsp/](http://videolectures.net/icaps2010_sanner_lhsp/)
- And also covers what we've seen here on landmarks and LAMA's Heuristic.

# Classical Planning Searching with Landmarks

6CCS3AIP – Artificial Intelligence Planning

Dr Tommy Thompson



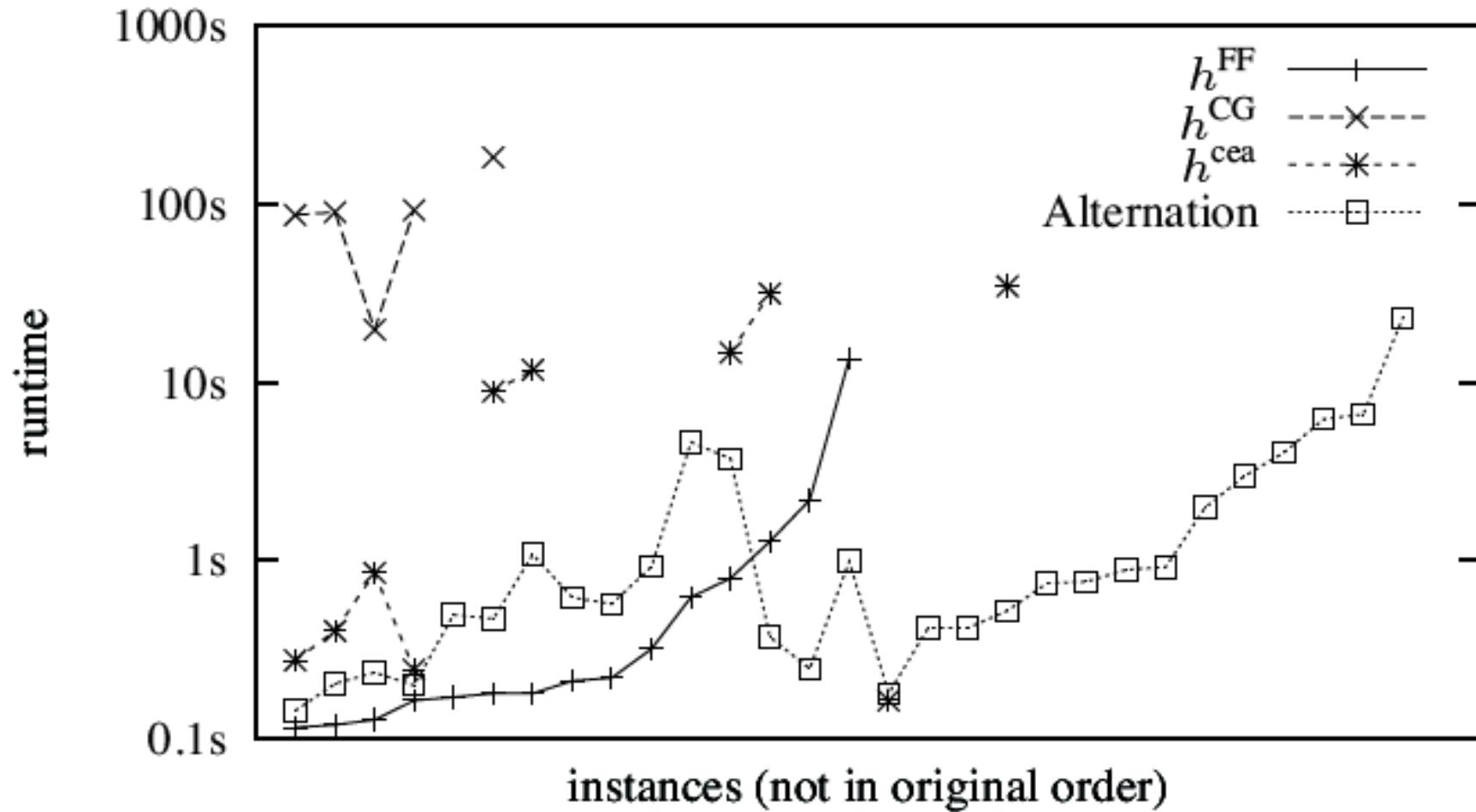
# Classical Planning Dual Heuristics & Local Search Optimisation

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Combining heuristics

- Q: What if we had two heuristics,  $h_1(S)$  and  $h_2(S)$ , both estimating the distance and/or cost to the goal?
- A: Could have two open lists, one for each heuristic, and alternate between them
  - Expand node from open list 1
  - Expand node from open list 2
  - Expand node from open list 1...
- Evaluate successors with both heuristics and put on both open lists



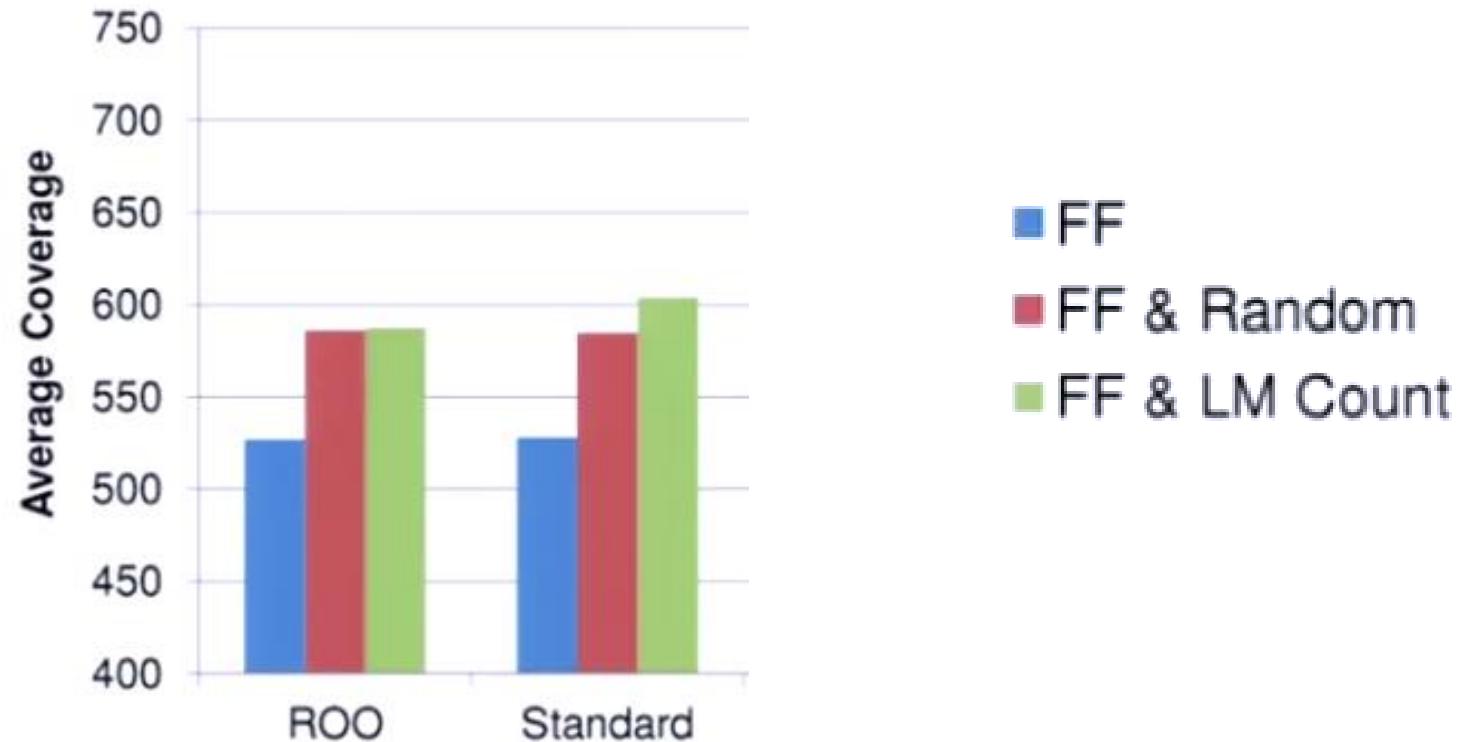
# What is the second heuristic doing?

- A mixture of two things:

## 1. If one heuristic is on a plateau:

- Plateau successors put on both open lists;
- Second open list will **order them by second heuristic** so will be guiding search on the plateau

## 2. In any case – diversification by avoiding expanding just the states one heuristic likes.



# Local Search

- In four bullet points:
  - 1) Start with a state  $S$
  - 2) Expand  $S$ , generating successors
  - 3)  $S' =$  one of these successors
  - 4)  $S = S'$  and repeat
- Heuristics are used at (3)
  - e.g. choose the best successor
  - Known as **gradient descent** (or **hill climbing**).

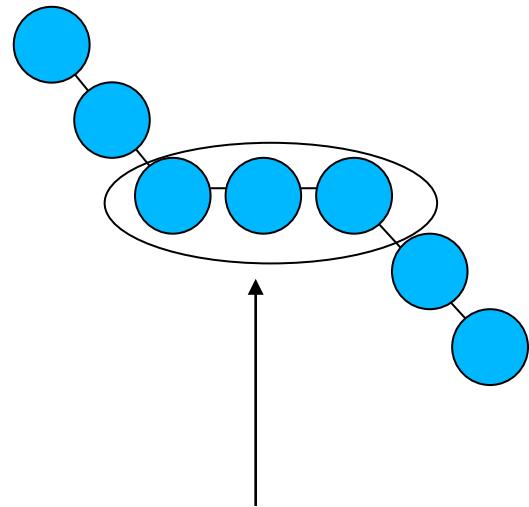
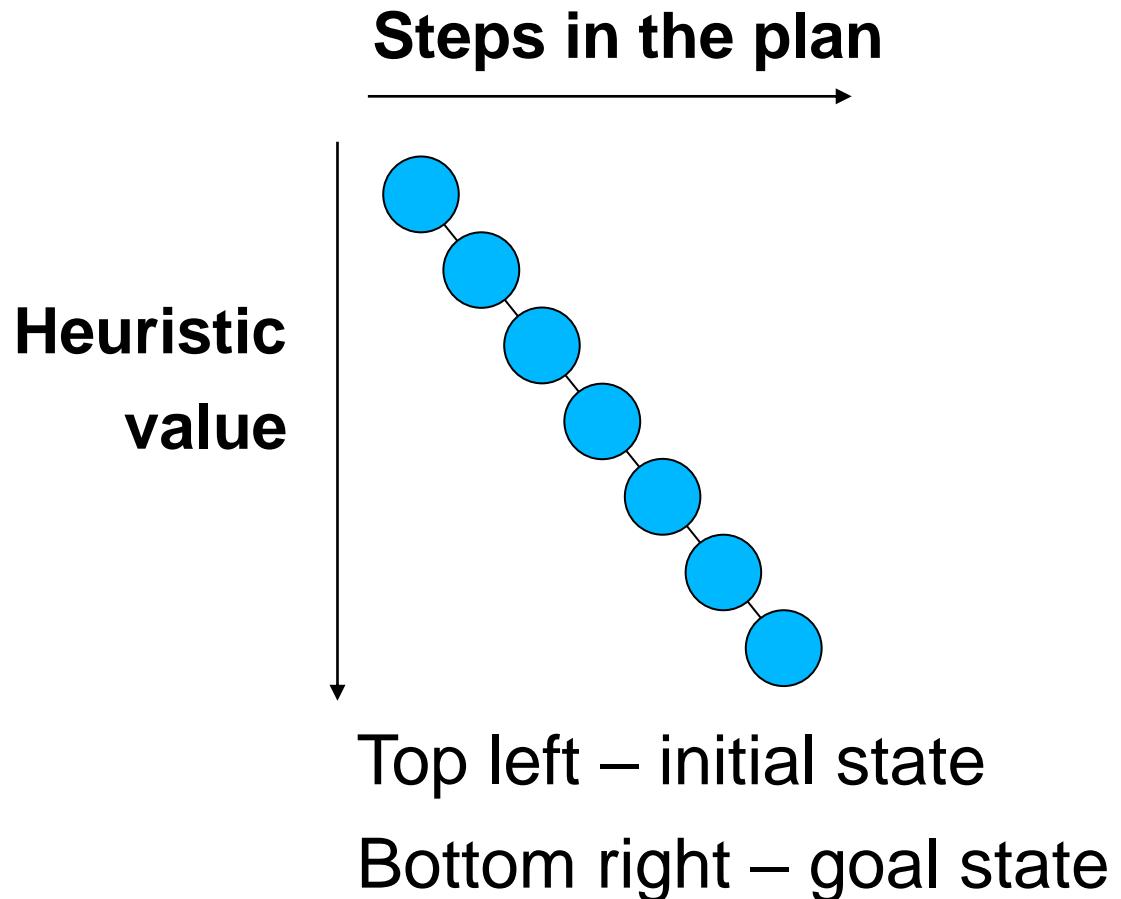
# The Trouble with Heuristics

- **The RPG heuristic is not perfect**
  - Few are, otherwise the problem would be easy!
  - ... or the heuristic would be very expensive to compute
- **Making the right move doesn't always lower the heuristic value:**
  - It can stay the same; or,
  - It can go higher.
- **Or, worse, the move with the best heuristic value can be a really bad decision:**
  - With local search, this can lead to no solution being found.

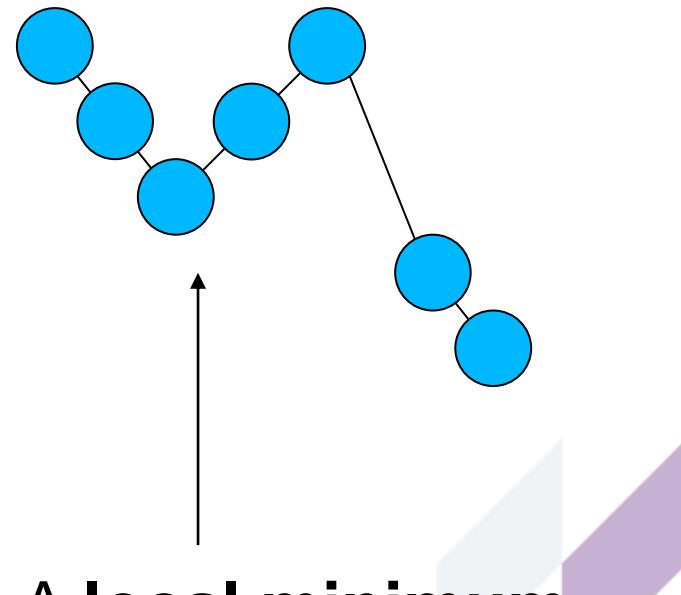
# Snags with EHC

- Sometimes can fail to find a solution:
  - The heuristic can lead it to **dead ends**
- If FF fails, resorts to systematic best-first search from the start:
  - Priority queue of states (lowest  $h(S)$  first)
  - Expand the next state each time
  - Add successors to the queue, and loop
- Also, searching on plateaux is expensive:
  - Systematic search on the part where the heuristic is worst

# The Search ‘Landscape’

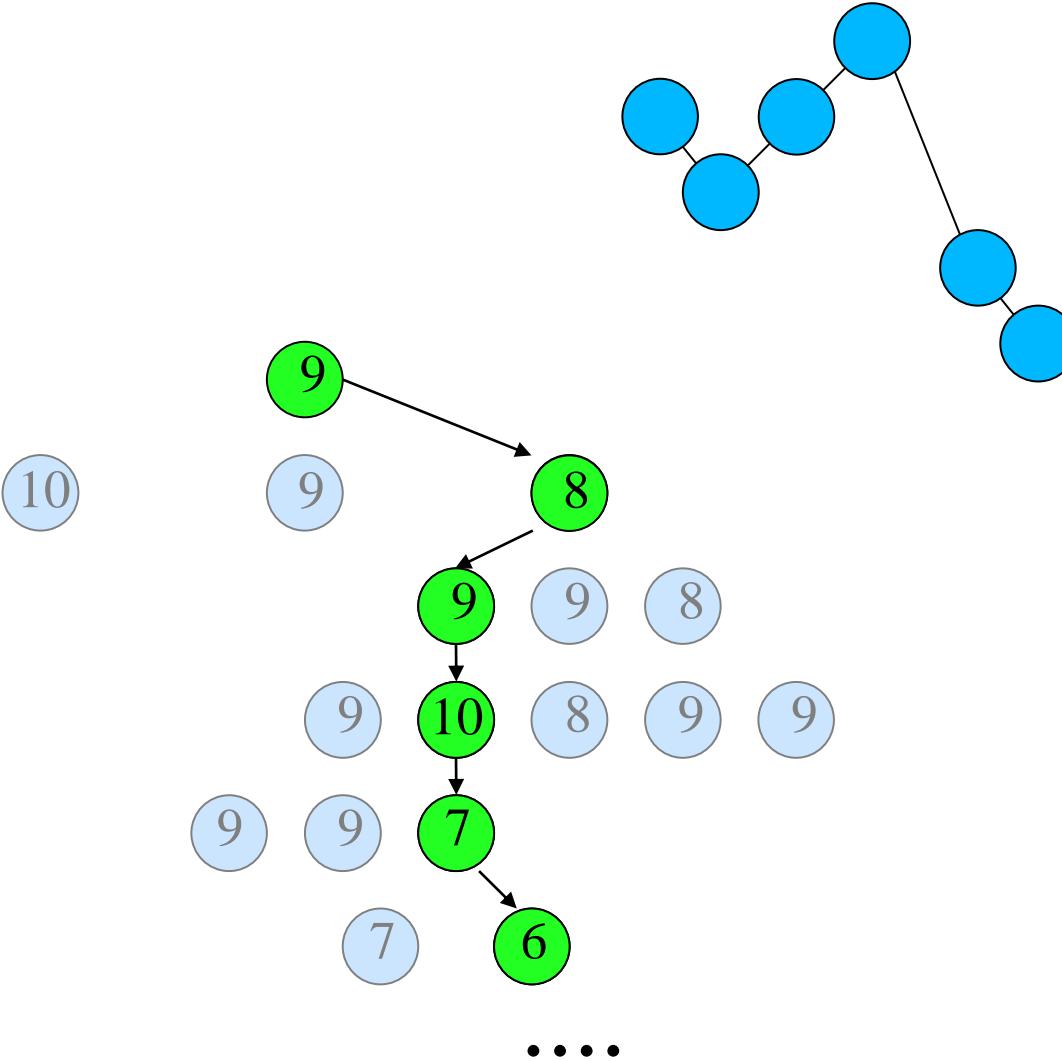


A **plateau** – all heuristic values the same, the right step doesn't seem better

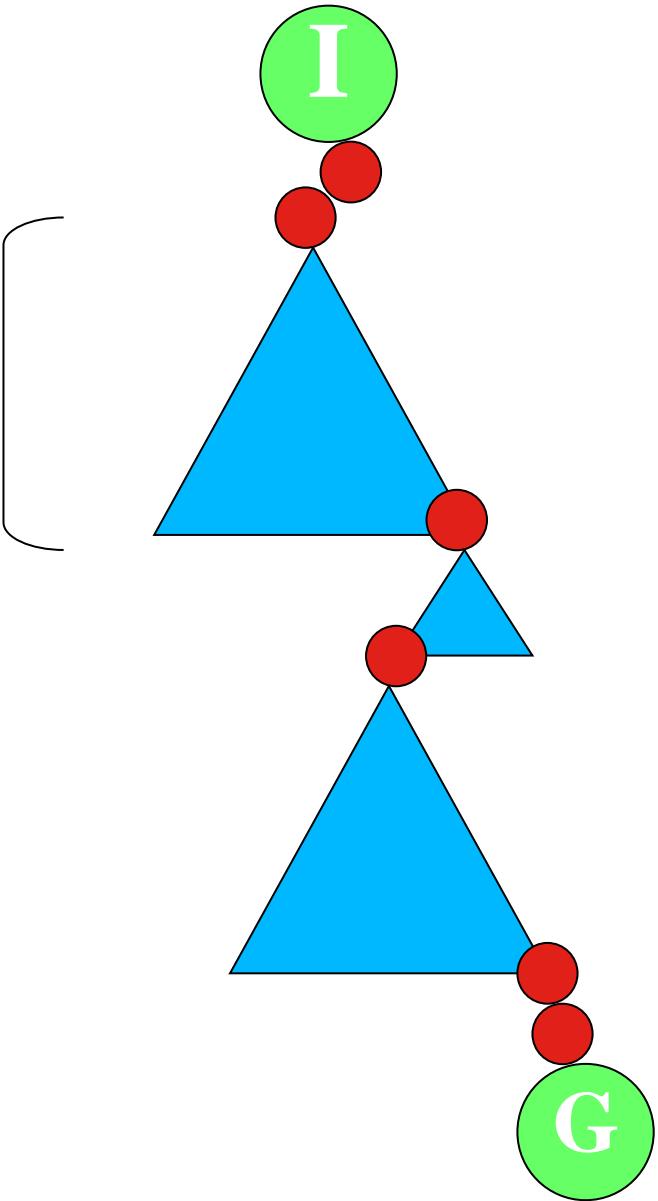


A **local minimum** – the right step seems worse

# EHC, Revisited

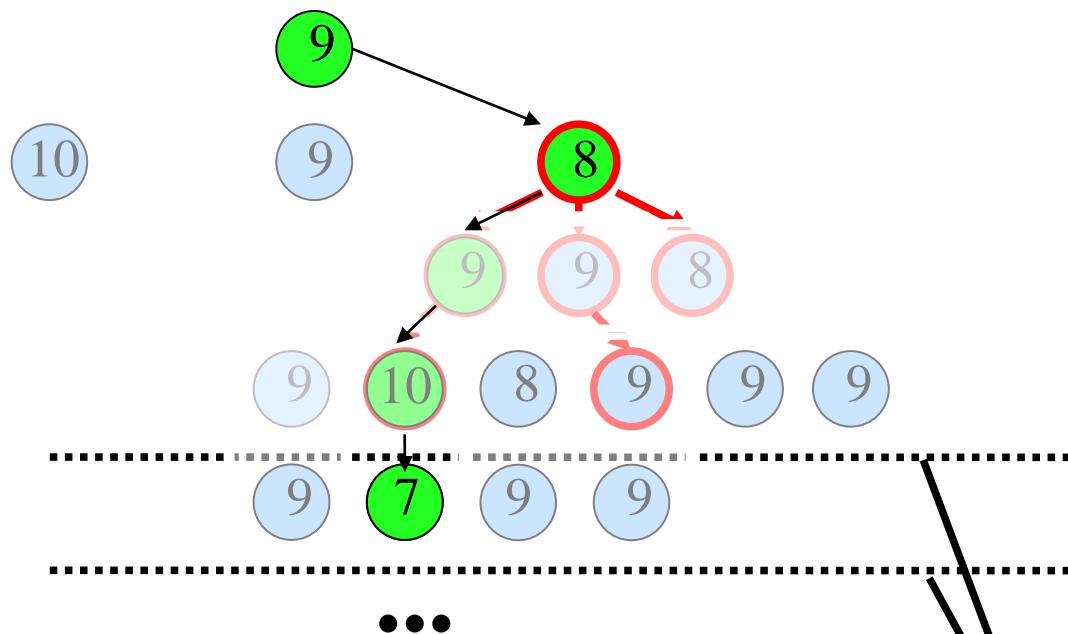


**1000s  
of  
states**

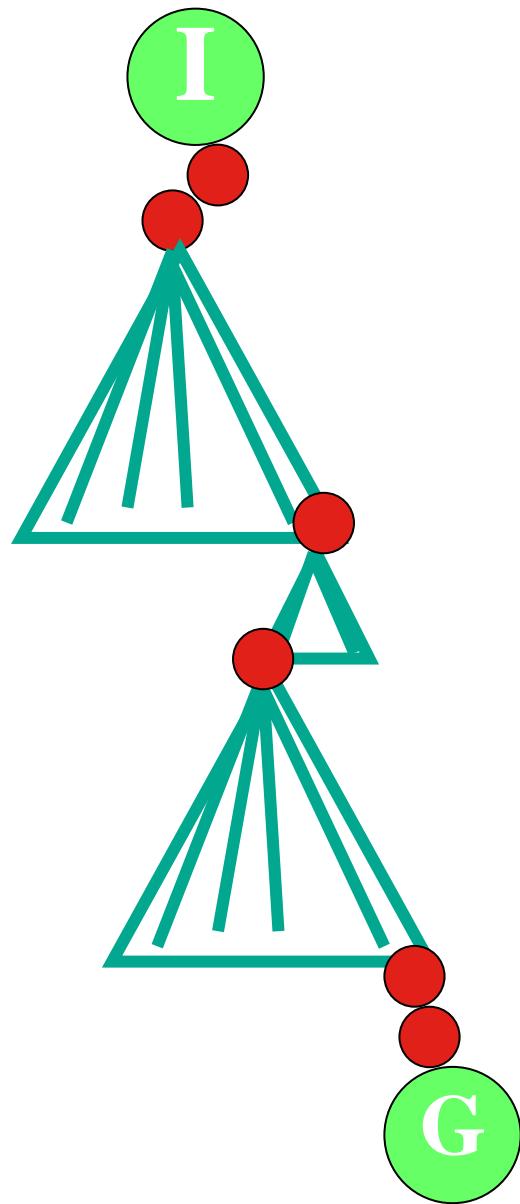


# Identidem – an alternative to EHC

- Replace systematic search on plateaux with local-search with restarts.
- As in EHC: record the best state seen so far.
  - When expanding a state  $S$  at the route of a plateau:
    - Choose one successor, **at random**, even if it's not better than the best seen so far (it's not as we're on a plateau);
    - If more than  $d$  steps since the best state: jump back to the best and search from there again:
    - $d$  is the **probe depth bound**



probe depth  
bound  
increased to  
3

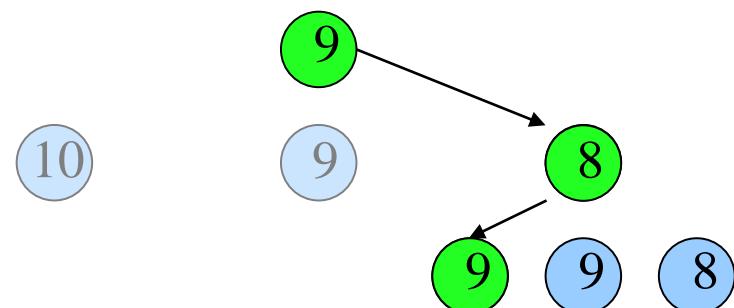


# Local Search: Two key terms

- **Diversification** – try lots of different things, to try to brute-force around the weaknesses of the heuristic
- **Intensification** – explore lots of options around a given state (e.g. near to where the goal might be)
- **Local search is a balancing act**
  - Long probes = good for diversification
  - Short probes = good for intensification

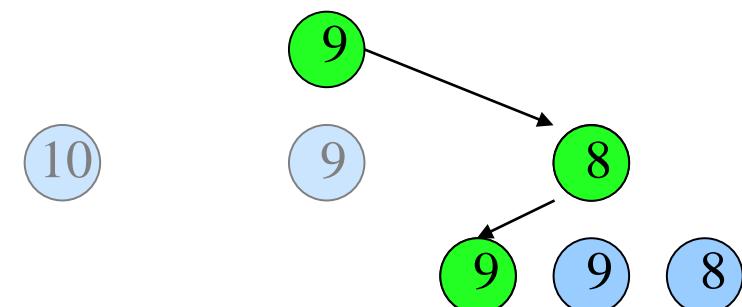
# Different Neighbourhoods - FF

- When expanding a state,  $S$ , a neighbourhood function is used.
  - The simplest neighbourhood: all states reachable from  $S$  by applying a single action.
  - Disadvantage: not all successors are interesting to consider
- In FF, when performing EHC: the helpful actions neighbourhood is a subset of these:
  - Those actions reached by applying actions related to the relaxed plan to the goal from  $S$  gives an interesting subset of actions



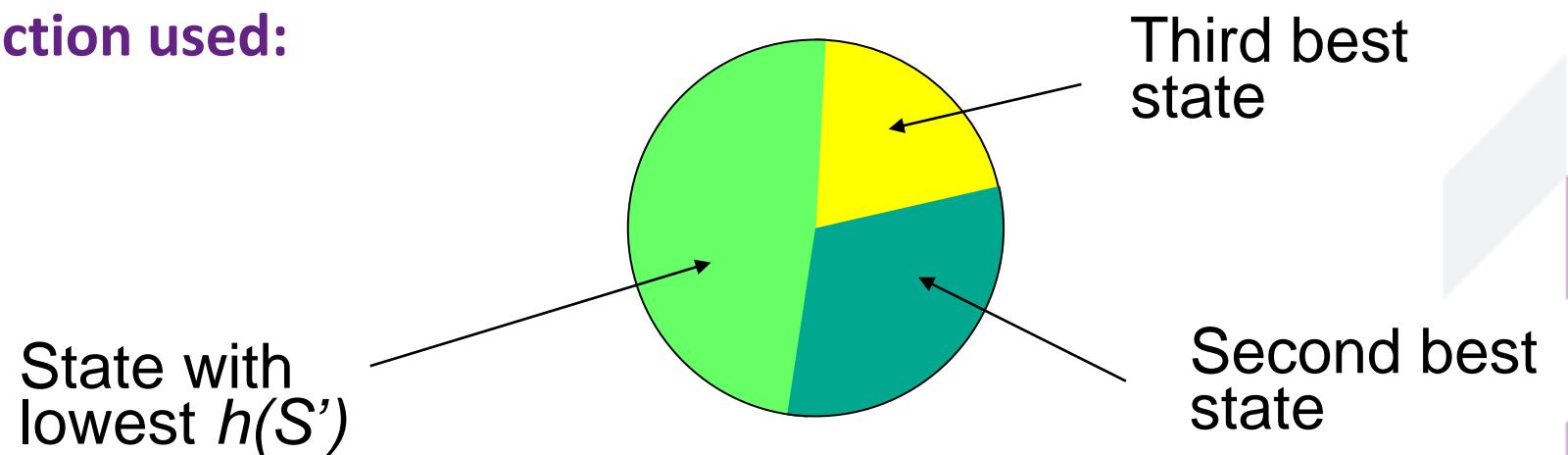
# Different Neighbourhoods - Identidem

- **Identidem only ever chooses a single successor:**
  - Many are evaluated, only one is chosen.
- **Evaluating successors carries a cost;**
  - **Solution:** The random subset neighbourhood:
    - Pick a subset of the neighbourhood, only evaluate those.
- **Ties in well with restarts: different subset chosen each time a state is visited**



# Successor Selection

- In EHC, the ‘best’ successor was always chosen;
  - Imperfect heuristic -> wrong decisions -> dead ends.
- In Identidem: roulette selection used:



- Make a random choice, biased by heuristic values

# Does Identidem perform better than FF?

- **Short answer – yes!**
- **Slightly longer answer – most of the time:**
  - Systematic search better in some domains
  - Sometimes the **parameters** need changing:
    - Random sample size, depth bound, whether to use roulette selection...

# Systematic Search vs Local Search

- Good for ‘puzzles’ with lots of bad moves (keeping one successor = probably a wrong move)
- Good at proving optimality.
- Are provably complete – will find a solution if one exists (or die trying)
- Generally quicker at finding some solution
- Using them repeatedly usually gets near-optimal solutions fairly quickly, but no proof
- Incomplete, but can always run systematic search afterwards (e.g. EHC then Best-First)

# Classical Planning Dual Heuristics & Local Search Optimisation

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



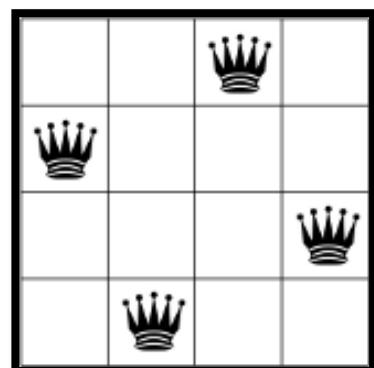
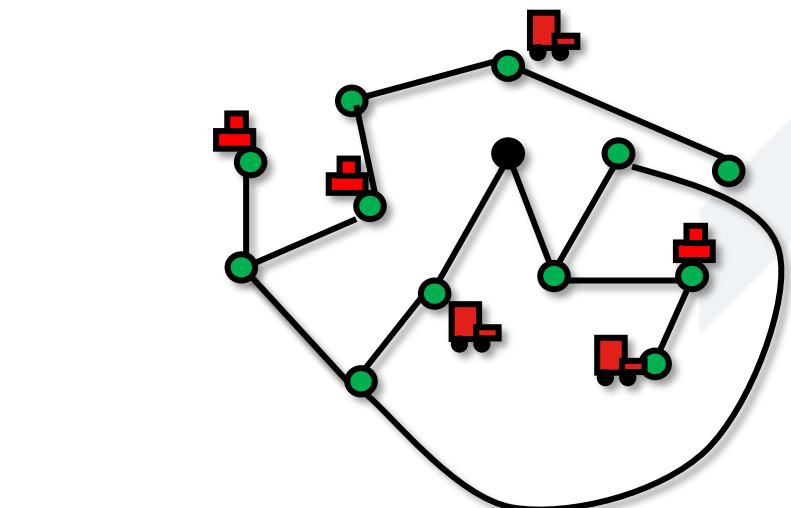
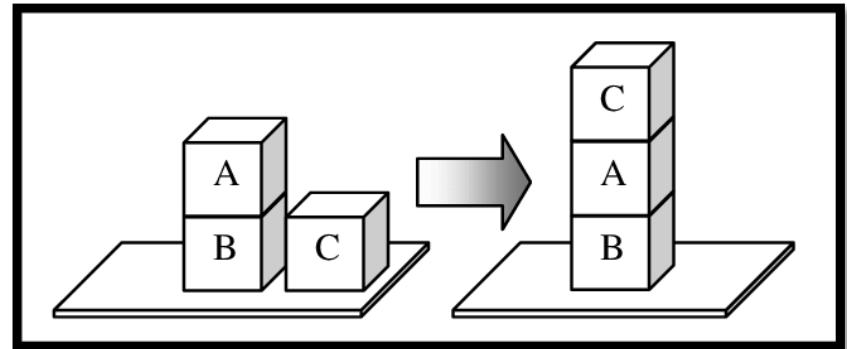
# Classical Planning Building Heuristics from RPG

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Domain Independent Heuristics

- Building heuristics that can adapt to different domains/problems.
- Not reliant on specific information about the problem.
- We analyse aspects of the search and planning process to find potential heuristics.



# Delete Relaxation

- **Delete Relaxation**

- Estimate cost to the goal by **removing negative effects** of actions.
- i.e. any PDDL effect that removes a fact is no longer considered.

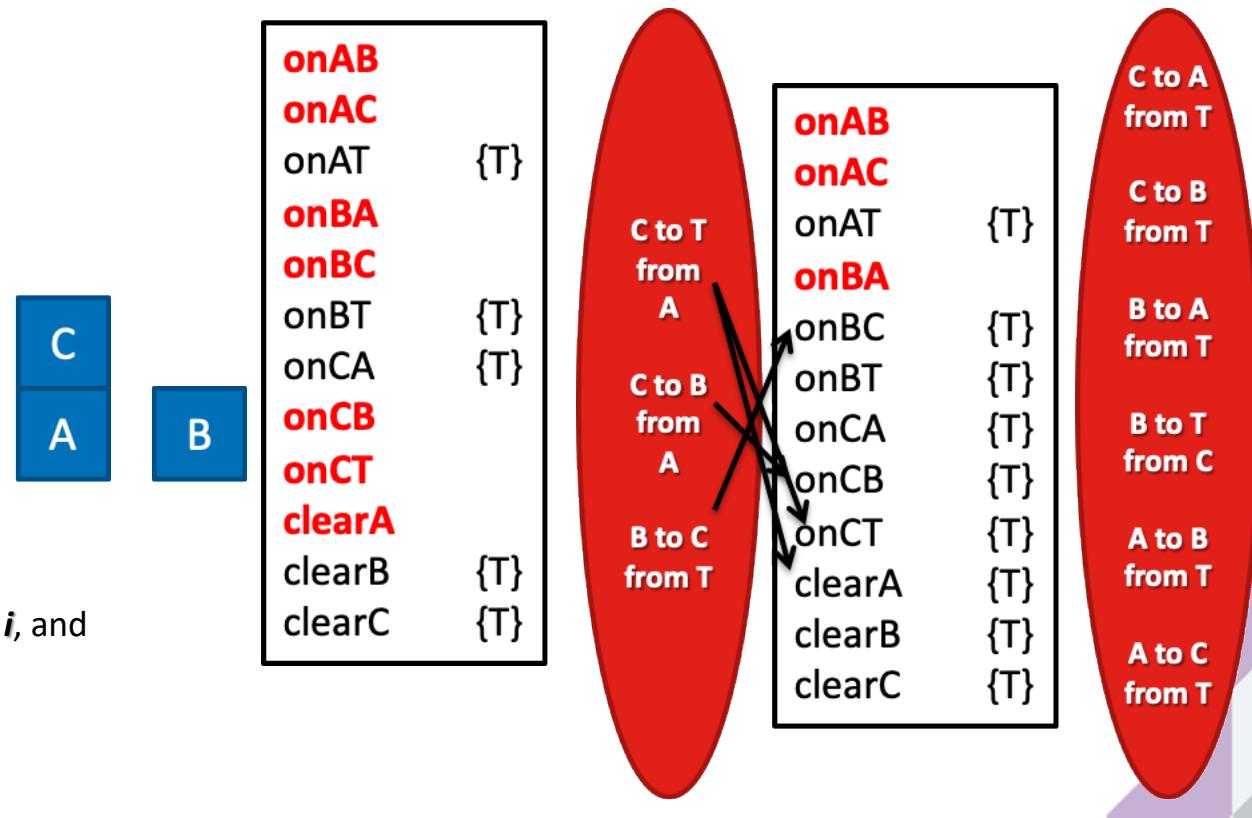
- **Example: FreeCell Solitaire**

- Free cells, tableau positions **remain available after moving cards onto them**.
- Cards **remain movable** and **remain valid targets for other cards** after moving cards on top of them.



# Relaxed Planning Graph Heuristic

- Relaxed Planning Graph (RPG) heuristic creates a simple layered approach to exploring the state space.
- Inspired by the GraphPlan system
  - GraphPlan covered later in this module.
  - For reference, see (Blum & Furst, 1995)
- The relaxed solution plan  $P' = \langle O_0, O_1, \dots, O_{m-1} \rangle$ 
  - Where each  $O_i$  is the set of actions selected in parallel at time step  $i$ , and  $m$  is the number of the first fact layer containing all goals
  - $h(s) := \sum |O_i|$

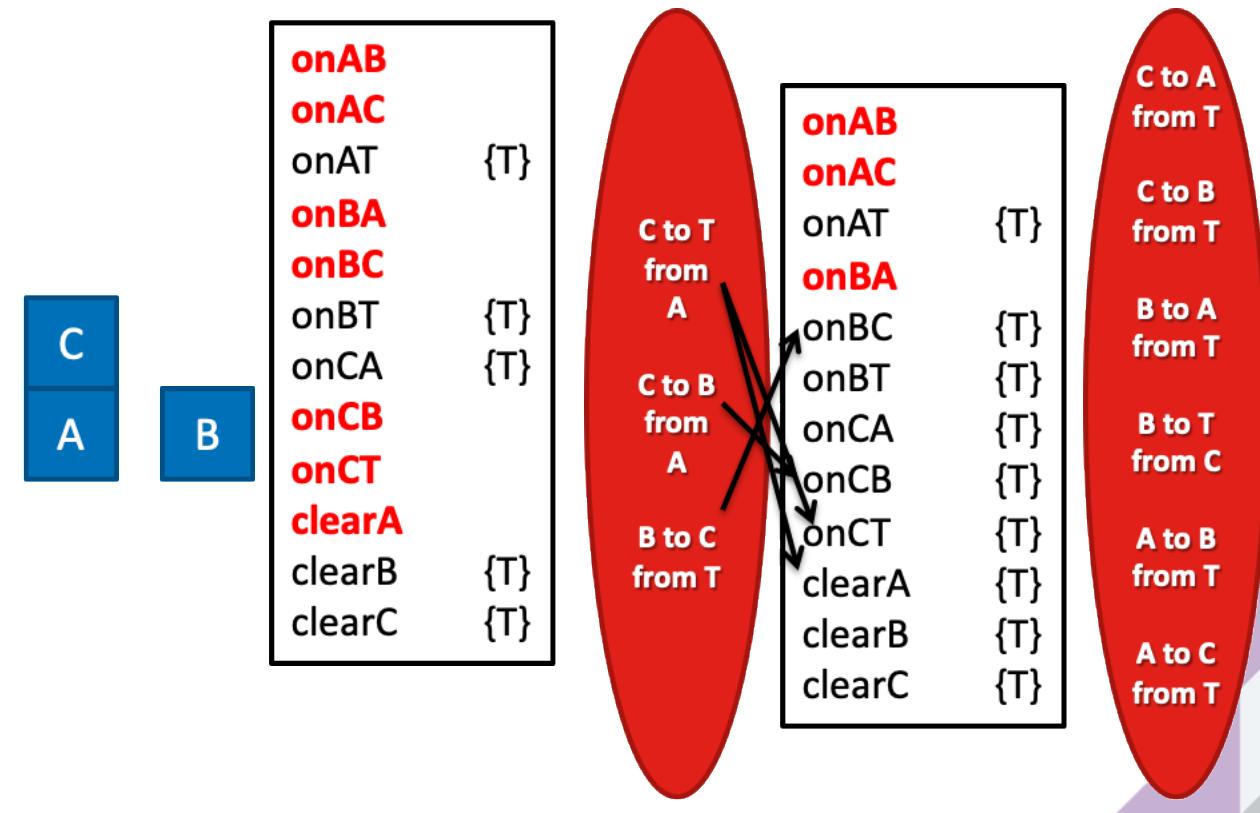


Blum, A. L., & Furst, M. L. (1995). Fast planning through planning graph analysis.

In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI95), pp. 1636

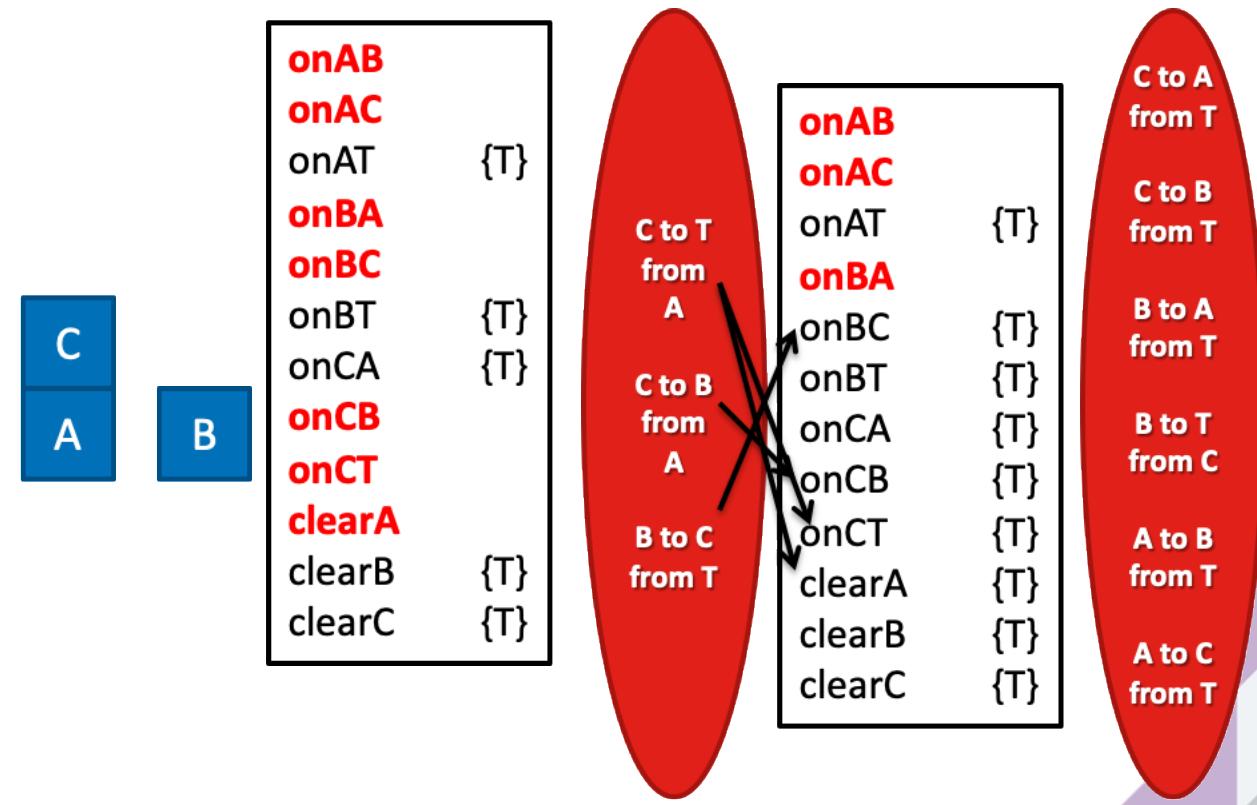
# What Else Can We Gain From Relaxation?

- Relaxed Planning Graph (RPG) heuristic gives us useful information from the relaxed problem.
- The relaxed solution is one useful piece of information we can grab from the planning graph.
- But is there any other useful information we can gather here?



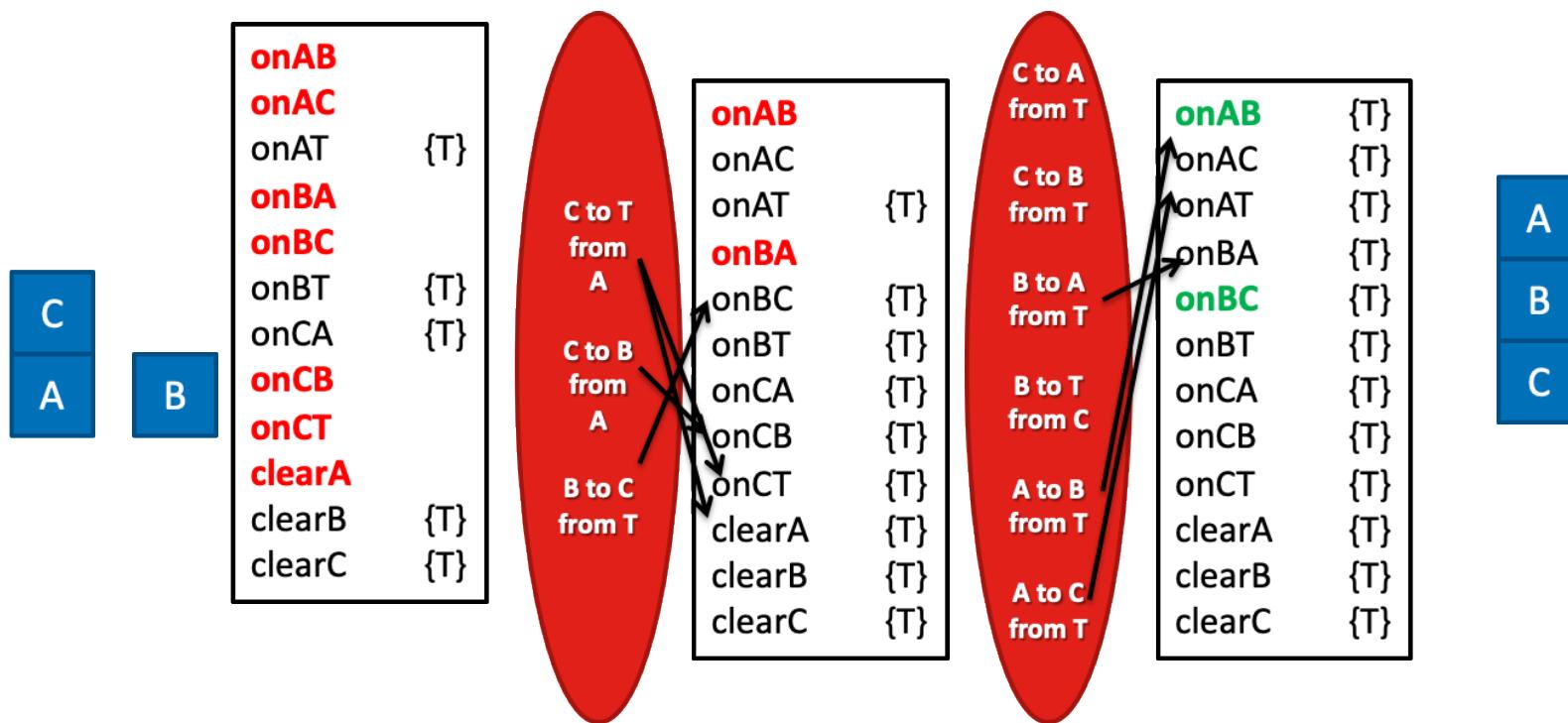
# What Else Can We Gain From Relaxation?

- One thing we have ignored throughout is the cost of actions involved.
- If we factor the costs of actions in the planning graph, we can learn more about how close a given state is to the goal.
- This works even in state spaces with uniform action costs.
- Let's consider two heuristics:  $h^{\text{Add}}$  and  $h^{\text{Max}}$ .



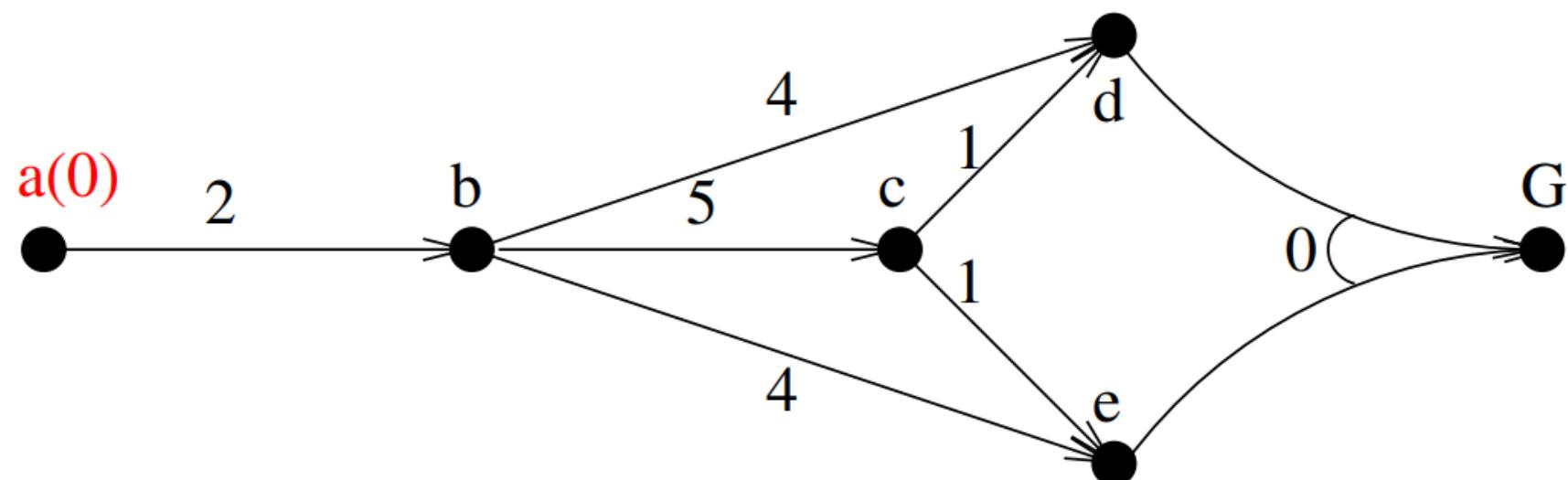
# Understanding Costs

- Consider the cost of the actions in order to satisfy a fact in our fact layers.
- If the facts are already achieved, the value is 0.
- If the facts are not achieved, the value is  $\infty$



# Understanding Costs

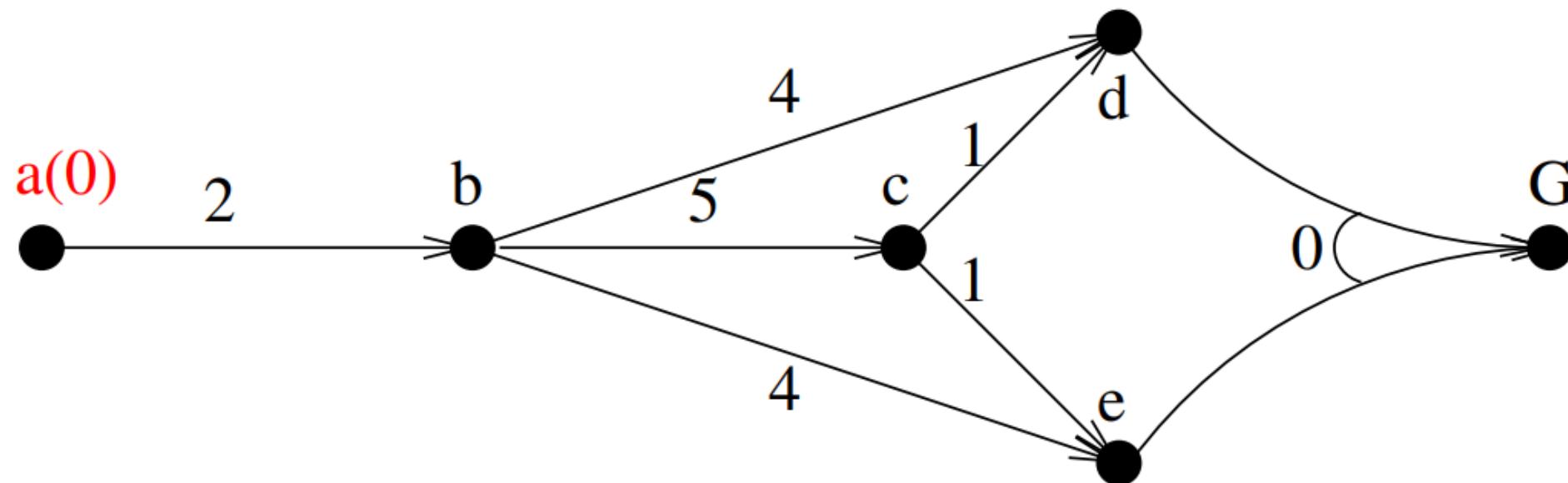
- Consider the cost of the actions in order to satisfy a fact in our fact layers.
- If the facts are already achieved, the value is 0.
- If the facts are not achieved, the value is  $\infty$



[ $a = 0, b = \infty, c = \infty, d = \infty, e = \infty, G = \infty$ ]

# $h^{\text{Add}}$ (additive) heuristic

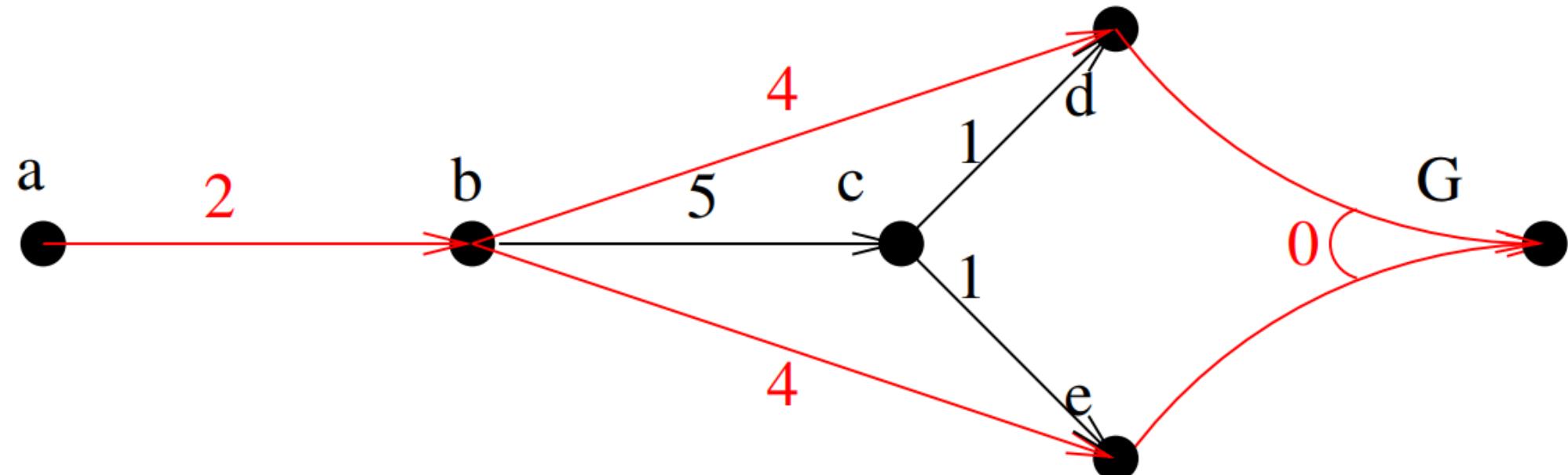
- $h^{\text{Add}} = \text{We sum the value of all costs to reach a given fact.}$
- Inadmissible, given emphasis on goal independence.



[ $a = 0, b = 2, c = 7, d = 6, e = 6, G = 12$ ]

# $h^{\text{Add}}$ (additive) heuristic

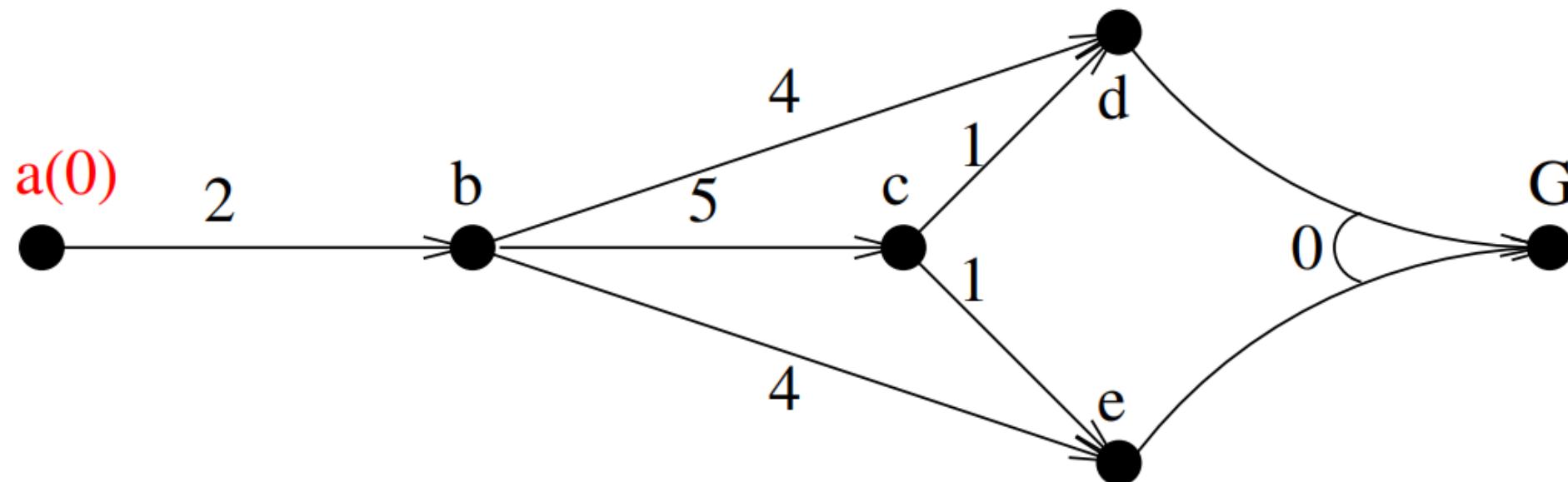
- $h^{\text{Add}} = \text{We sum the value of all costs to reach a given fact.}$
- Inadmissible, given emphasis on goal independence.



$[G = (d,e)] \rightarrow \text{relaxed plan cost is 10}$

# $h^{\text{Max}}$ heuristic

- $h^{\text{Max}} = \text{The most expensive path to reach a given fact.}$
- Admissible, given on most expensive path to achieve G.
- When costs are uniform (i.e. = 1),  $h^{\text{Max}}$  is simply number of RPG layers.



[ $a = 0, b = 2, c = 7, d = 6, e = 6, G = 6$ ]

# Classical Planning Building Heuristics from RPG

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Classical Planning Non-Forward Search Graphplan

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Classical Planning: Material Overview

## • ~~Classical Planning: The fundamentals.~~

- Non-Forward Search

## • ~~Improving Search~~

- Heuristic Design (RPG/LAMA)
- Dual Openlist Search

- Graphplan
- SAT Planning
- POP Planning
- HTN Planning

## • ~~Optimal Planning~~

- SAS+ Planning
- Pattern Databases

- Planning Under Uncertainty



# Fundamentals: Forward Search

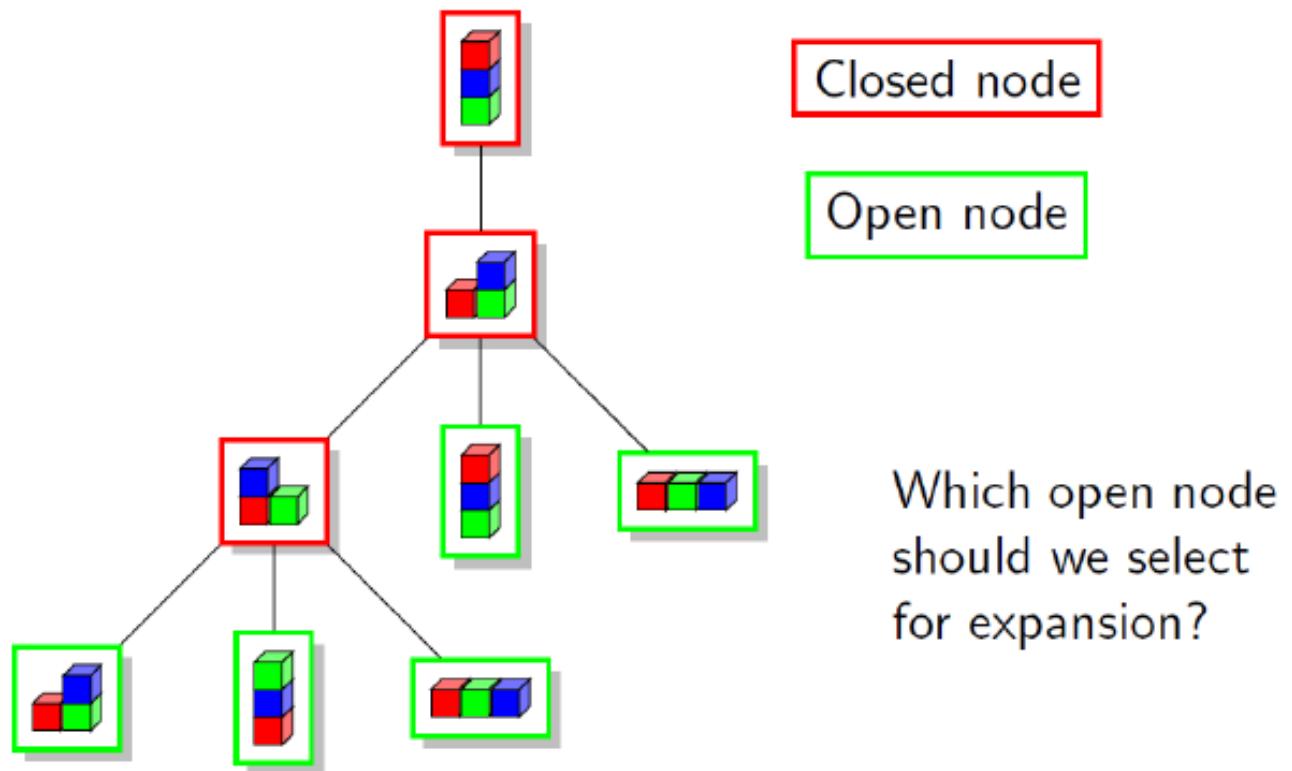
- **Definition of the problem – PDDL**

- **Forward Search**

- Breadth/Depth First

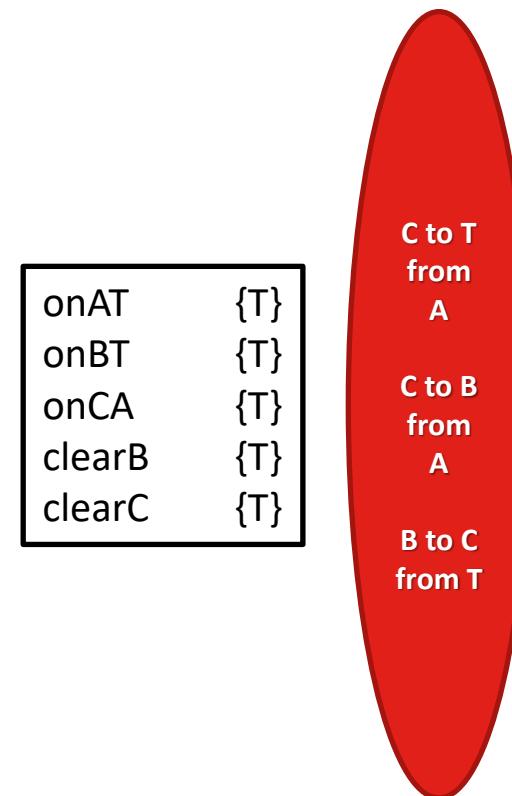
- **Introducing Heuristics**

- Best First Search/A\*



# Graphplan

- Construct a graph that encodes constraints on possible plans.
- The planning graph constrains search to a valid plan.
  - Finds shortest plans (i.e. shortest makespan).
  - Will search backwards from the end of the graph back to the initial state.
- Graphs can be built fairly quickly.
  - Graph construction is in polynomial time, but planning can be exptime.
  - Will terminate with failure if there is no plan.
- Originally developed for Graphplan planner, is the basis for the RPG heuristic.
  - A. Blum and M. Furst, "Fast Planning Through Planning Graph Analysis" [pdf], Artificial Intelligence, 90:281--300 (1997).



# Graphplan Example: Rocket Domain

- **Literals:**

- *at X Y*: X is at location Y
- *fuel R*: rocket R has fuel
- *in X R*: X is in rocket R

- **Actions:**

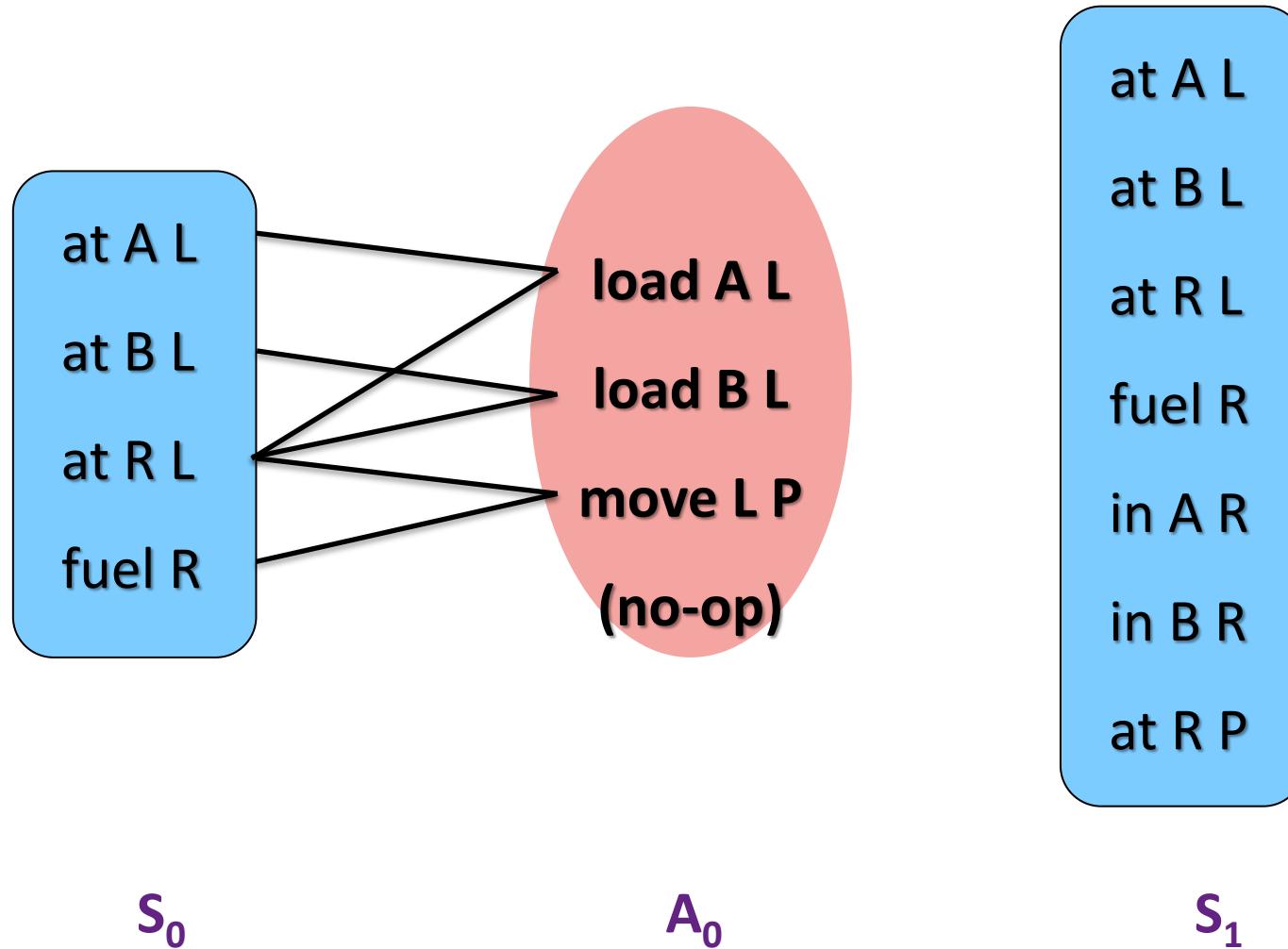
- *load X L*: load X (onto R) at location L (X and R must be at L)
- *unload X L*: unload X (from R) at location L (R must be at L)
- *move X Y*: move rocket R from X to Y(R must be at L and have fuel)

- **Graph representation:**

- **Solid black lines**: preconditions/effects
- **Dotted red lines**: negated preconditions/effects

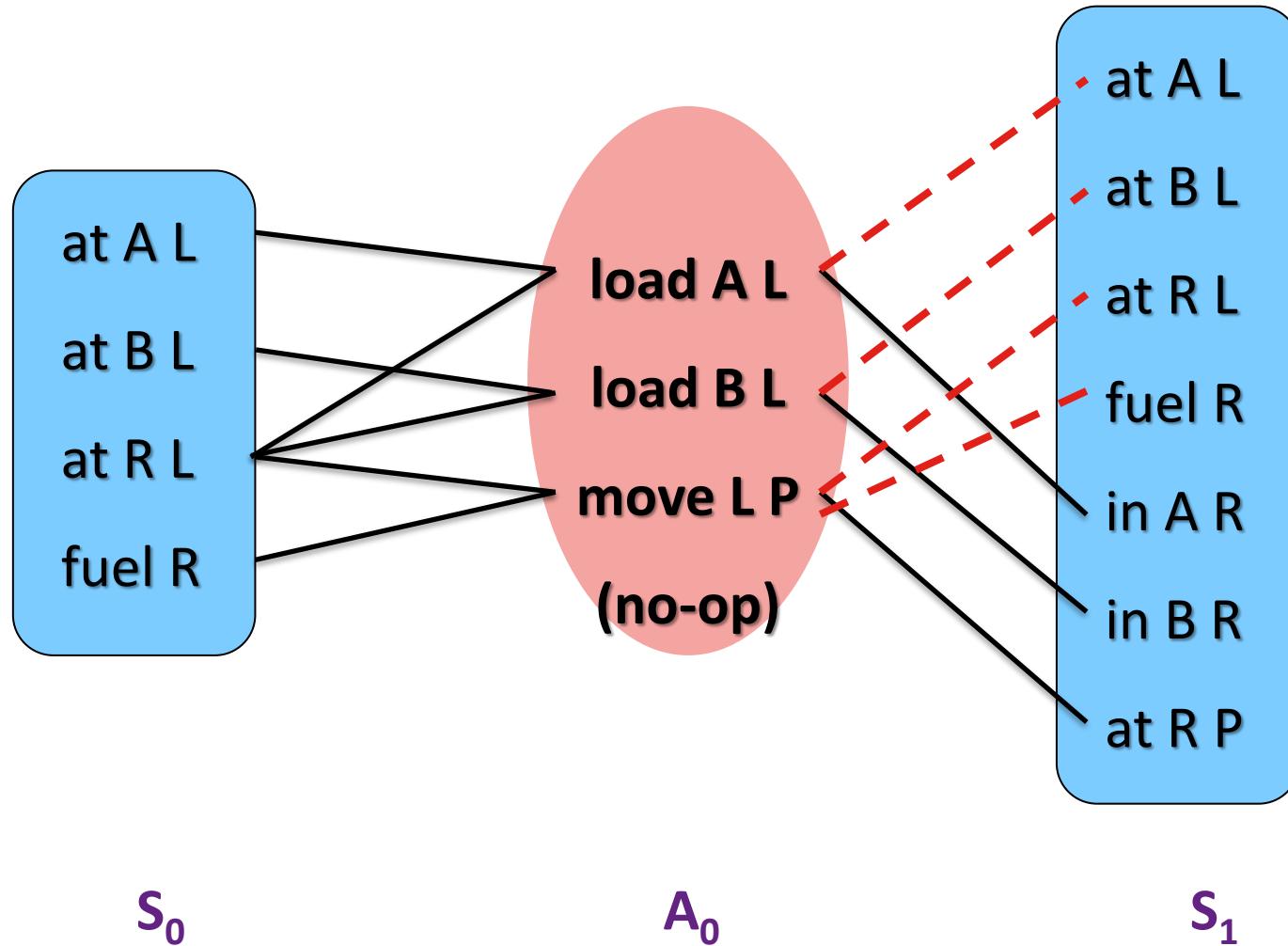


# Building a Simple Planning Graph



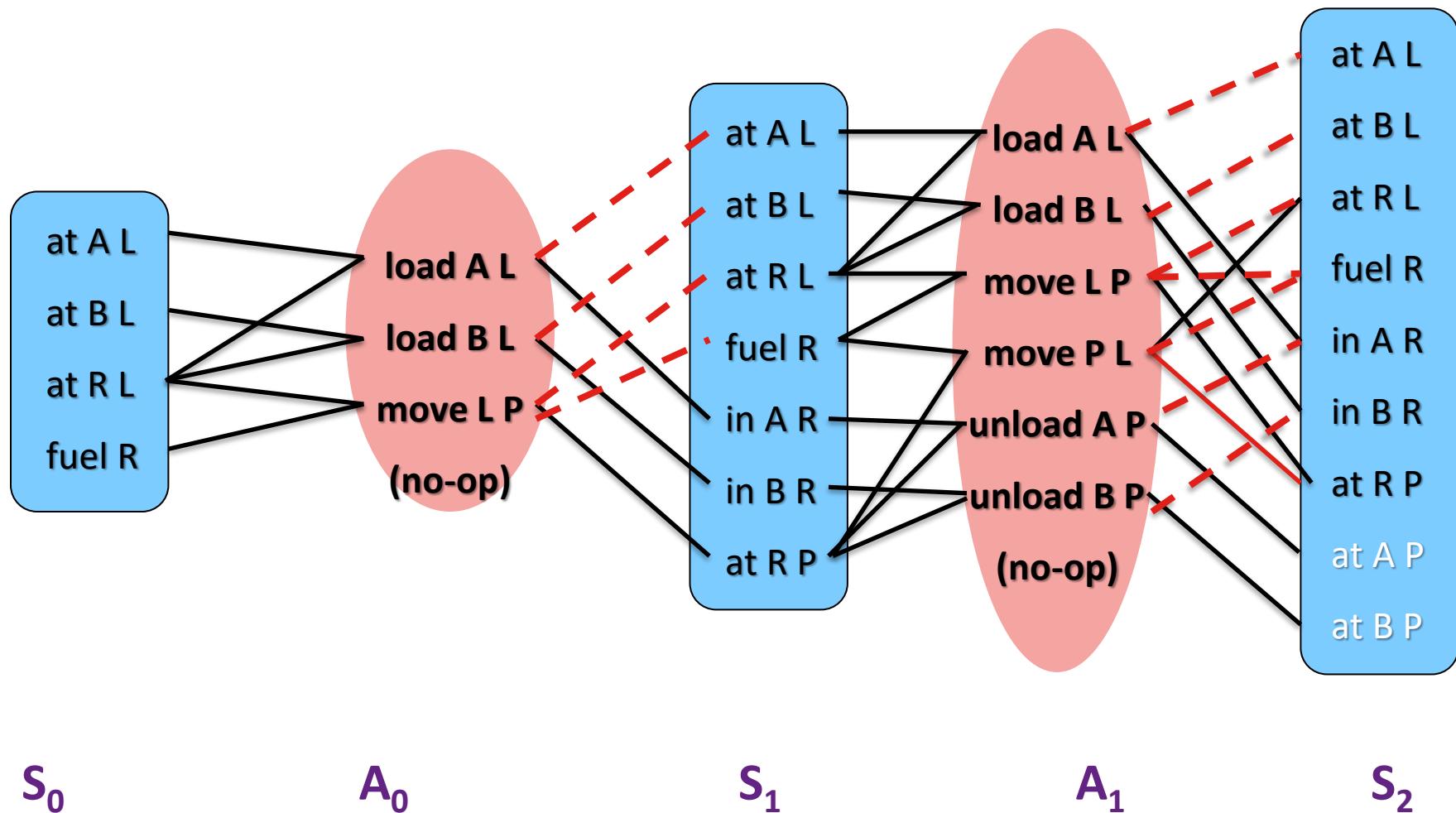
Goal  
at A P  
at B P

# Building a Simple Planning Graph



Goal  
at A P  
at B P

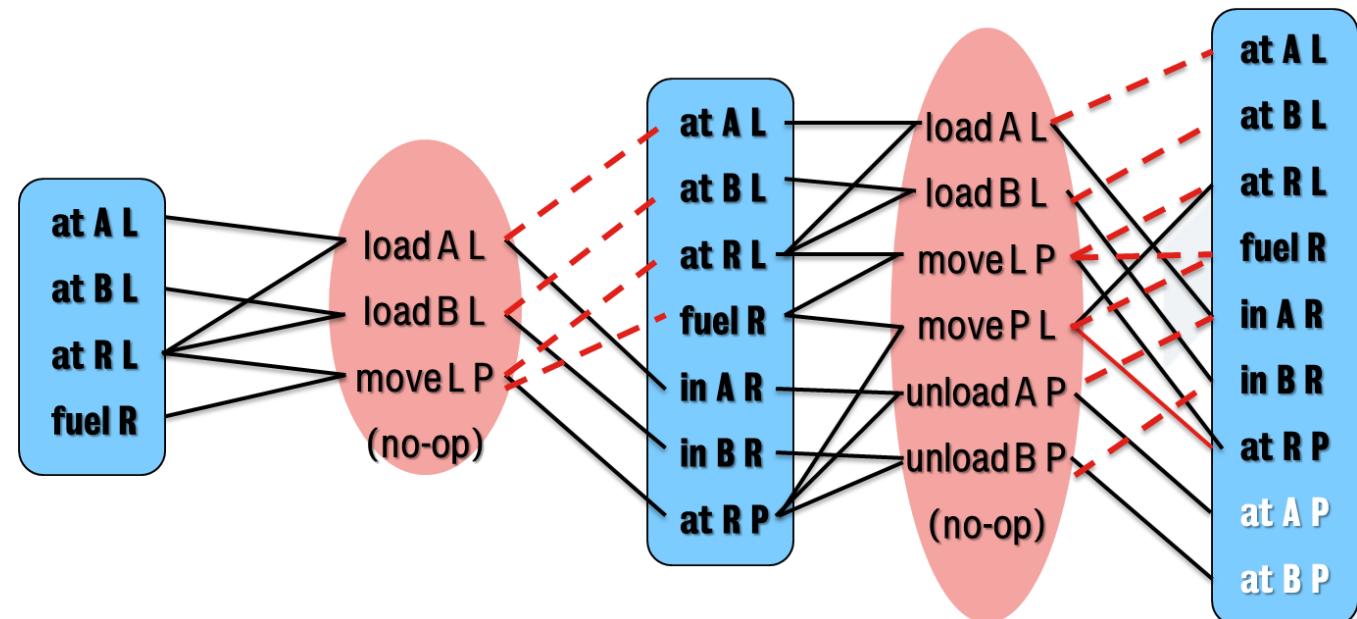
# Building a Simple Planning Graph



Goal  
at A P  
at B P

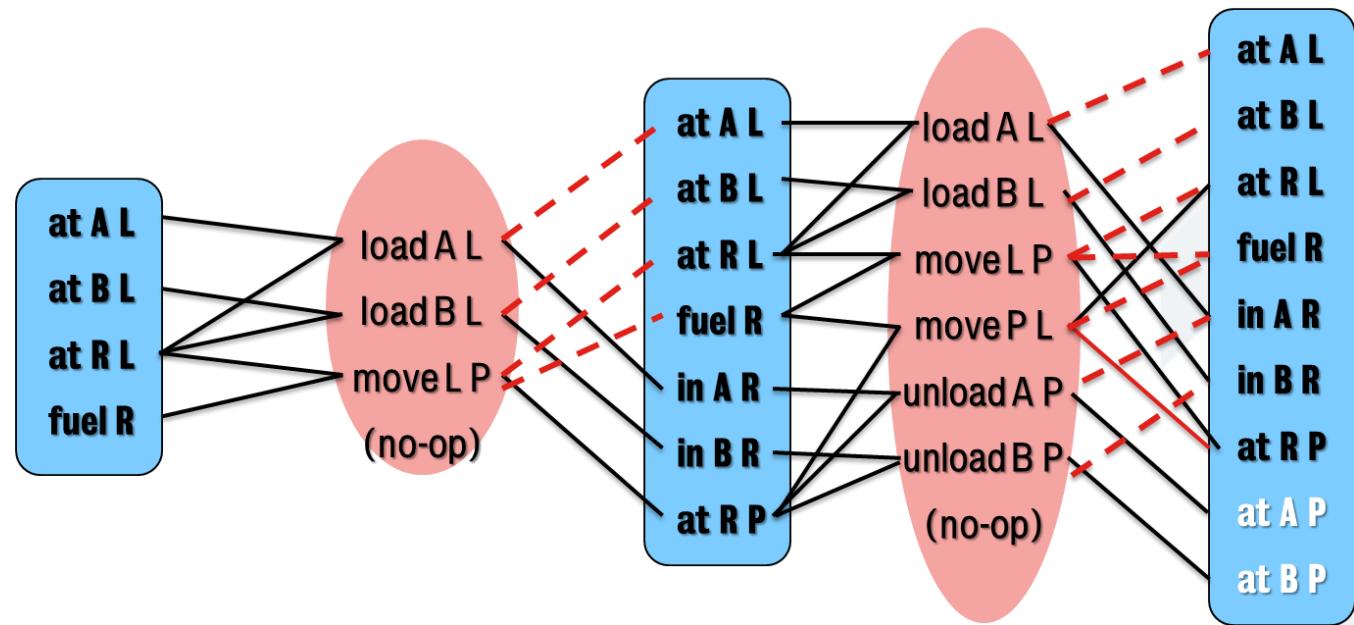
# Valid Plans

- **A valid plan from graphplan is one in which:**
  - Actions at the same level don't interfere with one another.
  - Each action's preconditions are true at that point **in the plan**.
  - Goals are satisfied at the end of the graph.
- **This is where mutexes kick in:**
  - Two actions or literals are mutex if no valid plan could contain both.



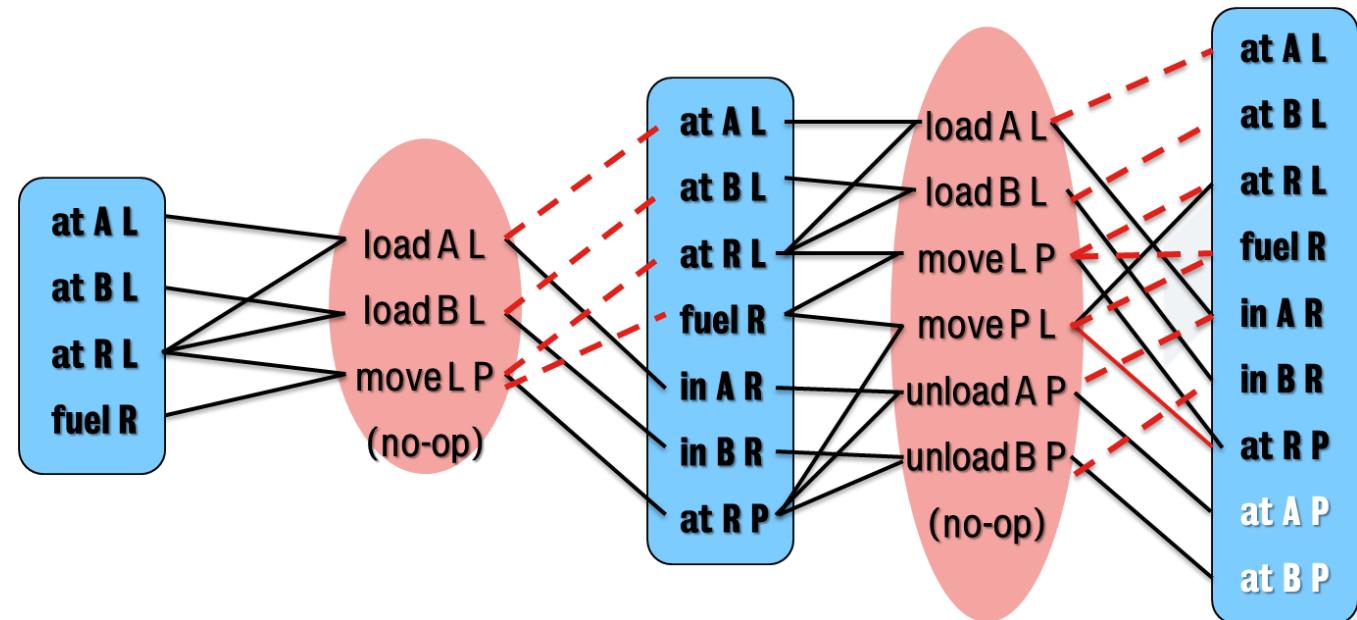
# Mutexes

- Different types of mutex relations can appear in a planning graph.
- Interference
  - Two actions that interfere with one another, where the effect of one negates the precondition of another.
- Competing Needs
  - Two actions where any of their preconditions are mutex with one another.
- Inconsistent Support
  - Two literals that are mutex given all ways of creating both are mutex.

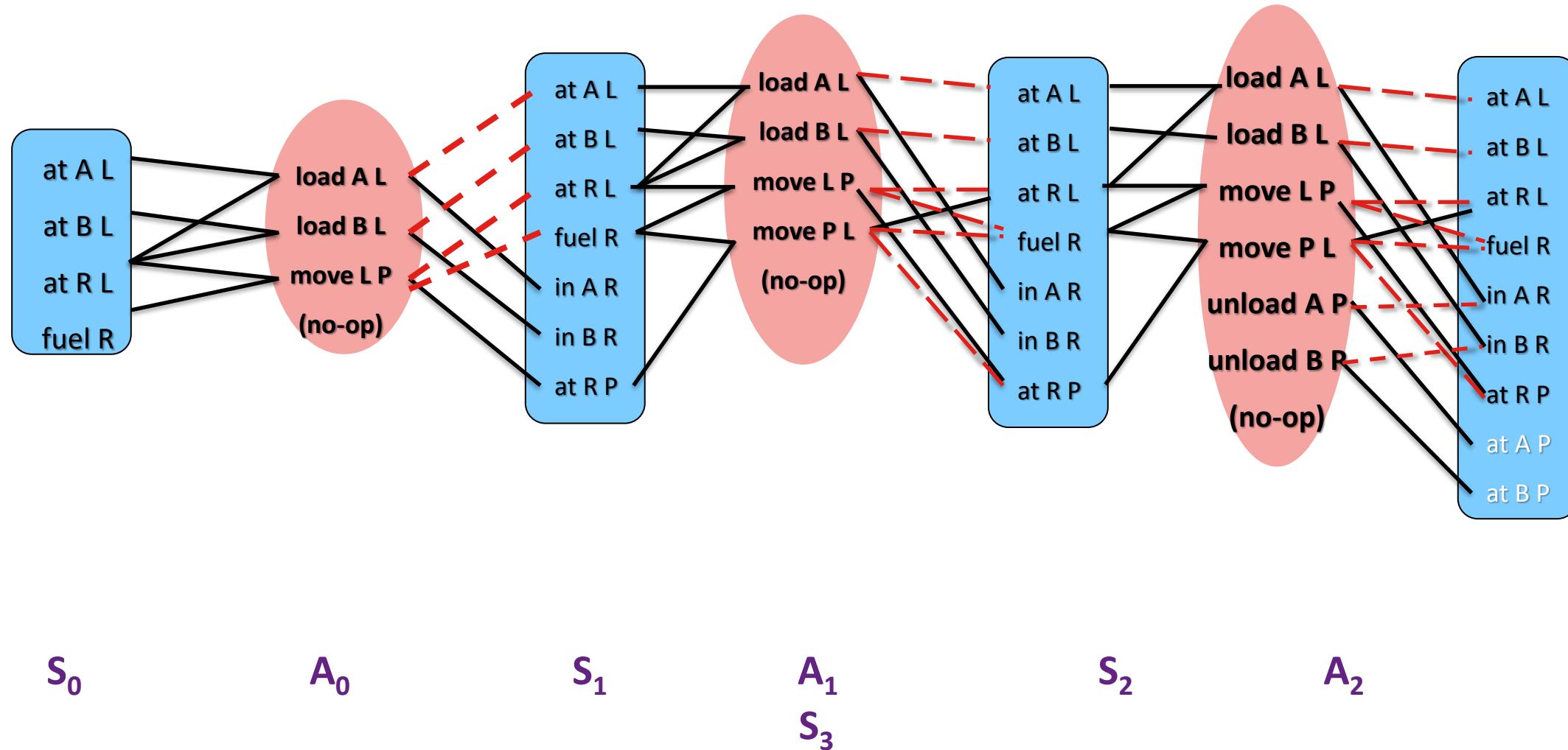


# Using Mutexes

- We extend the planning graph and acknowledge mutexes in fact/action layers.
- Action Layers
  - Include all actions (including no-op) that have all their preconditions satisfied and are not mutex.
  - Mark as mutex all **action pairs** that are incompatible
- Fact Layers
  - Generate all propositions that are the effect of an action in preceding fact layer.
  - Mark as mutex all pairs of propositions that can only be generated by **mutex action pairs**.



# Finding Mutexes

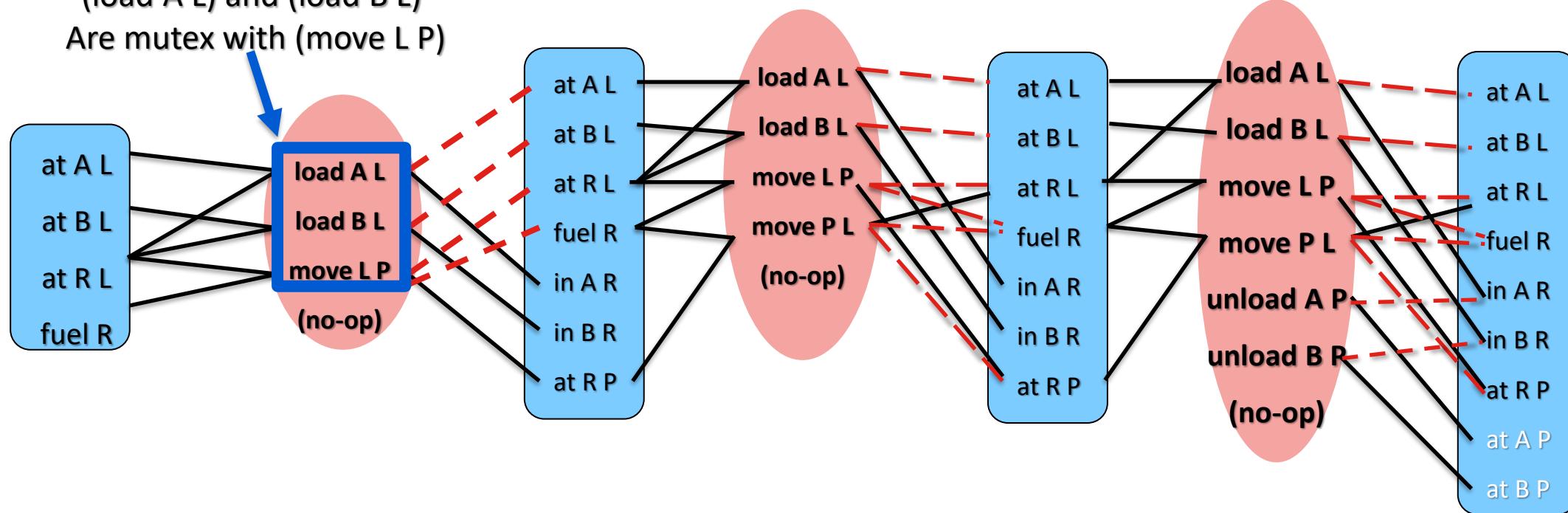


# Finding Mutexes

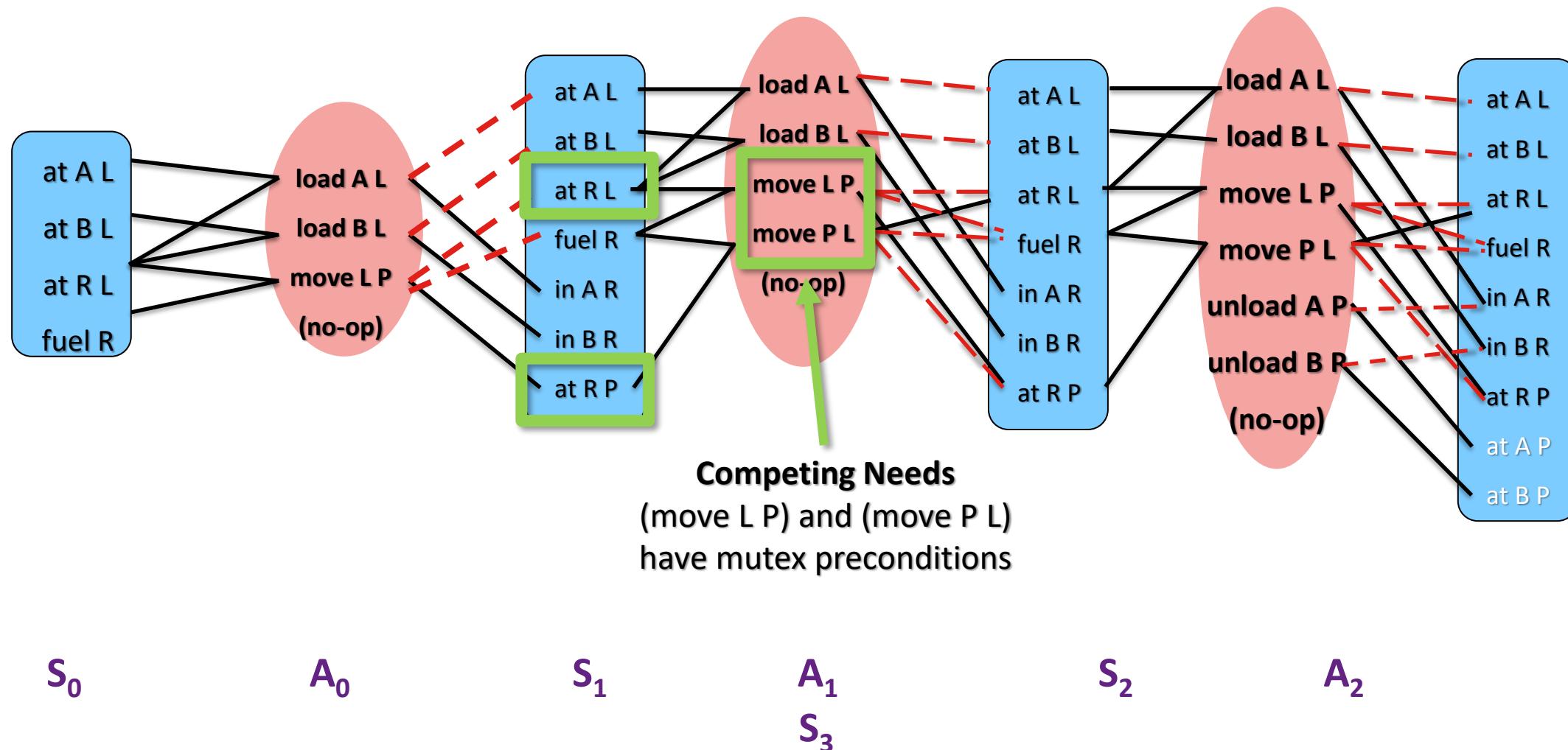
## Interference

(load A L) and (load B L)

Are mutex with (move L P)

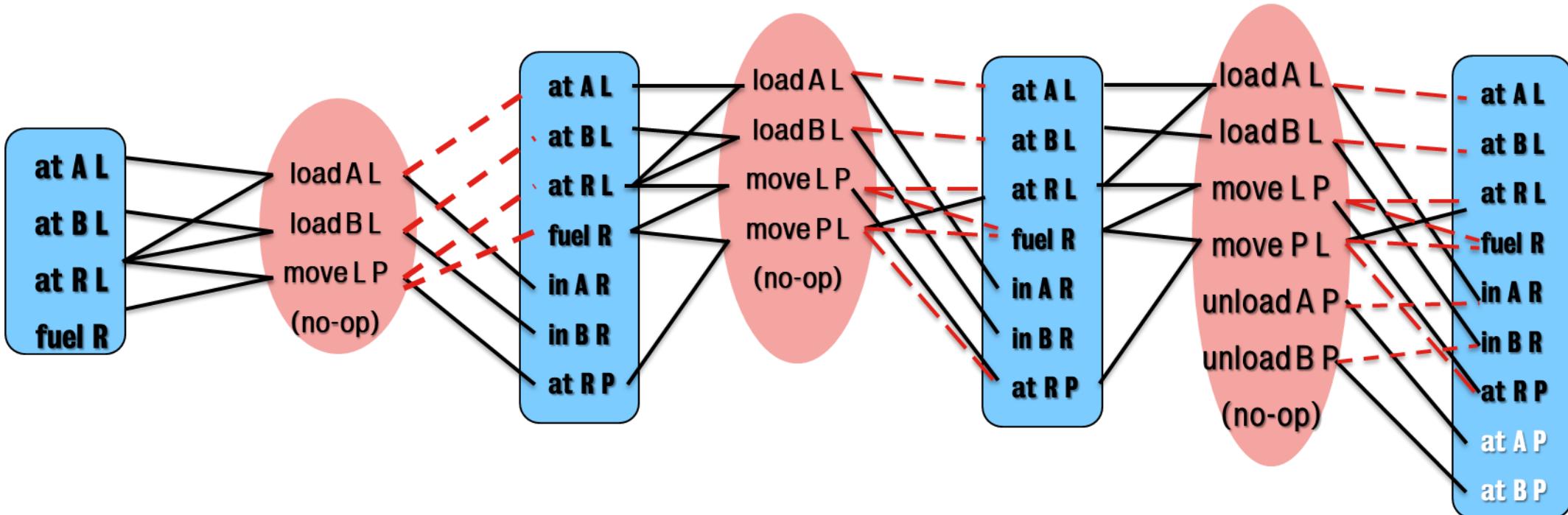


# Finding Mutexes



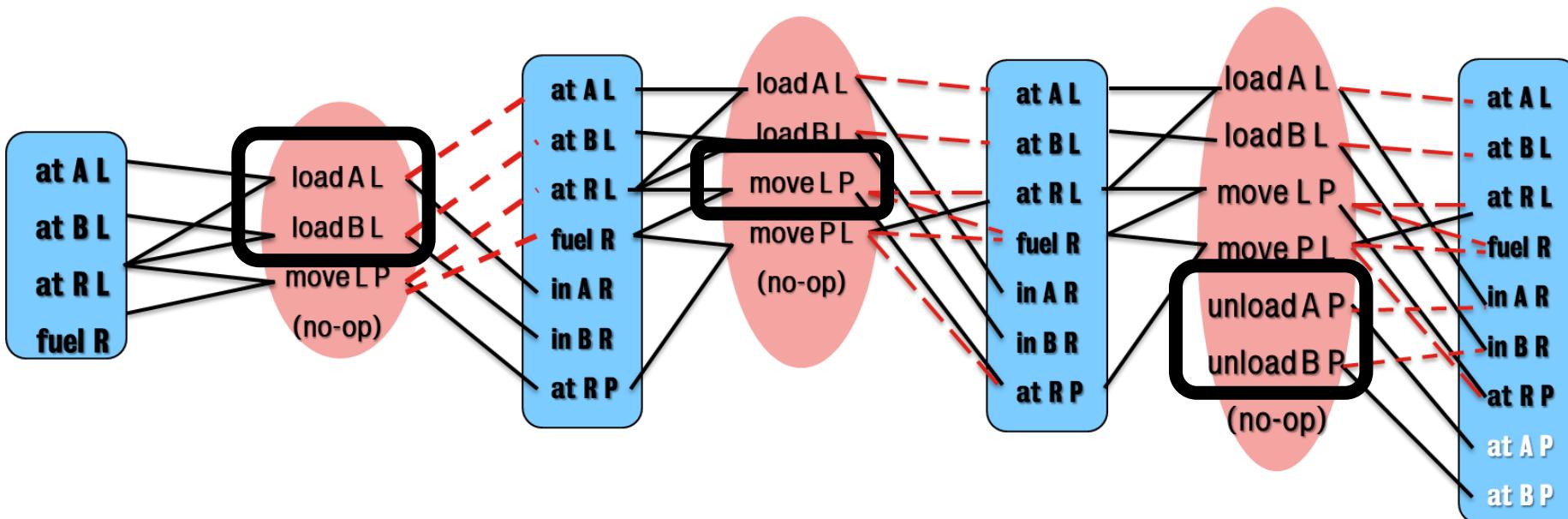
# Basic Graphplan Algorithm

- Grow the planning graph (PG) until all goals are reachable and none are pairwise mutex.
- (If PG levels off [reaches a steady state] first, fail).
- Search backwards from the max fact layer for a valid plan.
- If none found, add a level to the PG and try again.



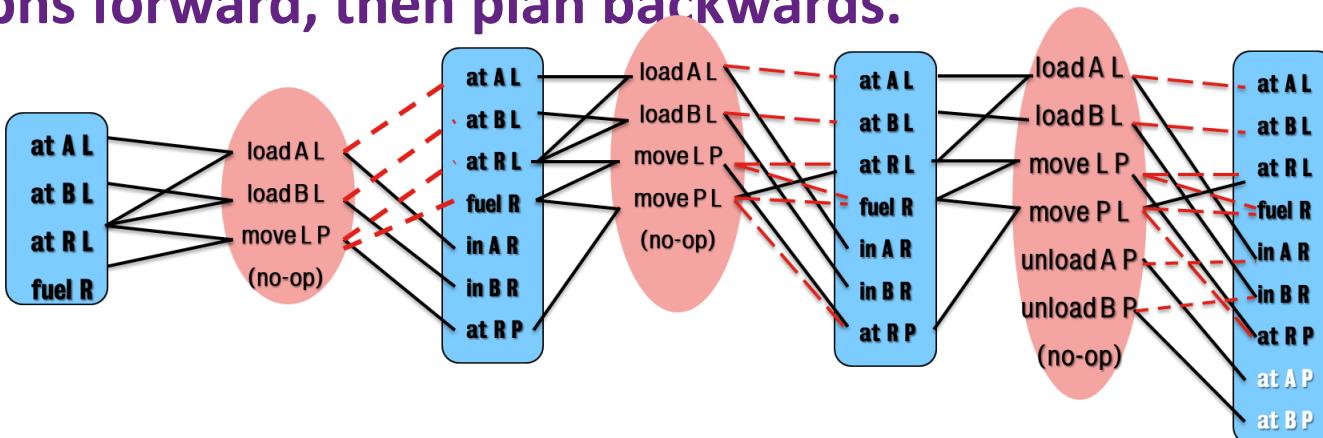
# Plan Generation

- **Backward chain on the planning graph.**
  - Complete all goals at one level before going back
- **At level  $i$ , pick a non-mutex subset of actions that achieve the goals at level  $i+1$ .**
  - The preconditions of these actions become the goals at level  $i$ .
- **Build the action subset by iterating over goals, choosing an action that has the goal as an effect.**
  - Use an action that was already selected if possible. Do forward checking on remaining goals.



# Summary

- Creating planning graphs is a polynomial time process.
- Finding solutions however, is still potentially exponential.
- Mutexes across facts and actions help ensure we find complete and sound plans.
- Generate graphs forward, then plan backwards.



# Classical Planning Non-Forward Search GraphPlan

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



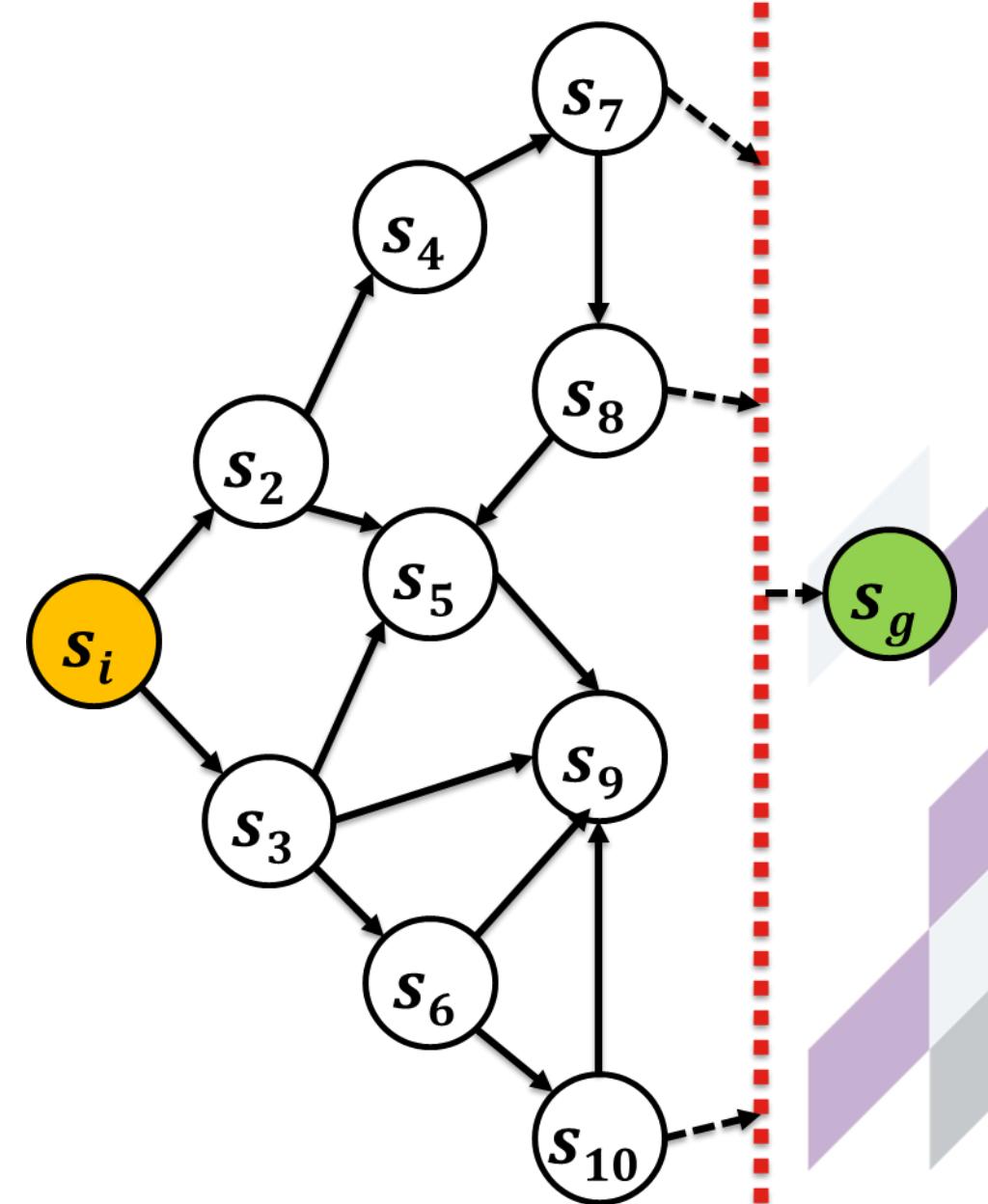
# Classical Planning Non-Forward Search Planning as SAT

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson



# Planning Formalisms

- Throughout this module, we've stuck to some key concepts of how planning is formulated and explored.
  - Set-Theoretic Representation (PDDL)
  - State Transition Model
  - Forward Search
- But there have been alternatives to how we explore a planning problem.
- Idea: What if we transform a planning problem into another problem type for which they are already efficient solvers to address it?



# Boolean Satisfiability Problems (SAT)

- Encoding a problem in such a way that there exists a valid interpretation.
- Create a Boolean expression comprised of variables with operators applied to them.
- If we can find a valid combination of values for this expression, we can consider that SAT problem to be satisfiable.

$$F = A \wedge \bar{B}$$

Satisfiable

$$F = A \wedge \bar{A}$$

Unsatisfiable

# Why Use SAT for Planning?

- SAT solving is a broad and rich research domain in and of itself.
- Typically used in software verification, theorem proving, model checking, pattern generation etc.
- There are existing SAT solvers established within this community that can be adopted.
- So how do we reformulate a planning problem into a SAT problem?

$$F = A \wedge \bar{B}$$

**Satisfiable**

$$F = A \wedge \bar{A}$$

**Unsatisfiable**

# Planning as a SAT Problem

- **Problem:** Our planning formulation  $\rho = (\Sigma, s_i, sg)$
- **Idea: Propositional variables for each state variable.**
- Create unit clauses that specify...
  - The **initial state**.
  - The **goal state**.
- Create clauses to describe how variables can change between two time points  $t$  and  $t + 1$ .
- **Solution:** Create a SAT formula  $\phi$  such that...
  - If  $\phi$  is satisfiable, then a plan exists for the planning problem  $\rho$
  - Every possible solution for  $\phi$  results in a different valid plan for  $\rho$

# SAT Encoding: Variables

- **State Variables**

- A set of propositional variables  $v^i$  that **encode the state** after  $i$  steps of the plan.
- There is a finite number of steps of the plan, encoding as  $T$ : the **planning horizon**

$$v^i \forall v \in V, 0 \leq i \leq T$$

- **Action Variables**

- A set of propositional variables  $a^i$  that **encode the actions applied** at the  $i^{th}$  step of the plan.

$$a^i \forall a \in A, 1 \leq i \leq T$$

# State Variables

- Our planning formulation  $\rho = (\Sigma, s_i, sg)$

- **Initial State**

- Unit clauses encoding the initial state.

$$v^0 \forall v \in s_i \text{ and } \neg v^0 \forall v \notin s_i$$

- **Goal State**

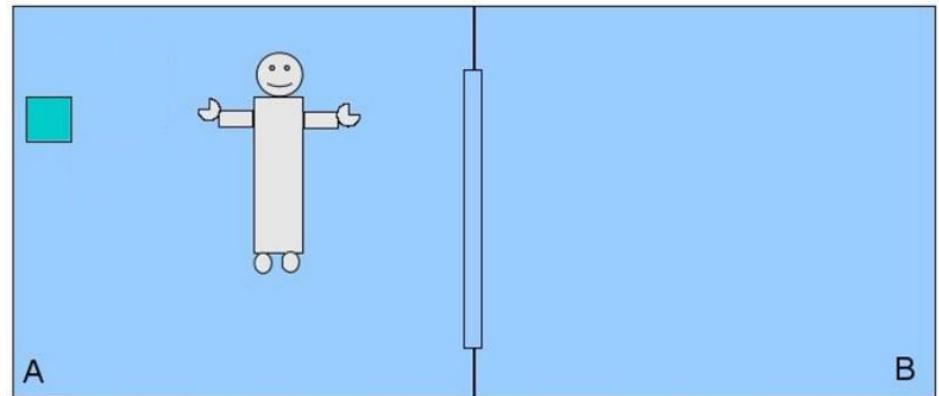
- Unit clauses encoding the goal state.

$$v^T \forall v \in sg$$

# State Variables (Example)

- **Simple Gripper domain problem.**

- Two locations, one box to move from A to B.

$$\{at(r1, locA), at(b1, locA), free(left), free(right)\}$$


- **Encoding for T = 1.**

- Initial State

$$\begin{aligned} & at(r1, locA, 0) \wedge at(b1, locA, 0) \wedge free(left, 0) \wedge free(right, 0) \wedge \neg at(r1, locB, 0) \wedge \neg at(b1, locB, 0) \\ & \wedge \neg holding(left, b1, 0) \wedge \neg holding(right, b1, 0) \end{aligned}$$

- Goal State

$$at(b1, locB, 1)$$

# SAT Encoding: Actions

- For all actions that exist in the domain, we need to encode the preconditions and the add and delete effects of the action.
- But, we need to do it for each time step  $i$  up to the planning horizon  $T$ .
- Sub formulas for encoding preconditions, add/delete effects

for  $0 \leq i \leq (T - 1)$

$$a^i \Rightarrow \{\wedge_{f \in \text{precond}(a)} f^i\} \wedge \{\wedge_{f \in \text{effectAdd}(a)} f^{i+1}\} \wedge \{\wedge_{f \in \text{effectDel}(a)} \neg f^{i+1}\}$$

# Actions (Example)

- **Gripper Actions (valid in initial state)**

- Pickup Box  $b$  using Gripper  $g$  on Robot  $r$  in Location  $x$ .

pre:  $at(b, x), at(r, x), free(g)$

eff:  $\neg at(b, x), \neg free(g), holding(g, b)$

- Move Robot  $r$  from Location  $x$  to Location  $y$ .

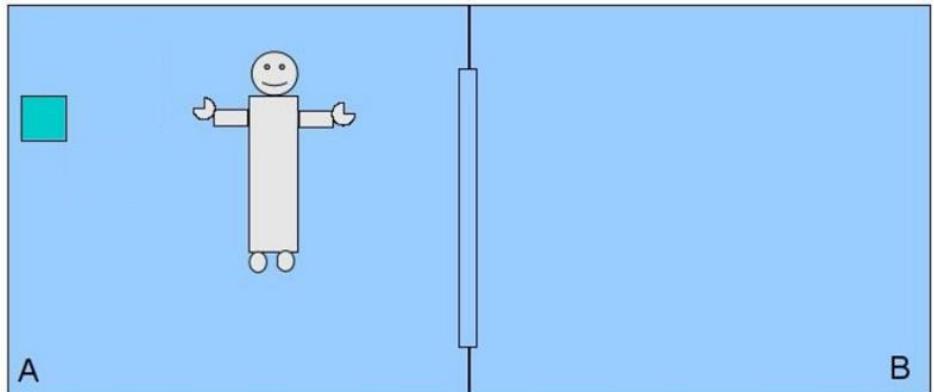
pre:  $at(r, x)$

eff:  $at(r, y), \neg at(r, x)$

- **SAT Encoding (at time step 0)**

$pickup(b1, left, locA, 0) \Rightarrow at(b1, locA, 0) \wedge at(r1, locA) \wedge free(left, 0) \wedge holding(b1, left, 1) \wedge \neg at(b1, locA, 1) \wedge \neg free(left, 1)$

$move(r1, locA, locB, 0) \Rightarrow at(r1, locA, 0) \wedge at(r1, locB, 1) \wedge \neg at(r1, locA, 1)$

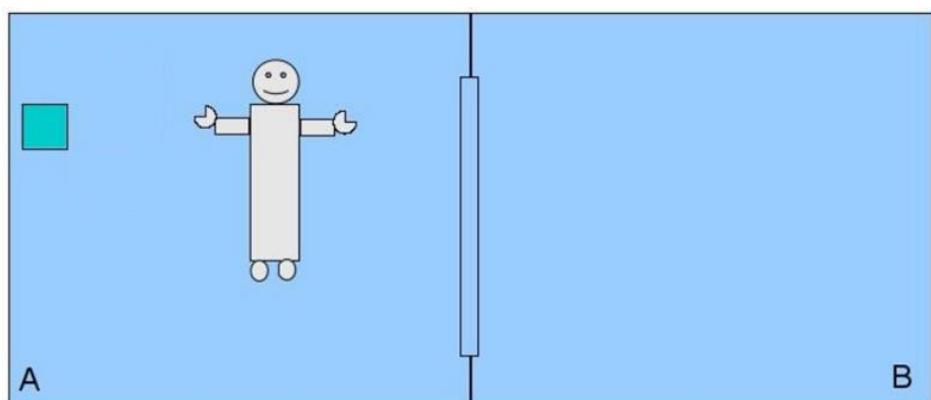


# The Framing Problem

- We need a way to capture information that does not change between time steps  $i$  and  $i + 1$ .
- Two approaches to solving this...
  - Classical Frame Axioms
    - State which facts are not effected for each action.
    - Must enumerate for all fact/action pairs where no the fact is not affected by the action.
    - Relies on one action being executed per time-step so axioms can be applied.
  - Explanatory Frame Axioms
    - Instead of listing what facts are not changed, explain why a fact might have changed between two time steps.
    - i.e. if a fact changes between  $i$  and  $i + 1$ , then an action at step  $i$  must have caused it.

# Explanatory Frame Axioms

- Enumerate the set of actions that could result in a fact changing between time steps.
- Examples

$$\neg at(r1, locB, 0) \wedge at(r1, locB, 1) \Rightarrow move(r1, locA, locB, 0)$$
$$\neg holding(left, b1, 0) \wedge holding(left, b1, 1) \Rightarrow pickup(b1, left, r1, locA, 0)$$


# Exclusion Axioms

- Returning to the notion of **mutual exclusion**, we identify what actions cannot occur at the same time.
- **Complete Exclusion Axiom:** only one action at a time.

$$\neg move(r1, locA, locB, 0) \vee \neg move(r1, locB, locA, 0)$$

- **Conflict Exclusion Axiom:** prevents invalid actions on the same timestep.
  - More on this topic partial order planning in another chapter.

# SAT Planning – Solving the Problem

- Our collection of axioms is converted into conjunctive normal form and then fed to a SAT solver.
- SAT solver will iterate out (increasing T) until it can find an assignment of truth values that satisfies  $\phi$ .
- Once reached – provided planning is total order – there will be exactly one action  $a$  such that  $a^i = \text{true}$ .
  - This is the  $i^{th}$  action of the plan.

# Classical Planning Non-Forward Search Planning as SAT

6CCS3AIP – Artificial Intelligence Planning  
Dr Tommy Thompson

