Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili

[DΣIM]

AI Planning

Hatem A. Rashwan

Graphplan and Advanced Heuristics

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili

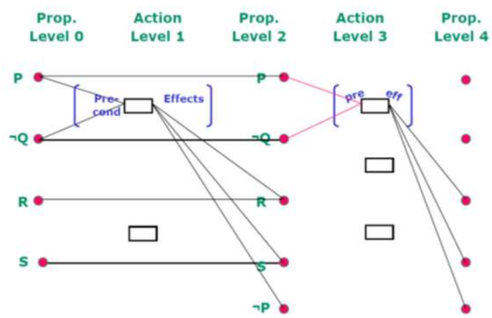
[DΣIM]

Graphplan

Graph Plan

- A propositional planner, that is, there are no variables
  - assertions Simpler – don't have to worry about matching
  - Bigger – if you have six blocks, you need 36 propositions to represent all On(x,y)

1. Make a plan graph of depth k
2. Search for a solution
3. If succeed, return a plan
4. Else k=k+1
5. Go to 1.



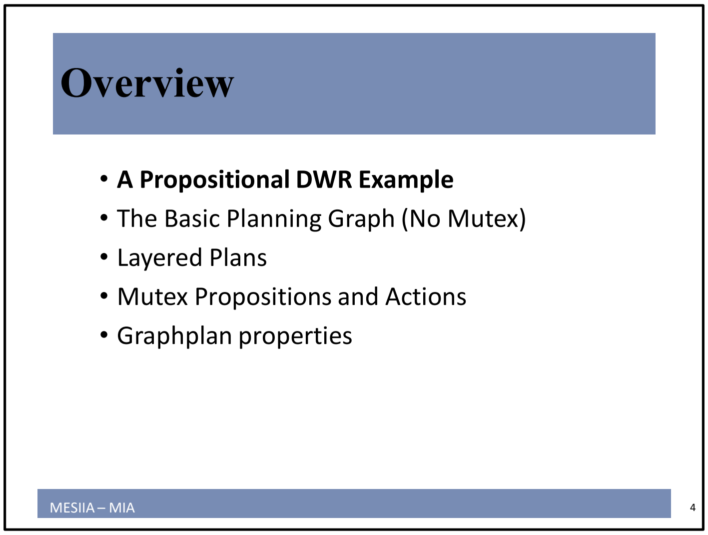
GraphPlan centres work on a data structure called a plan graph. A plan graph looks like this in the figure. You have a bunch of levels. You start with level zero, level one, level two.

The first big thing about Graphplan is that it's a propositional planner. So that means there are no variables around in the course of planning. When doing **the blocks world,** we had operator descriptions with variables that let us generally speak about moving blocks rather than naming particular ones. In GraphPlan, we're not going to be able to have any variables floating around during the planning process.

Not having any variables makes your life simpler because you don't have to worry about unification, variable matching, or anything else. But it may be more challenging if you have six blocks and action of on (A, B) relation. You'll have to make 36 propositions for every possible instantiation of the variables in that relationship. So if you need to talk about six different blocks and how they could be on each other, then that might be a lot of propositions. But at least in this work, it's going to turn out that it's worth having an extensive representation that's reasonably easy to deal with; that it's going to be more efficient to do that than to have a very concise but kind of complicated representation as we have when we have variables. So that's the tradeoff, and in this case, we're going to go for the propositional planner, big but simple.

The Graphplan algorithm has the following structure. This isn't going to make too much sense until we look at the pieces in detail, but the idea is that you make a plan graph of depth k and then search for a solution, and if you succeed, you return a plan. Otherwise, you increment k and try again. So that's the basic scheme.

Note that I wrote, "make a plan graph of depth k." We're going to look for plans of depth k, so if we look for a depth two plan, it will have two-time steps, but it will be partially ordered in the sense that multiple actions might take place in a single time step. Maybe you can or can not execute them in parallel, but there will be some actions where you don't care in what order they occur.



Overview

➢**A Propositional DWR Example**

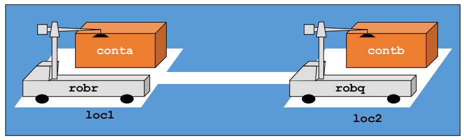
•The Basic Planning Graph (No Mutex)

•Layered Plans

•Mutex Propositions and Actions

•The Graphplan properties

Example: Simplified DWR Problem



- robots can load and unload autonomously
- locations may contain unlimited number of robots and containers
- problem: swap locations of containers

Example: Simplified DWR Problem

•**[figure]**

•initial state:

  •2 locations: loc1 and loc2, connected by path

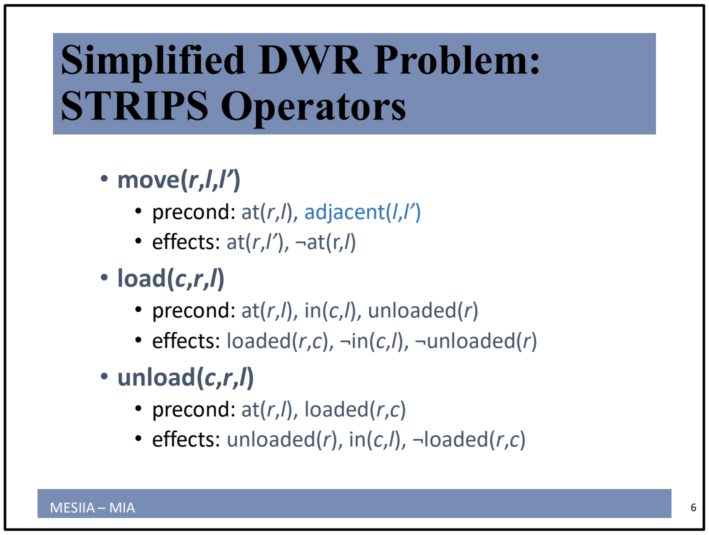
  •2 robots: robr and robq, both unloaded initially at loc1 and loc2 respectively

  •2 containers: conta and contb, initially at loc1 and loc2 respectively

•**robots can load and unload autonomously**

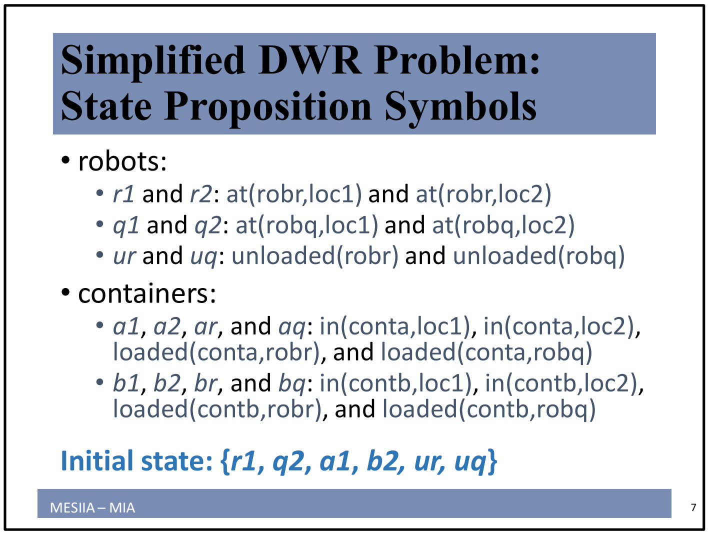
•**locations may contain unlimited number of robots and containers**

•**problem: swap locations of containers**



Simplified DWR Problem: STRIPS Actions

- **move(*r*,*l*,*l'*)**
  - move robot *r* from location *l* to adjacent location *l'* (4 possible actions; with rigid adjacent relation evaluated)
  - **precond: at(*r*,*l*), adjacent(*l*,*l'*)**
  - **effects: at(*r*,*l'*), ¬at(r,*l*)**
- **load(*c*,*r*,*l*)**
  - load container *c* onto robot *r* at location *l* (8 possible actions)
  - **precond: at(*r*,*l*), in(*c*,*l*), unloaded(*r*)**
  - **effects: loaded(*r*,*c*), ¬in(*c*,*l*), ¬unloaded(*r*)**
- **unload(*c*,*r*,*l*)**
  - unload container *c* from robot *r* at location *l* (8 possible actions)
  - **precond: at(*r*,*l*), loaded(*r*,*c*)**
  - **effects: unloaded(*r*), in(*c*,*l*), ¬loaded(*r*,*c*)**



Simplified DWR Problem: State Proposition Symbols

•idea: represent each atom that may occur in a state by a single (short) proposition symbol

•robots:

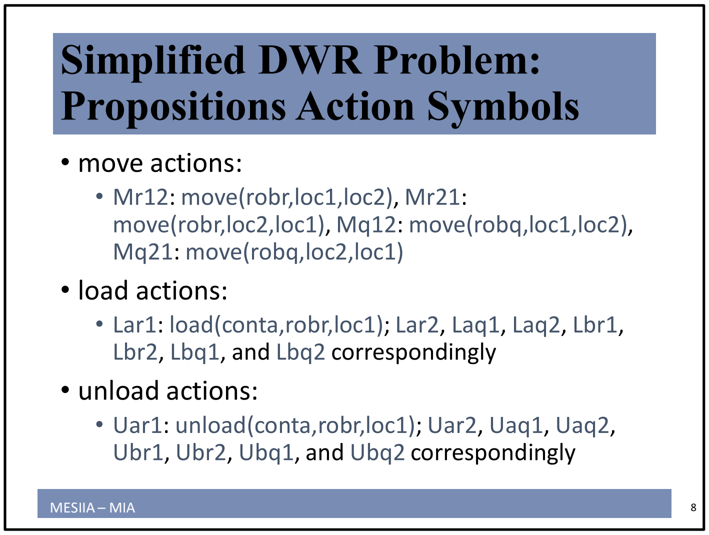
- **•*r1* and *r2*: at(robr,loc1) and at(robr,loc2)**
- **•*q1* and *q2*: at(robq,loc1) and at(robq,loc2)**
- **•*ur* and *uq*: unloaded(robr) and unloaded(robq)**

•containers:

- **•*a1*, *a2*, *ar*, and *aq*: in(conta,loc1), in(conta,loc2), loaded(conta,robr), and loaded(conta,robq)**
- **•*b1*, *b2*, *br*, and *bq*: in(contb,loc1), in(contb,loc2), loaded(contb,robr), and loaded(contb,robq)**

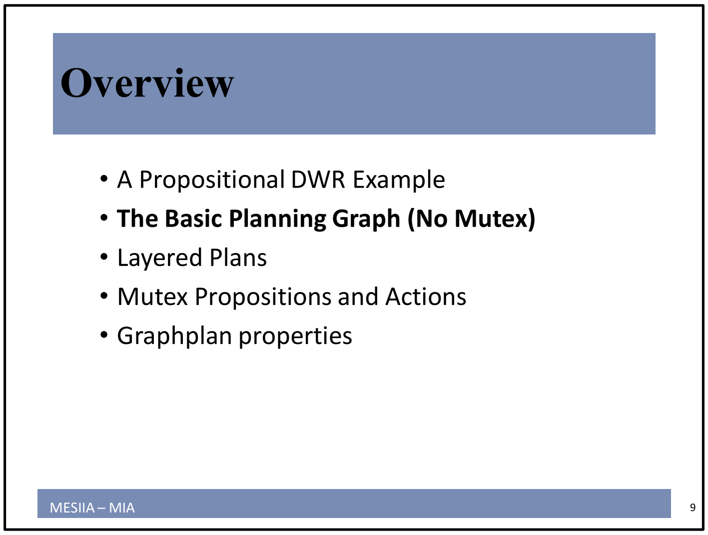
•14 state propositions

•initial state: {*r1*, *q2*, *a1*, *b2*, *ur*, *uq*}



Simplified DWR Problem: Action Symbols

- **move actions:**
  - **Mr12: move(robr,loc1,loc2), Mr21: move(robr,loc2,loc1), Mq12: move(robq,loc1,loc2), Mq21: move(robq,loc2,loc1)**
- **load actions:**
  - **Lar1: load(conta,robr,loc1); Lar2, Laq1, Laq2, Lar1, Lbr2, Lbq1, and Lbq2 correspondingly**
- **unload actions:**
  - **Uar1: unload(conta,robr,loc1); Uar2, Uaq1, Uaq2, Uar1, Ubr2, Ubq1, and Ubq2 correspondingly**
- 14 state symbols: lower case, italic
- 20 action symbols: uppercase, not italic



Overview

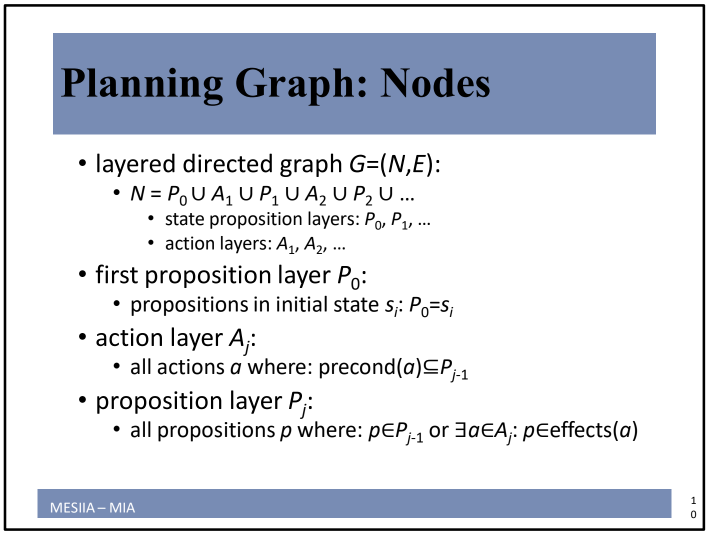
. A Propositional DWR Example

•**The Basic Planning Graph (No Mutex)**

•Layered Plans

•Mutex Propositions and Actions

•The Graphplan properties



Planning Graph: Nodes

- **layered directed graph $G=(N,E)$:**
  - layered = each node belongs to exactly one layer
  - **$N = P_0 \cup A_1 \cup P_1 \cup A_2 \cup P_2 \cup \ldots$**
    - proposition and action layers alternate
    - **state proposition layers: $P_0, P_1, \ldots$**
    - **action layers: $A_1, A_2, \ldots$**
- **first proposition layer P_0:**
  - **propositions in initial state s_i: $P_0=s_i$**
- **action layer A_j:**
  - **all actions a where: precond(a)$\subseteq P_{j-1}$**
- **proposition layer P_j:**
  - **all propositions p where: $p \in P_{j-1}$ or $\exists a \in A_j$: $p \in$ effects$^+$(a)**
  - propositions at layer P_j are all propositions in the union of all nodes in the reachability tree at depth j
    - note: negative effects are not deleted from next layer
- note: $P_{j-1} \subseteq P_j$; propositions in the graph monotonically (incrementally) increase from one proposition layer to the next

Planning Graph: Edges

- from proposition $p \in P_{j-1}$ to action $a \in A_j$:
  - if: $p \in$ precond(a)
- from action $a \in A_j$ to layer $p \in P_j$:
  - positive arc if: $p \in$ effects$^+$(a)
  - negative arc if: $p \in$ effects$^-$(a)

Planning Graph: Arcs

•directed and layered = arcs only from one layer to the next

•from proposition $p \in P_{j-1}$ to action $a \in A_j$:

•if: $p \in$ precond(a)

•from action $a \in A_j$ to layer $p \in P_j$:

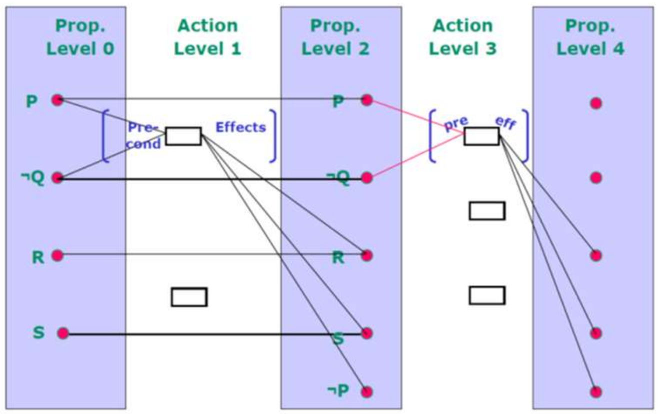
•positive arc if: $p \in$ effects$^+$(a)

•negative arc if: $p \in$ effects$^-$(a)

•no arcs between other layers

•note: $A_{j-1} \subseteq A_j$; actions in the graph monotonically increase from one action layer to the next

Graph Plan Example



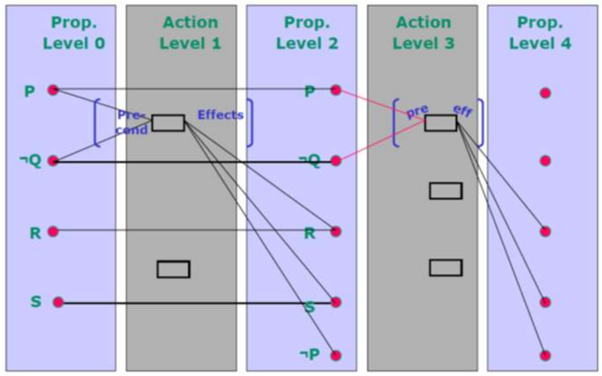
At the even- numbered levels you have propositions, which they draw as a little dot.

Not surprisingly, given its name, GraphPlan centers its work on a data structure called a plan graph.
A plan graph looks like this; see the figure.

You have a bunch of levels. You start with level zero, level one, level two etc.

At the even-numbered levels, you have propositions, which they draw as a little dot.

Graph Plan Example

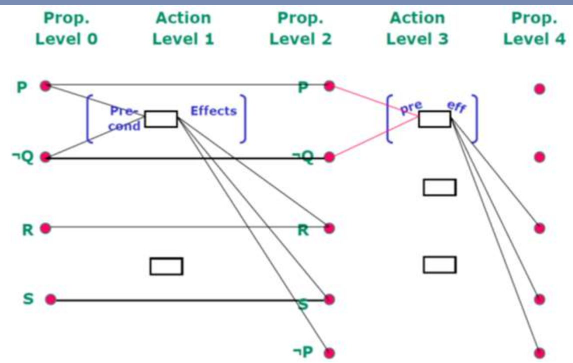


- Three proposition levels (levels 0, 2, and 4) and two action levels (levels 1 and 3).
- To encode depth-two plans (because there are two layers of actions).
  - Action level 1 has the actions that we might choose to do on the first step,
  - Action level 3 has the actions we might choose to do on the second step.

At the odd-numbered levels, you have actions, shown as boxes.

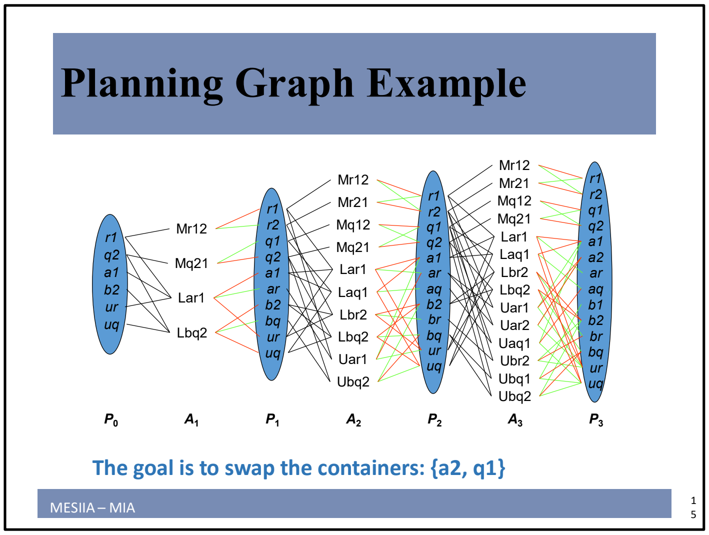
In this figure, we have three proposition levels (levels 0, 2, and 4) and two action levels (levels 1 and 3). In this graph, we are able to encode depth-two plans (because there are two layers of actions). Action level 1 has the actions that we might choose to do on the first step, and action level 3 has the actions we might choose to do on the second step.

Graph Plan Example



- Start by making a graph with levels 0 through 2, corresponding to a depth 1 plan,
- Search for a satisfactory plan within that graph. If we can't find one,
- Extend the graph out by two more layers (an action layer and a proposition layer),
- Then find a depth 2 plan.

And then it's within this structure that we're going to look for a plan. So we start by making a graph with levels 0 through 2, corresponding to a depth 1 plan, and search for a satisfactory plan within that graph. If we can't find one, we extend the graph out by two more layers (an action layer and a proposition layer), and then try to find a depth 2 plan.

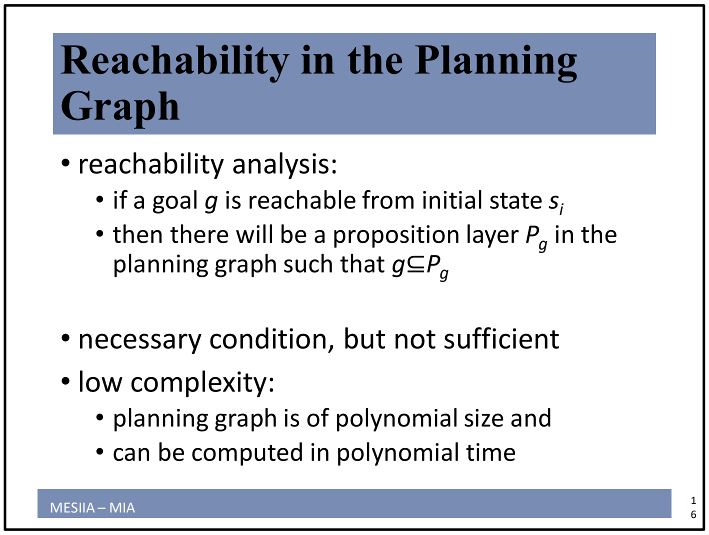


Planning Graph Example

•**[figure]**

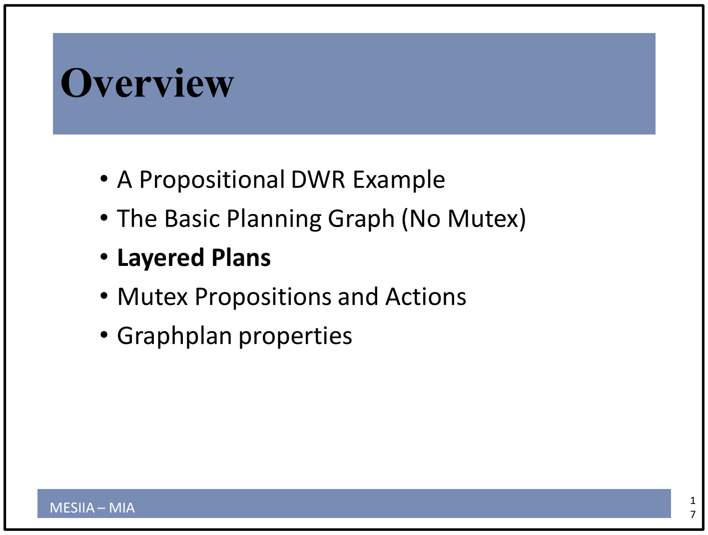
- •start with initial proposition layer
- •next action layer: applicable action; links from preconditions (black)
- •next proposition layer: previous proposition plus positive effects; links to positive effects (green); links to negative effects (red)
- •next action layer (A_2); precondition links; next proposition layer (P_2); effect links
- •next action layer (A_3); precondition links; next proposition layer (P_3); effect links

•action layers contain “inclusive disjunctions” of actions



Reachability in the Planning Graph

- **reachability analysis:**
  - **if a goal g is reachable from initial state s_i**
  - **then there will be a proposition layer P_g in the planning graph such that $g \subseteq P_g$**
  - or: if no proposition layer contains g then g is not reachable
- **necessary condition, but not sufficient**
  - necessary vs. sufficient:
    - planning graph:
      - proposition layers contains propositions that may possibly hold
      - propositions in one layer usually inconsistent (e.g. robots/containers in two places at once)
      - similarly, incompatible actions in one layer may interfere with each other
- **low complexity:**
  - **planning graph is of polynomial size and**
  - **can be computed in polynomial time**
- need more conditions (for sufficient criterion)



Overview

. A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

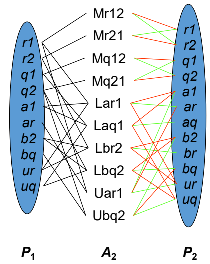
•Layered Plans

•Mutex Propositions and Actions

•The Graphplan properties

Independent Actions: Examples

- Mr12 and Lar1:
  - cannot occur together
  - Mr12 deletes precondition *r1* of Lar1
- Mr12 and Mr21:
  - cannot occur together
  - Mr12 deletes positive effect *r1* of Mr21
- Mr12 and Mq21:
  - may occur in same action layer



Independent Actions: Examples
•Mr12 and Lar1:
•cannot occur together
•Mr12 deletes precondition *r1* of Lar1
•Mr12 and Mr21:
•cannot occur together
•Mr12 deletes positive effect *r1* of Mr21
•Mr12 and Mq21:
•may occur in same action layer

Independent Actions

- Two actions a_1 and a_2 are <u>independent</u> iff:
  - effects$^-(a_1) \cap (\text{precond}(a_2) \cup \text{effects}^+(a_2)) = \{\}$ and
  - effects$^-(a_2) \cap (\text{precond}(a_1) \cup \text{effects}^+(a_1)) = \{\}$.
- A set of actions π is independent iff every pair of actions $a_1,a_2 \in \pi$ is independent.
- The final solution Π is {π1, π2, π3,…., πk}

Independent Actions

•idea: independent actions can be executed in any order (in same layer)

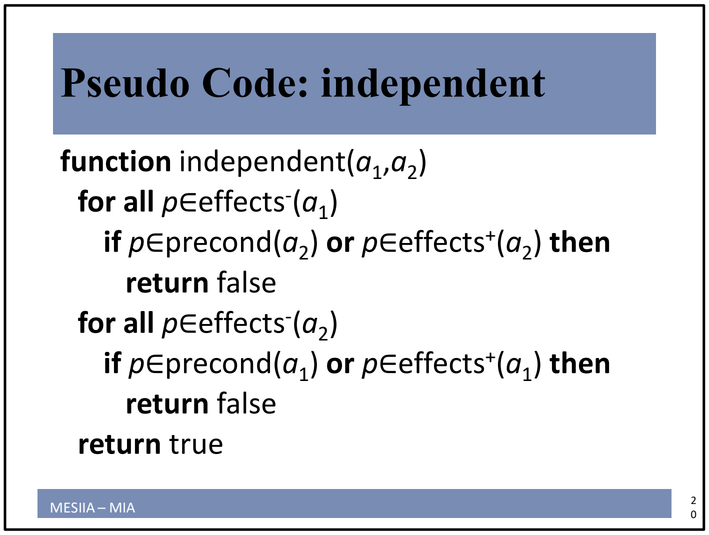
•**Two actions a_1 and a_2 are <u>independent</u> iff:**

- •**effects$^-(a_1) \cap (\text{precond}(a_2) \cup \text{effects}^+(a_2)) = \{\}$ and**
- •**effects$^-(a_2) \cap (\text{precond}(a_1) \cup \text{effects}^+(a_1)) = \{\}$.**
- •two actions are dependent iff:
  - •one deletes a precondition of the other or
  - •one deletes a positive effect of the other

•**A set of actions π is independent iff every pair of actions $a_1,a_2 \in \pi$ is independent.**

•note: independence does not depend on planning problem; can be pre-computed

•note: independence relation is symmetrical (follows from definition)



Pseudo Code: independent

- **function independent(a_1,a_2)**
    - returns true iff the two given actions are independent
- **for all p∈effects$^-$(a_1)**
- **if p∈precond(a_2) or p∈effects$^+$(a_2) then**
- **return false**
- **for all p∈effects$^-$(a_2)**
- **if p∈precond(a_1) or p∈effects$^+$(a_1) then**
- **return false**
- **return true**
- complexity:
    - let *b* be max. number of preconditions, positive, and negative effects of any action
    - element test in hash-set takes constant time
    - complexity: $O(b)$

Layered Plans

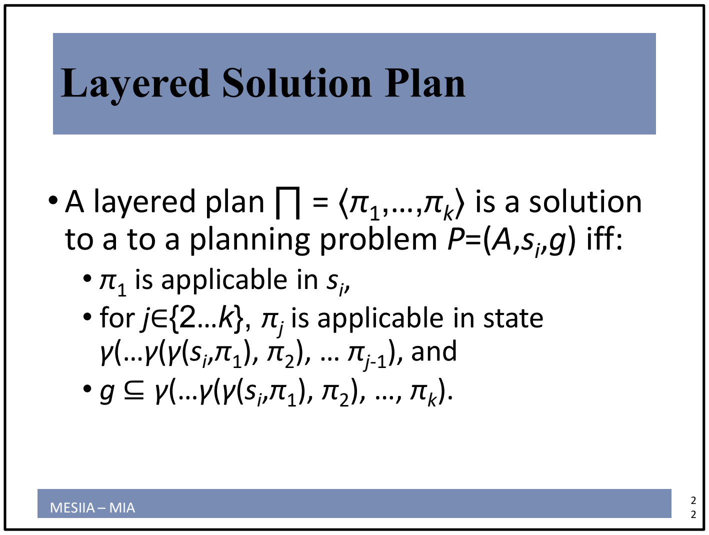
- Let $P = (A,s_i,g)$ be a statement of a propositional planning problem and $G = (N,E)$, $N = P_0 \cup A_1 \cup P_1 \cup A_2 \cup P_2 \cup \ldots$, the corresponding planning graph.
- A <u>layered plan</u> over G is a sequence of sets of actions: $\prod = \langle \pi_1,\ldots,\pi_k \rangle$ where, k is the graph depth, and:
  - $\pi_i \subseteq A_i \subseteq A$,
  - π_i is applicable in state P_{i-1}, and
  - the actions in π_i are independent.

Layered Plans

•Let $P = (A,s_i,g)$ be a statement of a propositional planning problem and $G = (N,E)$, $N = P_0 \cup A_1 \cup P_1 \cup A_2 \cup P_2 \cup \ldots$, the corresponding planning graph.

•A <u>layered plan</u> over G is a sequence of sets of actions: $\prod = \langle \pi_1,\ldots,\pi_k \rangle$ where:

- **$\pi_i \subseteq A_i \subseteq A$,**
- **π_i is applicable in state P_{i-1}, and**
- **the actions in π_i are independent.**

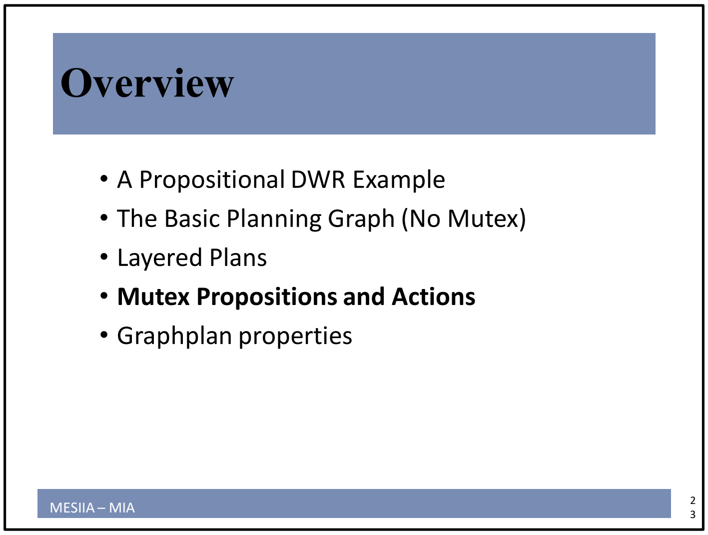


Layered Solution Plan

•**A layered plan ∏ = ⟨π_1,…,π_k⟩ is a solution to a to a planning problem P=(A,s_i,g) iff:**

- •**π_1 is applicable in s_i,**
- •**for $j \in \{2 \ldots k\}$, π_j is applicable in state $\gamma(\ldots\gamma(\gamma(s_i,\pi_1), \pi_2), \ldots \pi_{j-1})$, and**
- •**$g \subseteq \gamma(\ldots\gamma(\gamma(s_i,\pi_1), \pi_2), \ldots, \pi_k)$.**

•note: independence of actions still not sufficient criterion for solution



Overview

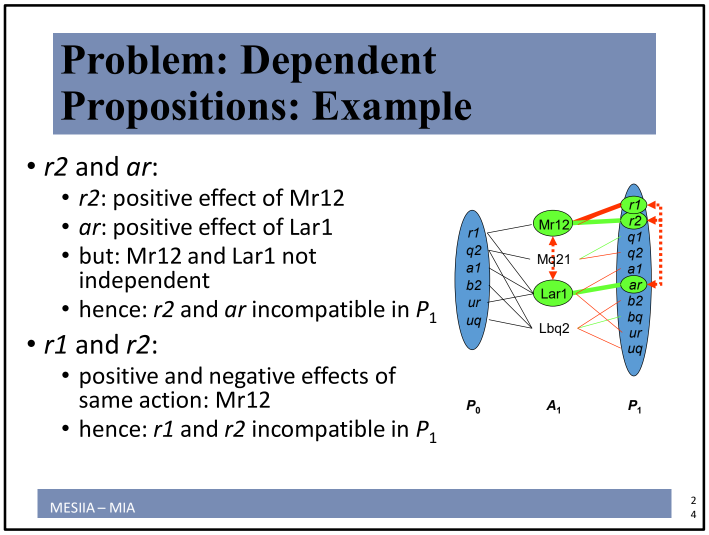
. A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

•Layered Plans

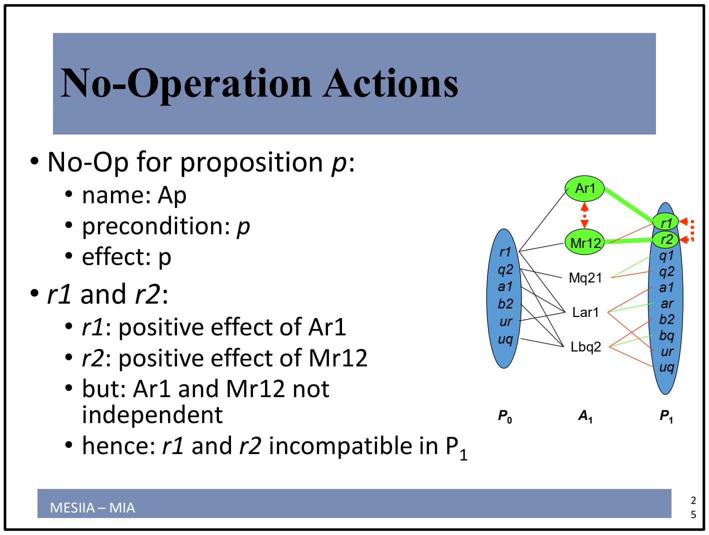
•Mutex Propositions and Actions

•The Graphplan properties



Problem: Dependent Propositions: Example

- ***r2* and *ar*:**
  - ***r2*: positive effect of Mr12**
  - ***ar*: positive effect of Lar1**
  - **but: Mr12 and Lar1 not independent**
    - dependent actions cannot occur together same set of actions in a layered plan, e.g. in π_1
  - **hence: *r2* and *ar* incompatible in P_1**
- ***r1* and *r2*:**
  - **positive and negative effects of same action: Mr12**
  - **hence: *r1* and *r2* incompatible in P_1**
- both cases: compatible if they are also
  - two positive effects of one action
  - the positive effects of two independent actions
- incompatible propositions: cannot be reached through preceding action layer (A_1)



No-Operation Actions

•**No-Op for proposition *p*:**

- •for every action layer and every proposition that may persist
- •**name: Ap**
- •**precondition: *p***
- •**effect: p**

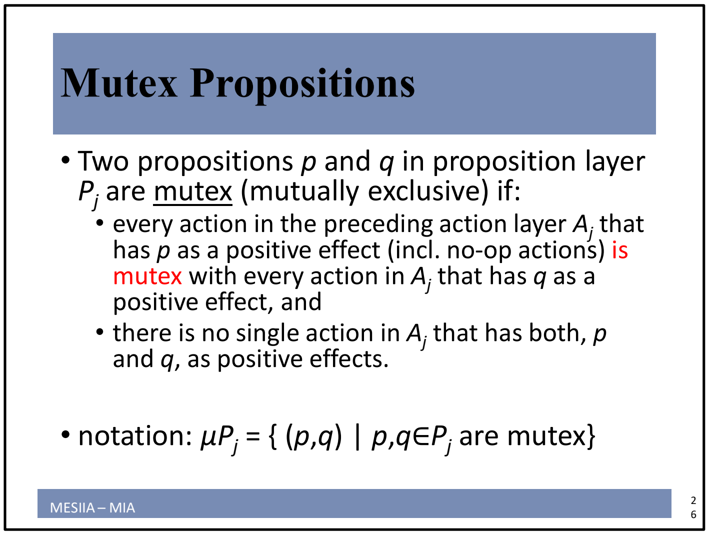
•***r1* and *r2*:**

- •***r1*: positive effect of Ar1**
- •***r2*: positive effect of Mr12**
- •**but: Ar1 and Mr12 not independent**
- •**hence: *r1* and *r2* incompatible in P_1**

•**only one incompatibility test**

•previous slide: two types of incompatibility (positive effects of dependent actions + positive and negative effects of same action)

- •with no-ops: only first type needed (simplification)



Mutex Propositions

- **Two propositions p and q in proposition layer P_j are <u>mutex</u> (mutually exclusive) if:**
  - **every action in the preceding action layer A_j that has p as a positive effect (incl. no-op actions) is mutex with every action in A_j that has q as a positive effect, and**
  - need to define when two actions are mutex
    - obvious case: if they are dependent
  - **there is no single action in A_j that has both, p and q, as positive effects.**
- **notation: $\mu P_j = \{ (p,q) \mid p,q \in P_j \text{ are mutex}\}$**
- note: mutex relation for propositions is symmetrical (follows from definition)
- proposition layer P_1 contains 8 mutex pairs

Pseudo Code: mutex for Propositions

function mutex(p_1,p_2,μA_j)
  for all $a_1 \in p_1$.producers()
    for all $a_2 \in p_2$.producers()
      if $(a_1,a_2) \notin \mu A_j$ **then**
        return false
  return true

Pseudo Code: mutex for Propositions

- **function mutex(p_1,p_2, μA_j)**
  - input: two propositions (from same layer), mutex relation between the actions in the preceding layer
- **for all $a_1 \in p_1$.producers()**
  - producers: actions in the preceding layer that have p_1 as a positive effect; should be stored with proposition node
- **for all $a_2 \in p_2$.producers()**
  - producers: see above
- **if $(a_1,a_2) \notin \mu A_j$ then**
  - test whether the action are in the given set of mutually exclusive actions
- **return false**
  - if not: consistent producers found; propositions are not mutex
- **return true**
  - no consistent producers found; propositions are mutex

- note: single action producing both is covered: action cannot be mutex with itself
- complexity: let m be number of actions in domain (incl. no-ops); $O(m^2)$

Mutex Actions

- Two actions a_1 and a_2 in action layer A_j are <u>mutex</u> if:
  - a_1 and a_2 are dependent, or
  - a precondition of a_1 is mutex with a precondition of a_2.
- notation: μA_j = { (a_1,a_2) | $a_1,a_2 \in A_j$ are mutex}

Mutex Actions

•**Two actions a_1 and a_2 in action layer A_j are <u>mutex</u> if:**

- •**a_1 and a_2 are dependent, or**
  - •dependent actions are necessarily mutex
- •**a precondition of a_1 is mutex with a precondition of a_2.**
- •dependency is domain-specific, i.e. not problem-specific
- •mutex-relation is problem specific
  - •pair of actions/propositions may be mutex in one layer but not so in another

•**notation:**

μA_j = { (a_1,a_2) | $a_1,a_2 \in A_j$ are mutex}

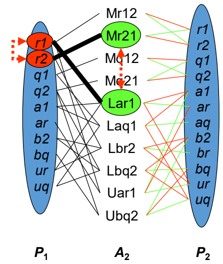
•action layer A_1 contains 2 mutex (dependent) pairs

•action layer A_2 contains 24 mutex pairs (not all dependent)

•note: mutex relation (for actions and propositions) is symmetrical (follows from definition)

Mutex Actions: Example

- *r1* and *r2* are mutex in P_1
- *r1* is precondition for Lar1 in A_2
- *r2* is precondition for Mr21 in A_2
- hence: Lar1 and Mr21 are mutex in A_2



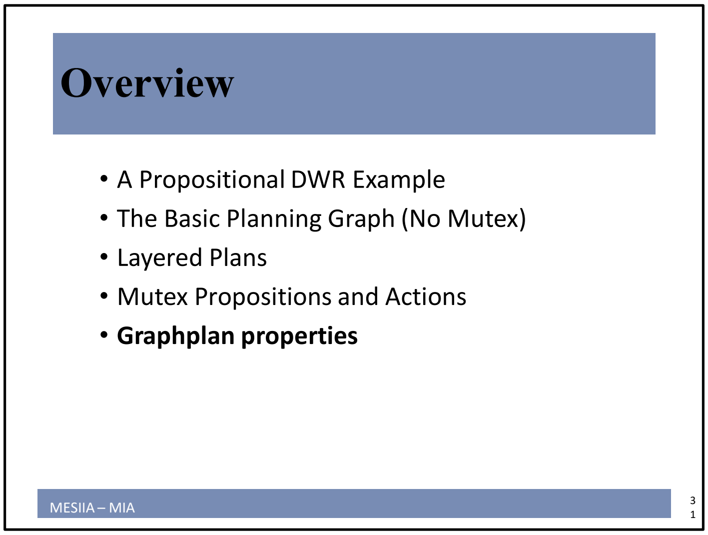
Mutex Actions: Example
•*r1* and *r2* are mutex in P_1
•*r1* is precondition for Lar1 in A_2
•*r2* is precondition for Mr21 in A_2
•hence: Lar1 and Mr21 are mutex in A_2
•dependency between actions in action layer A_j leads to mutex between propositions in P_j
•mutex between propositions in P_j leads to mutex between actions in action layer A_{j+1}

Pseudo Code: mutex for Actions

```
function mutex(a1,a2,μP)
  if ¬independent(a1,a2) then
    return true
  for all p1∈precond(a1)
    for all p2∈precond(a2)
      if (p1,p2)∈μP then return true
  return false
```


Pseudo Code: mutex for Actions

- **function mutex(a_1,a_2,μP)**
    - μP – mutex relations from the preceding proposition layer
- **if ¬independant(a_1,a_2) then**
- **return true**
- **for all $p_1 \in$precond(a_1)**
- **for all $p_2 \in$precond(a_2)**
- **if (p_1,p_2)$\in \mu P$ then return true**
- **return false**
- complexity: let b = max number preconditions/pos. effects/neg effects: $O(b^2)$



Overview

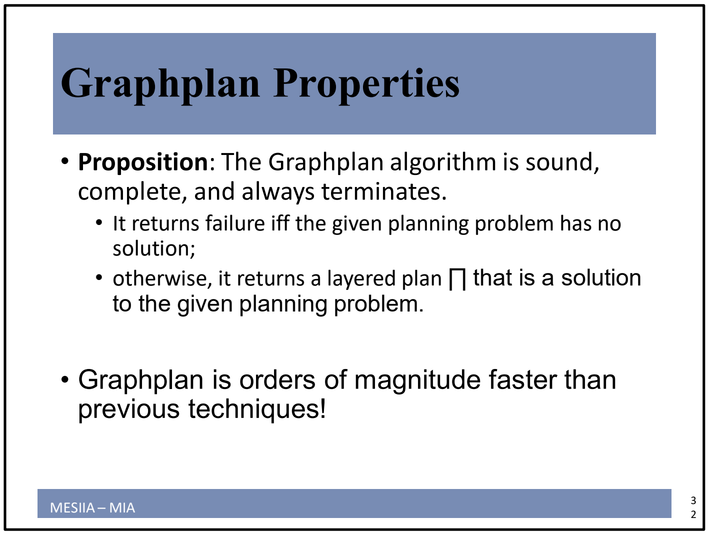
. A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

•Layered Plans

•Mutex Propositions and Actions

•The Graphplan properties



Graphplan Properties

- **Proposition: The Graphplan algorithm is sound, complete, and always terminates.**
    - **It returns failure iff the given planning problem has no solution;**
    - **otherwise, it returns a layered plan ∏ that is a solution to the given planning problem.**
- **Graphplan is orders of magnitude faster than previous techniques!**
    - caveat: restriction to propositional STRIPS

Advanced Heuristics

Overview

- **Simple Planning Graph Heuristics**
- The FF Planner

MESIIA – MIA 34

Overview

➢**Simple Planning Graph Heuristics**

•**Pattern Database Heuristics**

•**The FF Planner**

Forward State-Space Search with A*

- A* is optimally efficient: For a given heuristic function, no other algorithm is guaranteed to expand fewer nodes than A*.
- room for improvement: use better heuristic function!

Forward State-Space Search with A*

•A* is optimally efficient: For a given heuristic function, no other algorithm is guaranteed to expand fewer nodes than A*.

  •all planning algorithms seen so far use search

  •given an admissible heuristic and the need for a minimal length plan, we cannot do better than A*

  •caveats: only have non-admissible heuristic; do not need optimal solution; not enough memory

•room for improvement: use better heuristic function!

  •perfect heuristic uses linear time and memory

  •often: expensive but more accurate heuristic works better

Planning Graph Heuristics

- basic idea: use reachability graph analysis as a heuristic for forward search
  - $P = (A,s_i,g)$ be a propositional planning problem and $G = (N,E)$ the corresponding planning graph
  - $g = \{g_1,\ldots,g_n\}$
  - g_k, $k \in [1,n]$, is reachable from s_i if there is a proposition layer P_g such that $g_k \in P_g$
  - in proposition layer P_m: if g_k not in P_m then g_k not reachable in m steps
- define (admissible) $h_{PG}(g_k) = m$ for reachable $\{g_k\}$

Planning Graph Heuristics

•basic idea: use reachability analysis as a heuristic for forward search

•$P = (A,s_i,g)$ be a propositional planning problem and $G = (N,E)$ the corresponding planning graph

•$g = \{g_1,\ldots,g_n\}$

•g_k, $k \in [1,n]$, is reachable from s_i if there is a proposition layer P_g such that $g_k \in P_g$

•reverse statement:

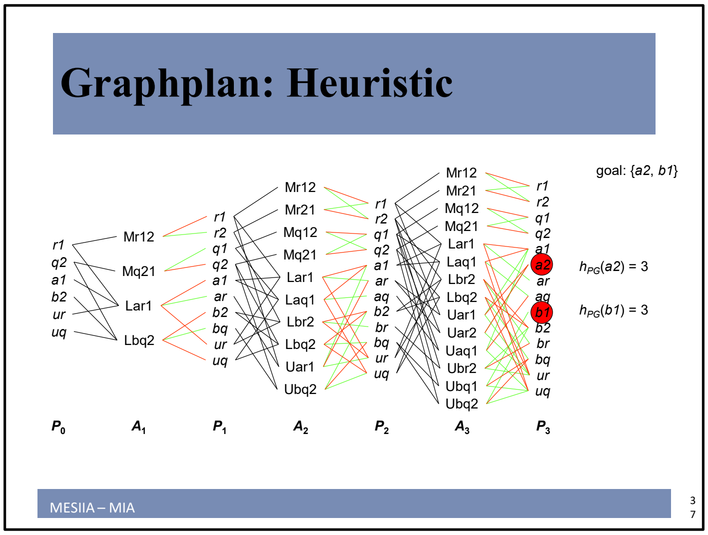
•in proposition layer P_m: if g_k not in P_m then g_k not reachable in m steps

•look for first proposition layer in which g_k appears

•define $h_{PG}(g_k) = m$ for reachable g_k

•works only for single goal condition

•inaccurate if multiple actions from preceding layers are required (but need at least one action from each layer)



Graphplan: Heuristic

•**goal: {*a2*, *b1*}**

- goal consists of two propositions

•***hPG*(*a2*) = 3**

- first proposition layer in which *a2* holds

•***hPG*(*b1*) = 3**

- first proposition layer in which *b1* holds

Overview

- Simple Planning Graph Heuristics
- **The FF (Fast Forward) Planner**

MESIIA – MIA

38

Overview

•Simple Planning Graph Heuristics

•Pattern Database Heuristics

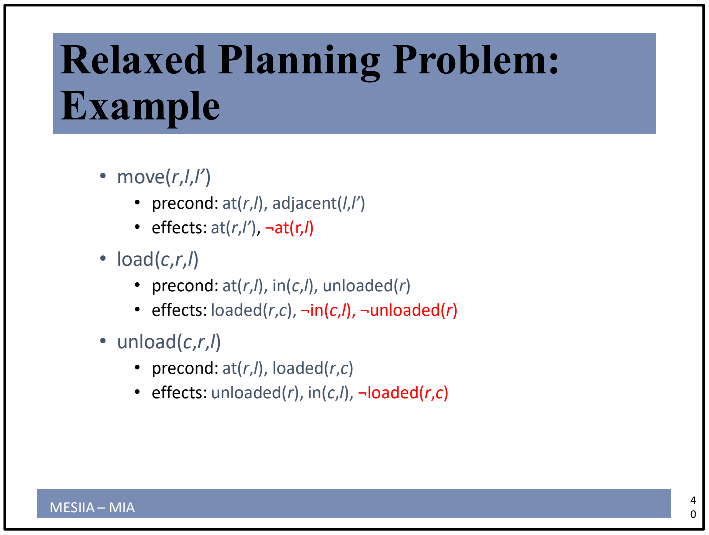
➢The FF Planner

The FF Planner

- performs forward state-space search (A*)
- relaxed problem heuristic (h^{FF})
  - construct relaxed problem: ignore delete lists
  - solve relaxed problem (in polynomial time)
    - chain forward to build a relaxed planning graph
    - chain backward to extract a relaxed plan from the graph
  - use length of relaxed plan as heuristic value

The FF Planner

- **performs forward state-space search (A* / EHC)**
  - EHC: commit first to better state; does not work well if state space has dead ends
- **relaxed problem heuristic (*h*^{FF})**
  - **construct relaxed problem: ignore delete lists**
    - Joerg's example: have a beer, drink the beer, have the beer in tummy, still have a beer!
  - **solve relaxed problem (in polynomial time)**
    - **chain forward to build a relaxed planning graph**
    - **chain backward to extract a relaxed plan from the graph**
  - **use length of relaxed plan as heuristic value**
- **pruned search with helpful actions**
  - use information gained during the computation of the heuristic value



Relaxed Planning Problem: Example

- **move(*r*,*l*,*l’*)**
  - **precond: at(*r*,*l*), adjacent(*l*,*l’*)**
  - **effects: at(*r*,*l’*), ¬at(r,*l*)**
  - robot now in two places
- **load(*c*,*r*,*l*)**
  - **precond: at(*r*,*l*), in(*c*,*l*), unloaded(*r*)**
  - **effects: loaded(*r*,*c*), ¬in(*c*,*l*), ¬unloaded(*r*)**
  - container now in two places
- **unload(*c*,*r*,*l*)**
  - **precond: at(*r*,*l*), loaded(*r*,*c*)**
  - **effects: unloaded(*r*), in(*c*,*l*), ¬loaded(*r*,*c*)**
  - container again in two places

Computing h^{FF}: Relaxed Planning Graph

function computeRPG(A,s_i,g)
  $F_0 \leftarrow s_i$; $t \leftarrow 0$
  while $g \nsubseteq F_t$ **do**
    $t \leftarrow t+1$
    $A_t \leftarrow \{a \in A \mid \text{precond}(a) \subseteq F_{t-1}\}$
    $F_t \leftarrow F_{t-1}$
    for all $a \in A_t$ **do**
      $F_t \leftarrow F_t \cup \text{effects}^+(a)$
    if $F_t = F_{t-1}$ **then return** failure
  return $[F_0, A_1, F_1, \ldots, A_t, F_t]$

Computing h^{FF}: Relaxed Planning Graph

- **function computeRPG(A,s_i,g)**
    - arguments: propositional planning problem (again)
- $F_0 \leftarrow s_i$; $t \leftarrow 0$
- **while** $g \nsubseteq F_t$ **do**
- $t \leftarrow t+1$
- $A_t \leftarrow \{a \in A \mid \text{precond}(a) \subseteq F_t\}$
- $F_t \leftarrow F_{t-1}$
- **for all** $a \in A_t$ **do**
- $F_t \leftarrow F_t \cup \text{effects}^+(a)$
- **if** $F_t = F_{t-1}$ **then return failure**
- **return** $[F_0, A_1, F_1, \ldots, A_t, F_t]$
- similar to planning graph expansion
    - no mutex relations needed
    - stops when goal first appears

Computing h^{FF}: Extracting a Relaxed Plan

function extractRPSize($[F_0, A_1, F_1, \ldots, A_k, F_k]$, g)
  if $g \not\subseteq F_k$ **then return** failure
  $M \leftarrow \max\{\text{firstlevel}(g_i, [F_0,\ldots,F_k]) \mid g_i \in g\}$
  for $t \leftarrow 0$ **to** M **do**
    $G_t \leftarrow \{g_i \in g \mid \text{firstlevel}(g_i, [F_0,\ldots,F_k]) = t)$
  for $t \leftarrow M$ **to** 1 **do**
    for all $g_t \in G_t$ **do**
      select a : firstlevel(a, $[A_1,\ldots,A_t]$) = t **and** $g_t \in$ effects$^+$(a)
      for all $p \in$ precond(a) **do**
        $G_{\text{firstlevel}(p, [F0,\ldots,Fk])} \leftarrow G_{\text{firstlevel}(p, [F0,\ldots,Fk])} \cup \{p\}$
  return number of selected actions

Computing h^{FF}: Extracting a Relaxed Plan

- **function extractRPSize($[F_0, A_1, F_1, \ldots, A_k, F_k]$, g)**
  - arguments: planning graph and goal
- **if $g \not\subseteq F_k$ then return failure**
- **$M \leftarrow \max\{\text{firstlevel}(g_i, [F_0,\ldots,F_k]) \mid g_i \in g\}$**
  - function firstlevel: computes level in PG where proposition first appears
- **for $t \leftarrow 0$ to M do**
- **$G_t \leftarrow \{g_i \in g \mid \text{firstlevel}(g_i, [F_0,\ldots,F_k]) = t)$**
  - start with goals in level where they first appear
- **for $t \leftarrow M$ to 1 do**
- **for all $g_t \in G_t$ do**
- **select a : firstlevel(a, $[A_1,\ldots,A_t]$) = t and $g_t \in$ effects$^+$(a)**
  - commit to selected action (no backtracking)
- **for all $p \in$ precond(a) do**
- **$G_{\text{firstlevel}(p, [F0,\ldots,Fk])} \leftarrow G_{\text{firstlevel}(p, [F0,\ldots,Fk])} \cup \{p\}$**
  - sub-goals in levels where they first appear
- **return number of selected actions**
- runs in polynomial time

End