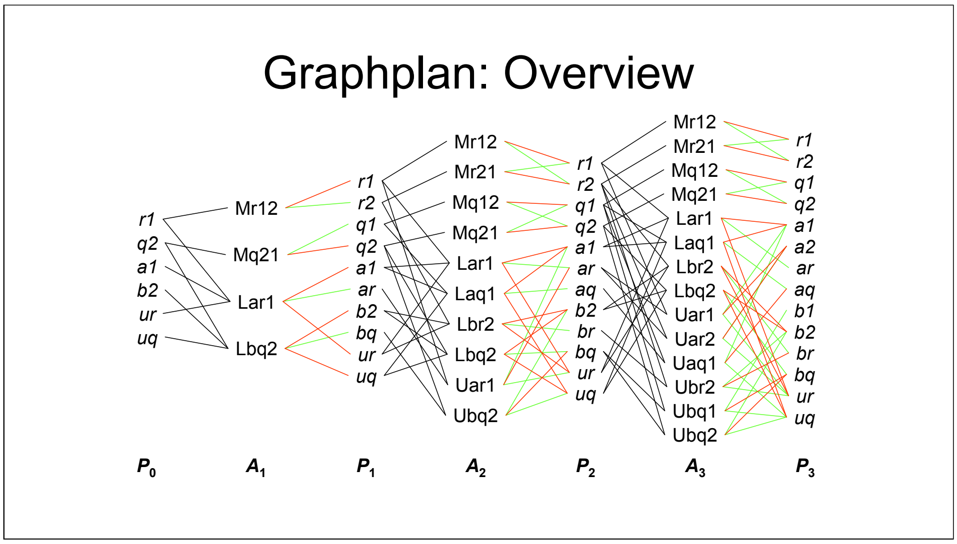
Artificial Intelligence Planning

Graphplan

Artificial Intelligence Planning

- **Graphplan**



Graphplan: Overview

•given a propositional planning domain and problem

•step 1: extend the graph with 2 layers (forward, left to right)

- edges shown are preconditions and effects
- other edges (not shown) express mutual exclusivity
- worst-case time complexity is polynomial

•step 2: search for a plan in the graph

- search backwards (right to left)
- worst-case time complexity is exponential

•repeat steps 1 and 2

Overview

- A Propositional DWR Example
- The Basic Planning Graph (No Mutex)
- Layered Plans
- Mutex Propositions and Actions
- Forward Planning Graph Expansion
- Backwards Search in the Planning Graph
- The Graphplan Algorithm

Overview

➢A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

•Layered Plans

•Mutex Propositions and Actions

•Forward Planning Graph Expansion

•Backwards Search in the Planning Graph

•The Graphplan Algorithm

Classical Representations

- propositional representation
  - world state is set of propositions
  - action consists of precondition propositions, propositions to be added and removed
- STRIPS representation
  - like propositional representation, but first-order literals instead of propositions
- state-variable representation
  - state is tuple of state variables $\{x_1,\ldots,x_n\}$
  - action is partial function over states

Classical Representations

•**propositional representation**

•**world state is set of propositions**

•**action consists of precondition propositions, propositions to be added and removed**

•**STRIPS representation**

•named after STRIPS planner

•**like propositional representation, but first-order literals instead of propositions**

•most popular for restricted state-transitions systems

•**state-variable representation**

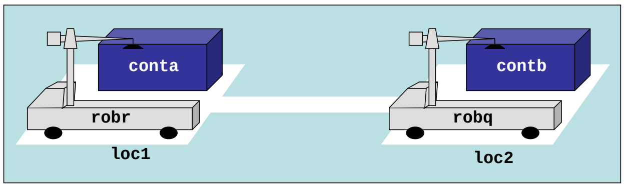
•**state is tuple of state variables $\{x_1,\ldots,x_n\}$**

•**action is partial function over states**

•useful where state is characterized by attributes over finite domains

•equally expressive: planning domain in one representation can also be represented in the others

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 Operators

- move(*r*,*l*,*l'*)
  - precond: at(*r*,*l*), adjacent(*l*,*l'*)
  - effects: at(*r*,*l'*), ¬at(r,*l*)
- load(*c*,*r*,*l*)
  - precond: at(*r*,*l*), in(*c*,*l*), unloaded(*r*)
  - effects: loaded(*r*,*c*), ¬in(*c*,*l*), ¬unloaded(*r*)
- unload(*c*,*r*,*l*)
  - precond: at(*r*,*l*), loaded(*r*,*c*)
  - effects: unloaded(*r*), in(*c*,*l*), ¬loaded(*r*,*c*)

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

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

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, Lbr1, Lbr2, Lbq1, and Lbq2 correspondingly
- unload actions:
  - Uar1: unload(conta,robr,loc1); Uar2, Uaq1, Uaq2, Ubr1, Ubr2, Ubq1, and Ubq2 correspondingly

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
- Forward Planning Graph Expansion
- Backwards Search in the Planning Graph
- The Graphplan Algorithm

Overview

•A Propositional DWR Example

➢The Basic Planning Graph (No Mutex)

•Layered Plans

•Mutex Propositions and Actions

•Forward Planning Graph Expansion

•Backwards Search in the Planning Graph

•The Graphplan Algorithm

Solution Existence

- **Proposition**: A propositional planning problem $\mathcal{P}=(\Sigma,s_i,g)$ has a solution iff
  $$S_g \cap \Gamma^{>}(\{s_i\}) \neq \{\}.$$
- **Proposition**: A propositional planning problem $\mathcal{P}=(\Sigma,s_i,g)$ has a solution iff
  $$\exists s \in \Gamma^{<}(\{g\}) : s \subseteq s_i.$$

Solution Existence

•**Proposition: A propositional planning problem $\mathcal{P}=(\Sigma,s_i,g)$ has a solution iff $S_g \cap \Gamma^{>}(\{s_i\}) \neq \{\}$.**

 •… iff there is a goal state that is also a reachable state

•**Proposition: A propositional planning problem $\mathcal{P}=(\Sigma,s_i,g)$ has a solution iff $\exists s \in \Gamma^{<}(\{g\}) : s \subseteq s_i$.**

 •… iff there is a minimal set of propositions amongst all regression sets that is a subset of the initial state

Reachability Tree

- tree structure, where:
  - root is initial state s_i
  - children of node s are $\Gamma(\{s\})$
  - arcs are labelled with actions
- all nodes in reachability tree are $\Gamma^{>}(\{s_i\})$
  - all nodes to depth d are $\Gamma^{d}(\{s_i\})$
  - solves problems with up to d actions in solution
- problem: $O(k^d)$ nodes;
  k = applicable actions per state

Reachability Tree

•tree structure, where:

•root is initial state s_i

•children of node s are $\Gamma(\{s\})$

•arcs are labelled with actions

•all nodes in reachability tree are $\Gamma^{>}(\{s_i\})$

•all nodes to depth d are $\Gamma^{d}(\{s_i\})$

•solves problems with up to d actions in solution

•problem: $O(k^d)$ nodes;
k = applicable actions per state

Planning Graph: Nodes

- layered directed graph G=(N,E):
  - $N = P_0 \cup A_1 \cup P_1 \cup A_2 \cup P_2 \cup \ldots$
    - state proposition layers: P_0, P_1, …
    - action layers: A_1, A_2, …
- 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)$\subseteq P_{j\text{-}1}$
- proposition layer P_j:
  - all propositions p where: $p \in P_{j\text{-}1}$ or $\exists a \in A_j$: $p \in$effects$^+$(a)

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, …**
    - **action layers: A_1, A_2, …**
- **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)$\subseteq P_{j\text{-}1}$**
- **proposition layer P_j:**
  - **all propositions p where: $p \in P_{j\text{-}1}$ 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: $P_{j\text{-}1} \subseteq P_j$; propositions in the graph monotonically 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)
- no arcs between other layers

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)

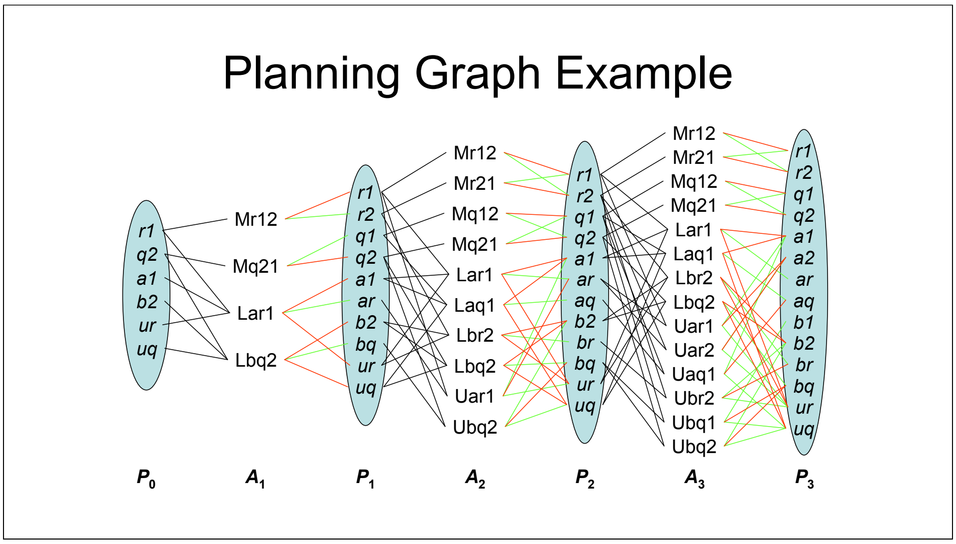
•from action $a \in A_j$ to layer $p \in P_j$:

•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



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 $g \subseteq P_g$
- necessary condition, but not sufficient
- low complexity:
  - planning graph is of polynomial size and
  - can be computed in polynomial time

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:

•reachability tree:

•nodes contain propositions that must necessarily hold

•propositions in one node are consistent

•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
- Forward Planning Graph Expansion
- Backwards Search in the Planning Graph
- The Graphplan Algorithm

Overview

•A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

➢Layered Plans

•Mutex Propositions and Actions

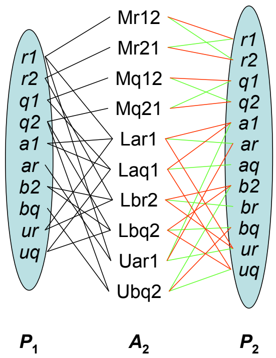
•Forward Planning Graph Expansion

•Backwards Search in the Planning Graph

•The Graphplan Algorithm

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.

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)
  for all $p \in$ effects$^-$(a_1)
    if $p \in$ precond(a_2) **or** $p \in$ effects$^+$(a_2) **then**
      return false
  for all $p \in$ effects$^-$(a_2)
    if $p \in$ precond(a_1) **or** $p \in$ effects$^+$(a_1) **then**
      return false
  return true

Pseudo Code: independent

- **function independent(a_1,a_2)**
  - returns true iff the two given actions are independent
- **for all $p \in$ effects$^-$(a_1)**
- **if $p \in$ precond(a_2) or $p \in$ effects$^+$(a_2) then**
- **return false**
- **for all $p \in$ effects$^-$(a_2)**
- **if $p \in$ precond(a_1) or $p \in$ 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: $O(b)$

Applying Independent Actions

- A set π of independent actions is <u>applicable</u> to a state s iff $\cup_{a \in \pi}$precond(a) $\subseteq s$.
- The <u>result</u> of applying the set π in s is defined as: $\gamma(s,\pi) = (s - \text{effects}^-(\pi)) \cup \text{effects}^+(\pi)$, where:
  – precond(π) = $\cup_{a \in \pi}$precond(a),
  – effects$^+$(π) = $\cup_{a \in \pi}$effects$^+$(a), and
  – effects$^-$(π) = $\cup_{a \in \pi}$effects$^-$(a).

Applying Independent Actions

•**A set π of independent actions is <u>applicable</u> to a state s iff $\cup_{a \in \pi}$precond(a) $\subseteq s$.**

•note: applying a set of independent actions can be done in any order

•**The <u>result</u> of applying the set π in s is defined as:**
$\gamma(s,\pi) = (s - \text{effects}^-(\pi)) \cup \text{effects}^+(\pi)$, where:

•**precond(π) = $\cup_{a \in \pi}$precond(a),**

•**effects$^+$(π) = $\cup_{a \in \pi}$effects$^+$(a), and**

•**effects$^-$(π) = $\cup_{a \in \pi}$effects$^-$(a).**

Execution Order of Independent Actions

- **Proposition**: If a set π of independent actions is applicable in state s then, for any permutation $\langle a_1,\ldots,a_k\rangle$ of the elements of π:
  - the sequence $\langle a_1,\ldots,a_k\rangle$ is applicable to s, and
  - the state resulting from the application of π to s is the same as from the application of $\langle a_1,\ldots,a_k\rangle$, i.e.: $\gamma(s,\pi) = \gamma(s,\langle a_1,\ldots,a_k\rangle)$.

Execution Order of Independent Actions

•Proposition: If a set π of independent actions is applicable in state s then, for any permutation $\langle a_1,\ldots,a_k\rangle$ of the elements of π:

•the sequence $\langle a_1,\ldots,a_k\rangle$ is applicable to s, and

•the state resulting from the application of π to s is the same as from the application of $\langle a_1,\ldots,a_k\rangle$, i.e.:
$\gamma(s,\pi) = \gamma(s,\langle a_1,\ldots,a_k\rangle)$.

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 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 $\prod = \langle \pi_1,\ldots,\pi_k \rangle$ 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
  - $g \subseteq \gamma(\ldots\gamma(\gamma(s_i,\pi_1), \pi_2), \ldots, \pi_k)$.

Layered Solution Plan

•**A layered plan $\prod = \langle \pi_1,\ldots,\pi_k \rangle$ 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
- Forward Planning Graph Expansion
- Backwards Search in the Planning Graph
- The Graphplan Algorithm

Overview

•A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

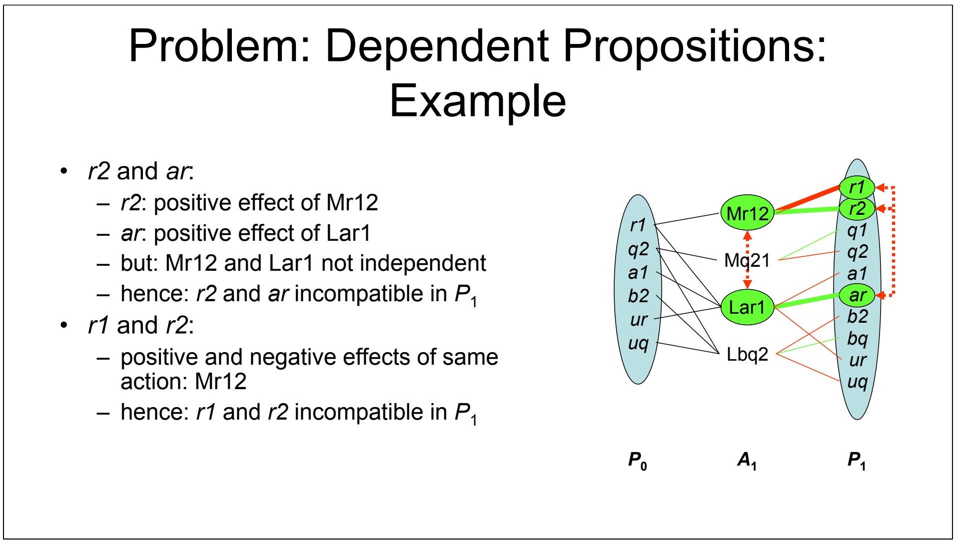
•Layered Plans

➢Mutex Propositions and Actions

•Forward Planning Graph Expansion

•Backwards Search in the Planning Graph

•The Graphplan Algorithm



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

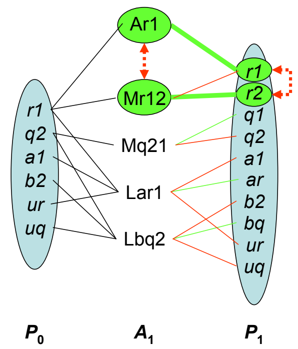
  •two positive effects of one action

  •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*:
  - 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



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
  - there is no single action in A_j that has both, *p* and *q*, as positive effects.
- notation: μP_j = { (p,q) | $p,q \in P_j$ are mutex}

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: μP_j = { (p,q) | $p,q \in P_j$ 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(p1,p2,μAj)
  for all a1∈p1.producers()
    for all a2∈p2.producers()
      if (a1,a2)∉μAj then
        return false
  return true
```

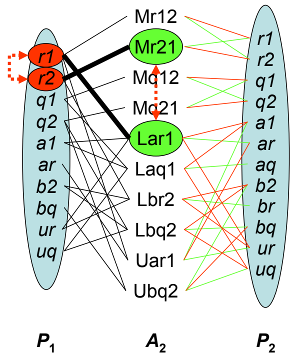
Pseudo Code: mutex for Propositions

- **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: 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}

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: $\mu A_j = \{ (a_1,a_2) \mid a_1,a_2 \in A_j \text{ are mutex}\}$

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:**
$\mu A_j = \{ (a_1,a_2) \mid a_1,a_2 \in A_j \text{ are mutex}\}$

•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)

Pseudo Code: mutex for Actions

function mutex(a_1,a_2,μP)
  if ¬independent(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

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: $O(b^2)$

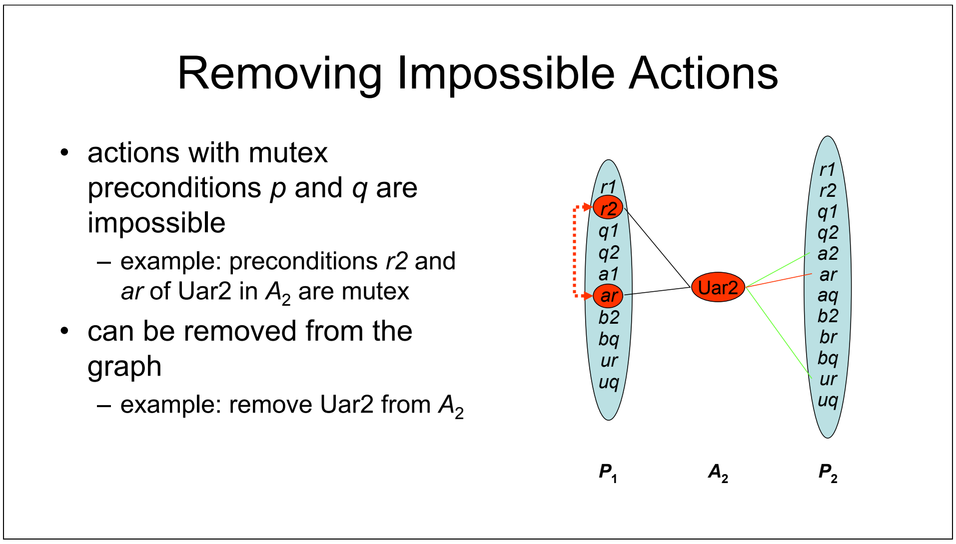
Decreasing Mutex Relations

- **Proposition**: If $p,q \in P_{j-1}$ and $(p,q) \notin \mu P_{j-1}$ then $(p,q) \notin \mu P_j$.
  - Proof:
    - if $p,q \in P_{j-1}$ then Ap,Aq$\in A_j$
    - if $(p,q) \notin \mu P_{j-1}$ then (Ap,Aq)$\notin \mu A_j$
    - since Ap,Aq$\in A_j$ and (Ap,Aq)$\notin \mu A_j$, $(p,q) \notin \mu P_j$ must hold
- **Proposition**: If $a_1,a_2 \in A_{j-1}$ and $(a_1,a_2) \notin \mu A_{j-1}$ then $(a_1,a_2) \notin \mu A_j$.
  - Proof:
    - if $a_1,a_2 \in A_{j-1}$ and $(a_1,a_2) \notin \mu A_{j-1}$ then
      - a_1 and a_2 are independent and
      - their preconditions in P_{j-1} are not mutex
    - both properties remain true for P_j
    - hence: $a_1,a_2 \in A_j$ and $(a_1,a_2) \notin \mu A_j$

Decreasing Mutex Relations

- **Proposition: If $p,q \in P_{j-1}$ and $(p,q) \notin \mu P_{j-1}$ then $(p,q) \notin \mu P_j$.**
  - Proof:
    - if $p,q \in P_{j-1}$ then Ap,Aq$\in A_j$
    - if $(p,q) \notin \mu P_{j-1}$ then (Ap,Aq)$\notin \mu A_j$
    - since Ap,Aq$\in A_j$ and (Ap,Aq)$\notin \mu A_j$, $(p,q) \notin \mu P_j$ **must hold**
- **Proposition: If $a_1,a_2 \in A_{j-1}$ and $(a_1,a_2) \notin \mu A_{j-1}$ then $(a_1,a_2) \notin \mu A_j$.**
  - **Proof:**
    - **if $a_1,a_2 \in A_{j-1}$ and $(a_1,a_2) \notin \mu A_{j-1}$ then**
      - **a_1 and a_2 are independent and**
      - **their preconditions in P_{j-1} are not mutex**
    - **both properties remain true for P_j**
    - **hence: $a_1,a_2 \in A_j$ and $(a_1,a_2) \notin \mu A_j$**

- mutex relations are monotonically decreasing (between layers with the same propositions)



Removing Impossible Actions

•actions with mutex preconditions *p* and *q* are impossible

        •example: preconditions *r2* and *ar* of Uar2 in A_2 are mutex

•action with mutex preconditions can never be part of any layered plan (will violate applicability condition in definition)

•can be removed from the graph

        •example: remove Uar2 from A_2

•mutex pair of actions must remain in graph because one of the actions may be used in final plan

•note: still consistent with monotonically increasing actions

Overview

- A Propositional DWR Example
- The Basic Planning Graph (No Mutex)
- Layered Plans
- Mutex Propositions and Actions
- Forward Planning Graph Expansion
- Backwards Search in the Planning Graph
- The Graphplan Algorithm

Overview

•A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

•Layered Plans

•Mutex Propositions and Actions

➢Forward Planning Graph Expansion

•Backwards Search in the Planning Graph

•The Graphplan Algorithm

Reachability in Planning Graphs

- **Proposition**: Let $P = (A,s_i,g)$ be 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. If
  - g is reachable from s_i

  then
  - there is a proposition layer P_g such that
    - $g \subseteq P_g$ and
    - $\neg\exists\, g_1,g_2 \in g$: $(g_1,g_2) \in \mu P_g$.

Reachability in Planning Graphs

•**Proposition: Let $P = (A,s_i,g)$ be 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. If**

•**g is reachable from s_i**

• **then**

•**there is a proposition layer P_g such that**

•**$g \subseteq P_g$ and**

•**$\neg\exists\, g_1,g_2 \in g$: $(g_1,g_2) \in \mu P_g$.**

•still only necessary condition, but relatively efficient to compute

The Graphplan Algorithm: Basic Idea

- expand the planning graph, one action layer and one proposition layer at a time
- from the first graph for which P_g is the last proposition layer such that
  - $g \subseteq P_g$ and
  - $\neg\exists\ g_1,g_2 \in g$: $(g_1,g_2) \in \mu P_g$
- search backwards from the last (proposition) layer for a solution

The Graphplan Algorithm: Basic Idea

•expand the planning graph, one action layer and one proposition layer at a time

•similar to iterative deepening: discover new part of the search space with each iteration

•from the first graph for which P_g is the last proposition layer such that

•$g \subseteq P_g$ and

•$\neg\exists\ g_1,g_2 \in g$: $(g_1,g_2) \in \mu P_g$

•no need to search for solutions in graph with fewer layers; see last proposition

•search backwards from the last (proposition) layer for a solution

•two major steps:

•expansion of planning graph to next proposition layer

•searching a given planning graph for a solution

Planning Graph Data Structure

- k-th planning graph G_k:
  - nodes N:
    - array of proposition layers $P_0 \ldots P_k$
      - proposition layer j: set of proposition symbols
    - array of action layers $A_1 \ldots A_k$
      - proposition layer j: set of action symbols
  - edges E:
    - precondition links: $pre_j \subseteq P_{j-1} \times A_j$, $j \in \{1 \ldots k\}$
    - positive effect links: $e_j^+ \subseteq A_j \times P_j$, $j \in \{1 \ldots k\}$
    - negative effect links: $e_j^- \subseteq A_j \times P_j$, $j \in \{1 \ldots k\}$
    - proposition mutex links: $\mu P_j \subseteq P_j \times P_j$, $j \in \{1 \ldots k\}$
    - action mutex links: $\mu A_j \subseteq A_j \times A_j$, $j \in \{1 \ldots k\}$

Planning Graph Data Structure

•k-th planning graph G_k:

•nodes N:

•array of proposition layers $P_0 \ldots P_k$

•proposition layer j: set of proposition symbols

•array of action layers $A_1 \ldots A_k$

•proposition layer j: set of action symbols

•edges E:

•precondition links: $pre_j \subseteq P_{j-1} \times A_j$, $j \in \{1 \ldots k\}$

•positive effect links: $e_j^+ \subseteq A_j \times P_j$, $j \in \{1 \ldots k\}$

•negative effect links: $e_j^- \subseteq A_j \times P_j$, $j \in \{1 \ldots k\}$

•proposition mutex links: $\mu A_j \subseteq A_j \times A_j$, $j \in \{1 \ldots k\}$

•action mutex links: $\mu P_j \subseteq P_j \times P_j$, $j \in \{1 \ldots k\}$

•note: instance of this data structure does not depend on problem

•initial planning graph: $P_0 = s_i$; rest is empty sets

Pseudo Code: expand

function expand(G_{k-1})
$A_k \leftarrow \{a \in A \mid \text{precond}(a) \subseteq P_{k-1}$ and
$\{(p_1,p_2) \mid p_1,p_2 \in \text{precond}(a)\} \cap \mu P_{k-1} = \{\}\ \}$
$\mu A_k \leftarrow \{(a_1,a_2) \mid a_1,a_2 \in A_k,\ a_1 \neq a_2$, and $\text{mutex}(a_1,a_2,\mu P_{k-1})\ \}$
$P_k \leftarrow \{p \mid \exists a \in A_k : p \in \text{effects}^+(a)\ \}$
$\mu P_k \leftarrow \{(p_1,p_2) \mid p_1,p_2 \in P_k,\ p_1 \neq p_2$, and $\text{mutex}(p_1,p_2,\mu A_k)\ \}$
for all $a \in A_k$
$pre_k \leftarrow pre_k \cup (\{p \mid p \in P_{k-1}$ and $p \in \text{precond}(a)\} \times a)$
$e_k^+ \leftarrow e_k^+ \cup (a \times \{p \mid p \in P_k$ and $p \in \text{effects}^+(a)\})$
$e_k^- \leftarrow e_k^- \cup (a \times \{p \mid p \in P_k$ and $p \in \text{effects}^-(a)\})$

Pseudo Code: expand

- **function expand(G_{k-1})**
- **$A_k \leftarrow \{a \in A \mid \text{precond}(a) \subseteq P_{k-1}$ and $\{(p_1,p_2) \mid p_1,p_2 \in \text{precond}(a)\} \cap \mu P_{k-1} = \{\}\ \}$**
  - actions with satisfied, non-mutex preconditions (incl. no-ops)
- **$\mu A_k \leftarrow \{(a_1,a_2) \mid a_1,a_2 \in A_k,\ a_1 \neq a_2$, and $\text{mutex}(a_1,a_2,\mu P_{k-1})\ \}$**
- **$P_k \leftarrow \{p \mid \exists a \in A_k : p \in \text{effects}^+(a)\ \}$**
  - union of all positive effects
- **$\mu P_k \leftarrow \{(p_1,p_2) \mid p_1,p_2 \in P_k,\ p_1 \neq p_2$, and $\text{mutex}(p_1,p_2,\mu A_k)\ \}$**
- **for all $a \in A_k$**
- **$pre_k \leftarrow pre_k \cup (\{p \mid p \in P_{k-1}$ and $p \in \text{precond}(a)\} \times a)$**
- **$e_k^+ \leftarrow e_k^+ \cup (a \times \{p \mid p \in P_k$ and $p \in \text{effects}^+(a)\})$**
- **$e_k^- \leftarrow e_k^- \cup (a \times \{p \mid p \in P_k$ and $p \in \text{effects}^-(a)\})$**

Planning Graph Complexity

- **Proposition**: The size of a planning graph up to level *k* and the time required to expand it to that level are polynomial in the size of the planning problem.
- Proof:
  - problem size: *n* propositions and *m* actions
  - $|P_j| \leq n$ and $|A_j| \leq n+m$ (incl. no-op actions)
  - algorithms for generating each layer and all link types are polynomial in size of layer

Planning Graph Complexity

•Proposition: The size of a planning graph up to level *k* and the time required to expand it to that level are polynomial in the size of the planning problem.

•Proof:

- **problem size: *n* propositions and *m* actions**
- **$|P_j| \leq n$ and $|A_j| \leq n+m$ (incl. no-op actions)**
- **algorithms for generating each layer and all link types are polynomial in size of layer**

Fixed-Point Levels

- A <u>fixed-point level</u> in a planning graph G is a level κ such that for all i, $i>\kappa$, level i of G is identical to level κ, i.e. $P_i=P_\kappa$, $\mu P_i=\mu P_\kappa$, $A_i=A_\kappa$, and $\mu A_i=\mu A_\kappa$.

- **Proposition**: Every planning graph G has a fixed-point level κ, which is the smallest k such that $|P_k|=|P_{k+1}|$ and $|\mu P_k|=|\mu P_{k+1}|$.
- Proof:
  - P_i grows monotonically and μP_i shrinks monotonically
  - A_i and P_i only depend on P_{i-1} and μP_{i-1}

Fixed-Point Levels

•**A <u>fixed-point level</u> in a planning graph G is a level κ such that for all i, $i>\kappa$, level i of G is identical to level κ, i.e. $P_i=P_\kappa$, $\mu P_i=\mu P_\kappa$, $A_i=A_\kappa$, and $\mu A_i=\mu A_\kappa$.**

•**Proposition: Every planning graph G has a fixed-point level κ, which is the smallest k such that $|P_k|=|P_{k+1}|$ and $|\mu P_k|=|\mu P_{k+1}|$.**

    •$|P_k|=|P_{k+1}|$ implies $P_k=P_{k+1}$

•**Proof:**

    •**P_i grows monotonically and μP_i shrinks monotonically**

        •μP_i shrinks monotonically: for equal P_i

    •**A_i and P_i only depend on P_{i-1} and μP_{i-1}**

•time complexity: $O(n+m)$ from fixed point level; only copying required

Overview

- A Propositional DWR Example
- The Basic Planning Graph (No Mutex)
- Layered Plans
- Mutex Propositions and Actions
- Forward Planning Graph Expansion
- Backwards Search in the Planning Graph
- The Graphplan Algorithm

Overview

•A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

•Layered Plans

•Mutex Propositions and Actions

•Forward Planning Graph Expansion

➢Backwards Search in the Planning Graph

•The Graphplan Algorithm

Searching the Planning Graph

- general idea:
  - search backwards from the last proposition layer P_k in the current graph
  - let g be the set of goal propositions that need to be achieved at a given proposition layer P_j (initially the last layer)
  - find a set of actions $\pi_j \subseteq A_j$ such that these actions are not mutex and together achieve g
  - take the union of the preconditions of π_j as the new goal set to be achieved in proposition layer $P_{j\text{-}1}$

Searching the Planning Graph

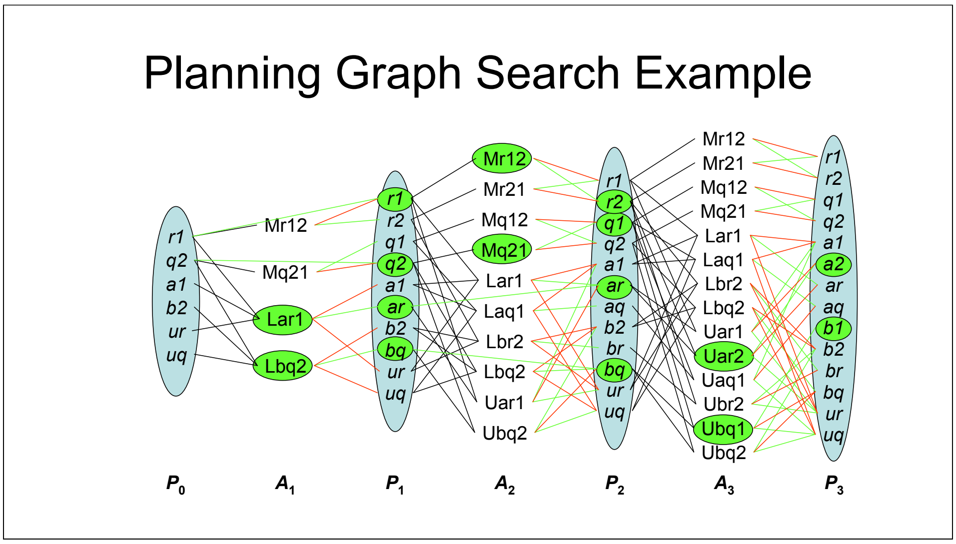
•general idea:

•search backwards from the last proposition layer P_k in the current graph

•let g be the set of goal propositions that need to be achieved at a given proposition layer P_j (initially the last layer)

•find a set of actions $\pi_j \subseteq A_j$ such that these actions are not mutex and together achieve g

•take the union of the preconditions of π_j as the new goal set to be achieved in proposition layer $P_{j\text{-}1}$



Planning Graph Search Example

•initial goal: *a2* and *b1*

•only one incoming positive effect link per goal (but no-ops not shown)

•achievable with Uar2 and Ubq1 (which are not mutex; mutex relations not shown)

•precondition links indicate sub-goal at next layer

•new sub-goal at P_2: *r2*, *q1*, *ar*, *bq*

•only one incoming positive effect link per goal condition (but no-ops not shown)

- •achieve *ar* and *bq* with no-ops
- •achieve *r2* with Mr12 and *q1* with Mq21

•precondition links (for Mr12 and Mq21) indicate some sub-goal at next layer

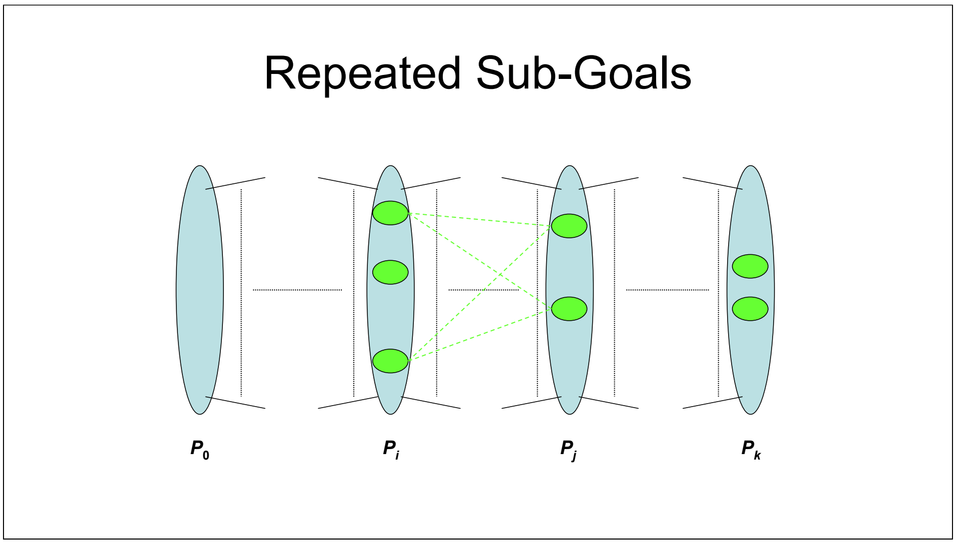
•complete sub-goal (incl. preconditions of no-ops) at P_1: *r1*, *q2*, *ar*, *bq*

•only one incoming positive effect link per goal condition (but no-ops not shown)

- •achieve *r1* and *q2* with no-ops
- •achieve *ar* with Lar1 and *bq* with Lbq2

•precondition links (for Lar1 and Lbq2) indicate some sub-goal at next layer

•complete sub-goal (incl. preconditions of no-ops) at P_0: complete initial state



Repeated Sub-Goals

•ultimate goal leads to possible sub-goals at P_j

•possible sub-goals at P_j lead to possible sub-goals at P_i

  •search to initial proposition layer to see whether sub-goals can be achieved

  •suppose: sub-goals at P_i cannot be achieved

•backtrack to later layer, say P_j

•possible sub-goals at P_j may lead to same possible sub-goals at P_i, but in a different way

  •no need to repeat search: same sub-goals at same layer still cannot be achieved

  •generalization: same some sub-goals at same or earlier layer still cannot be achieved

    •otherwise no-op would achieve sub-goal at later layer

The *nogood* Table

- *nogood* table (denoted ∇) for planning graph up to layer k:
  - array of k sets of sets of goal propositions
    - inner set: one combination of propositions that cannot be achieved
    - outer set: all combinations that cannot be achieved (at that layer)
- before searching for set g in P_j:
  - check whether $g \in \nabla(j)$
- when search for set g in P_j has failed:
  - add g to $\nabla(j)$

The *nogood* Table

- ***nogood* table (denoted ∇) for planning graph up to layer k:**
  - **array of k sets of sets of goal propositions**
    - **inner set: one combination of propositions that cannot be achieved**
    - **outer set: all combinations that cannot be achieved (at that layer)**
  - mutex only gives pairs of propositions that cannot be achieved together, *nogood* table gives impossible tuples
- **before searching for set g in P_j:**
  - **check whether $g \in \nabla(j)$**
  - actually: in j or later layer
- **when search for set g in P_j has failed:**
  - **add g to $\nabla(j)$**
  - or move?

Pseudo Code: extract

function extract(G,g,i)
  if i=0 **then return** $\langle\rangle$
  if $g \in \nabla(i)$ **then return** failure
  $\prod \leftarrow$ gpSearch(G,g,{},i)
  if $\prod \neq$failure **then return** $\prod$
  $\nabla(i) \leftarrow \nabla(i) + g$
  return failure

Pseudo Code: extract

- **function extract(G,g,i)**
  - inputs: planning graph G, set of propositions (sub-goals) g, and layer at which sub-goals need to be achieved i
  - output: a layered plan $\langle \pi_1,\ldots,\pi_i \rangle$ that achieves g at i in G or failure if there is no such plan
- **if i=0 then return** $\langle\rangle$
  - trivial success with empty plan
- **if $g \in \nabla(i)$ then return failure**
  - sub-goals have resulted in failure before
- **$\pi_i \leftarrow$ gpSearch(G,g,{},i)**
  - perform the search
- **if $\pi_i \neq$failure then return π_i**
  - the search was successful
- **$\nabla(i) \leftarrow \nabla(i) + g$**
  - unsuccessful search: remember unachievable sub-goals
- **return failure**

Pseudo Code: gpSearch

function gpSearch(G,g,π,i)
  if g={} **then**
    $\prod \leftarrow$ extract(G,$\cup_{a\in\pi}$precond(a),i-1)
    if $\prod$ =failure **then return** failure
    return $\prod \bullet\langle\pi\rangle$
  $p \leftarrow g$.selectOne()
  $providers \leftarrow \{a\in A_i \mid p\in$effects$^+(a)$ and $\neg\exists a'\in\pi$: $(a,a')\in\mu A_i\}$
  if $providers$={} **then return** failure
  $a \leftarrow providers$.chooseOne()
  return gpSearch(G,g-effects$^+(a)$,$\pi+a$,i)

Pseudo Code: gpSearch

- **function gpSearch(G,g,π,i)**
  - inputs: planning graph G, remaining sub-goals g, and set of actions already committed to π, both at level i
  - outputs: layered plan
- **if g={} then**
  - all actions chosen
- $\prod \leftarrow$ extract(G,$\cup_{a\in\pi}$**precond(a),i-1)**
- **if $\prod$=failure then return failure**
- **return $\prod\bullet\langle\pi\rangle$**
- **$p \leftarrow g$.selectOne()**
  - no need to backtrack here; order only important for efficiency
- **resolvers $\leftarrow \{a\in A_i \mid p\in$effects$^+(a)$ and $\neg\exists a'\in\pi$: $(a,a')\in\mu A_i\}$**
- **if *resolvers*={} then return failure**
- **$a \leftarrow$ *resolvers*.chooseOne()**
  - non-deterministic choice point; backtrack to here
- **return GPSearch(G,g-effects$^+(a)$,$\pi+a$,i)**

Overview

- A Propositional DWR Example
- The Basic Planning Graph (No Mutex)
- Layered Plans
- Mutex Propositions and Actions
- Forward Planning Graph Expansion
- Backwards Search in the Planning Graph
- The Graphplan Algorithm

Overview

•A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

•Layered Plans

•Mutex Propositions and Actions

•Forward Planning Graph Expansion

•Backwards Search in the Planning Graph

➢The Graphplan Algorithm

Pseudo Code: Graphplan

function graphplan(A,s_i,g)
  $i \leftarrow 0$; $\nabla \leftarrow []$; $P_0 \leftarrow s_i$; $G \leftarrow (P_0,\{\})$
  while ($g \not\subseteq P_i$ **or** $g^2 \cap \mu P_i \neq \{\}$) **and** $\neg$fixedPoint(G) **do**
    $i \leftarrow i+1$; expand(G)
  if $g \not\subseteq P_i$ **or** $g^2 \cap \mu P_i \neq \{\}$ **then return** failure
  $\eta \leftarrow$ fixedPoint(G) ? $|\nabla(\kappa)|$: 0
  $\prod \leftarrow$ extract(G,g,i)
  while $\prod$=failure **do**
    $i \leftarrow i+1$; expand(G)
    $\prod \leftarrow$ extract(G,g,i)
    if $\prod$=failure **and** fixedPoint(G) **then**
      if $\eta = |\nabla(\kappa)|$ **then return** failure
      $\eta \leftarrow |\nabla(\kappa)|$
  return $\prod$

Pseudo Code: graphplan

- **function graphplan(A,s_i,g)**
  - given planning problem, return layered solution plan
- **$i \leftarrow 0$; $\nabla \leftarrow []$; $P_0 \leftarrow s_i$; $G \leftarrow (P_0,\{\})$**
- **while ($g \not\subseteq P_i$ or $g^2 \cap \mu P_i \neq \{\}$) and $\neg$fixedPoint(G) do**
- **$i \leftarrow i+1$; expand(G)**
  - planning graph expanded until solution possible or fixed point reached
- **if $g \not\subseteq P_i$ or $g^2 \cap \mu P_i \neq \{\}$ then return failure**
  - test necessary criterion
- **$\eta \leftarrow$ fixedPoint(G) ? $|\nabla(\kappa)|$: 0**
  - used to test when expansion will not work
- **$\prod \leftarrow$ extract(G,g,i)**
- **while $\prod$=failure do**
- **$i \leftarrow i+1$; expand(G)**
- **$\prod \leftarrow$ extract(G,g,i)**
- **if $\prod$=failure and fixedPoint(G) then**
- **if $\eta = |\nabla(\kappa)|$ then return failure**
- **$\eta \leftarrow |\nabla(\kappa)|$**
- **return $\prod$**

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 Π that is a solution to the given planning problem.

- Graphplan is orders of magnitude faster than previous techniques!

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** Π that is a solution to the given planning problem.

•Graphplan is orders of magnitude faster than previous techniques!

- •caveat: restriction to propositional STRIPS

Overview

- A Propositional DWR Example
- The Basic Planning Graph (No Mutex)
- Layered Plans
- Mutex Propositions and Actions
- Forward Planning Graph Expansion
- Backwards Search in the Planning Graph
- The Graphplan Algorithm

Overview

•A Propositional DWR Example

•The Basic Planning Graph (No Mutex)

•Layered Plans

•Mutex Propositions and Actions

•Forward Planning Graph Expansion

•Backwards Search in the Planning Graph

•The Graphplan Algorithm