Planning & Scheduling

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Planning Graph and Graphplan

Classical planning

- **search nodes** correspond to **partial plans**
- **a solution plan** reachable from a given search node contains **all the actions** from the partial plan

- state-space planning
- plan-space planning

Neoclassical planning

- **search nodes** correspond to **several partial plans**
- **not all actions** from the partial plans appear in the solution plan reachable from the search node

- planning-graph planning

Problems of existing techniques

- **choice of bad action**, that is discovered late, causes exploration of large search space
- **large branching factor** is the source of such mistakes because it gives too many options

How to discover "promising" actions for the plan?

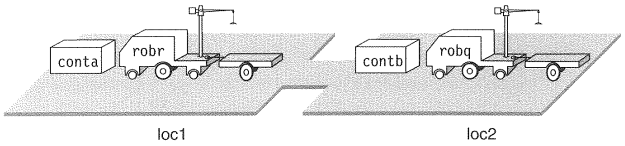
- **solve a „relaxed" problem**, whose solution set contains all solutions of the original problem
  - **relaxed problem** = remove some constraints from the original problem (for example negative effects)
- use only the actions from the solution of the relaxed problem when selecting actions for the original problem

What do we need?

- a compact representation of several plans such that not all the actions has to be used in the solution plan

- Instead of the least-commitment strategy we will use a **strong-commitment strategy**:
  - actions are fully instantiated
  - and their order in the plan is fixed
- Planning-graph planning is based on two techniques:
  - **reachability analysis**
    - we find if a given world state is reachable from the initial state in a given number of steps
  - **disjunctive refinement**
    - flaws are repaired by disjunctive refinement and maintaining interference between the refinements using constraints

Working example



move(r, l, l') ;; robot r at location l moves to a connected location l'
precond: at(r, l), adjacent(l, l')
effects: at(r, l'), ¬ at(r, l)

load(c, r, l) ;; robot r loads container c at location l
precond: at(r, l), in(c, l), unloaded(r)
effects: loaded(r, c), ¬ in(c, l), ¬ unloaded(r)

unload(c, r, l) ;; robot r unloads container c at location l
precond: at(r, l), loaded(r, c)
effects: unloaded(r), in(c, l), ¬ loaded(r, c)

We will work with fully instantiated atoms and actions:

Atoms:

- r_1, r_2, q_1, q_2 – robot positions
- a_1, a_2, a_r, a_q, b_1, b_2, b_r, b_q – container positions
- u_r,u_q – robot is empty

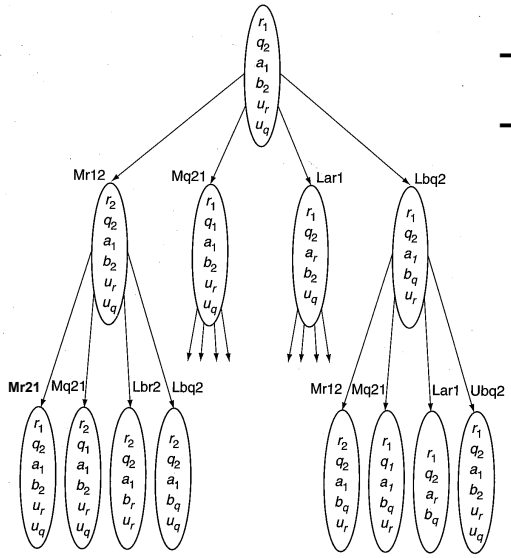
The initial state is {r_1,q_2,a_1,b_2,u_r,u_q}.

Actions:

- Mr12, Mr21, Mq12, Mq21 – robot moves
- Lar1, Lar2, Laq1, Laq2, Lbr1, Lbr2, Lbq1, Lbq2 – loading container to robot
- Uar1, Uar2, Uaq1, Uaq2, Ubr1, Ubr2, Ubq1, Ubq2 – unloading container from robot

Reachability tree

Nodes correspond to states and arcs to transitions



– **root** – initial state s_0

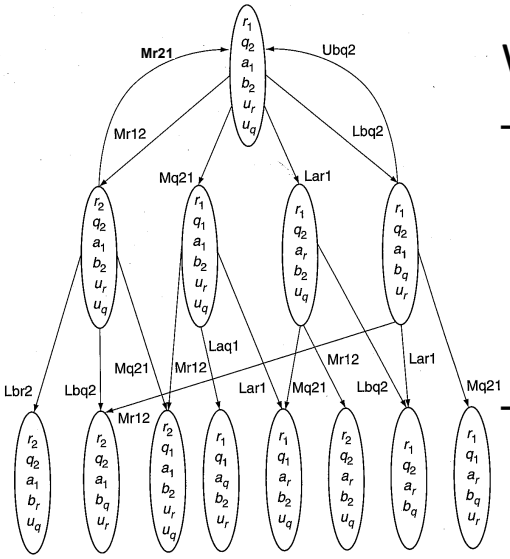
– Reachability tree of depth d with the root s_0 contains **all solution plans** for problems, where the goal is reachable from s_0 with at most d actions.

- **A plan exists if any goal state is in the reachability tree!**

Problem:

The reachability tree contains $O(k^d)$ nodes, where k is #applicable actions per state.

- Some nodes are repeated in the reachability tree (are reachable by different plans). They can be assumed just once!



We get a **reachability graph**

– **Problem**
  - A reachability graph is still **too large**.
    – its size can be equal to the size of the state space

– **What can we do with it?**
  - Let us relax the reachability analysis!

Reachability graph gives sufficient and necessary conditions for a reachability of a given state.

– a state is reachable if any only if it appears in the reachability graph

Planning graph will give only the necessary condition of reachability.

– if a state is reachable than it can be found in the planning graph
– however, not all the states in the planning graph are reachable

How to exploit a planning graph?

– If we can construct the planning graph fast with small memory consumption, it gives an estimate which actions are necessary to reach the goal state
– from the planning graph we still need to extract the solution plan

How to relax the reachability tree/graph?

– The nodes describe approximations of states at a given level of the reachability tree.

How to approximate the state?

Recall that a state is represented as a set of propositions.

– we can approximate all states at the same level by union of propositions from these states

– **OK, but how to do it in practice?**

- Apply all applicable actions in parallel to a given state and ignore negative effects of these actions.
- Consequence: the number of propositions in approximated states will never decrease
- Remember which actions provide a given proposition and which actions want to delete it

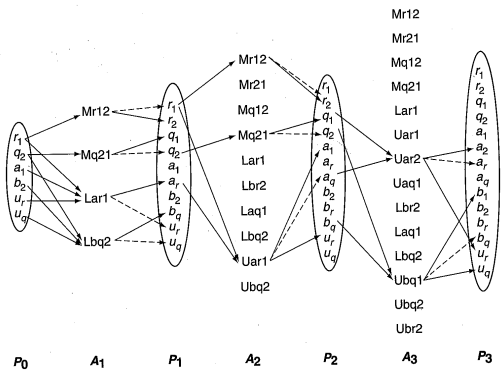
Planning graph is a directed layered graph, where

– each layer contains (exclusively)
- instantiated propositions (a state layer) or
- instantiated actions (an action layer)

– **state and action layers** interleave
- the zero layer describes the initial state
- the next layer describes all actions applicable to the initial state
- the next layer contains positive effects of these actions

– an action layer with the following state layer forms a **level**

– **arcs** are between
- propositions and actions that use them as preconditions
- actions and propositions that are effects of the action
  - special arcs for negative effects
  - negative effects are not deleted!



Where are the plans in the planning graph?

- Instead of a single sequence of actions we will use a sequence of sets of actions – **a layered plan**.
  - a set of actions at position j will be a subset of actions from j-th action layer
- The sequential plan is obtained from the layered plan by using any permutation of actions from each set.

Example:

A layered plan $\langle\{a1,a2\},\{a3,a4\},\{a5,a6,a7\}\rangle$ gives 2x2x6=24 sequential plans.

- **How to ensure that actions selected to a layered plan can be ordered without influencing the final states?**
  - Such actions must be independent!
- **When is a pair of actions (a,b) dependent?**
  - When different orders of the actions give different states, in particular:
    - **a** deletes some precondition of **b** (hence **a** cannot be right before **b**)
    - **a** deletes some positive effect of **b** (hence the result depends on the order)
    - symmetrically for **b**
- **A pair of actions (a,b) is independent** if and only if:
  - $\text{effects}^-(a) \cap (\text{precond}(b) \cup \text{effects}^+(b)) = \varnothing$
  - $\text{effects}^-(b) \cap (\text{precond}(a) \cup \text{effects}^+(a)) = \varnothing$

Note:

Independence of actions is given by the planning domain and it is not influenced by a particular planning problem!

Applying a set of actions

Let π **be a set of pairwise independent actions**, then

- π **is applicable to a state s** if and only if
  - preconditions of all actions from π are satisfied by s
  - $\text{precond}(\pi) = \cup \{\text{precond}(a) \mid \forall a \in \pi\} \subseteq s$
- **The result of applying π to state s** is a state
  - $\gamma(s,\pi) = (s - \text{effects}^-(\pi)) \cup \text{effects}^+(\pi)$
    - $\text{effects}^{-/+}(\pi)$ is a union of effects from π

Claim:

- If π consists of pairwise indepedent actions and it is applicable to state s, then for any permutation of actions $\langle a_1,a_2,\ldots,a_k\rangle$ from π we get $\gamma(s,\pi)= \gamma(\ldots\gamma(\gamma(s,a_1),a_2)\ldots a_k)$.

This claims makes using layered graphs practical!

A solution layered plan

- A layered plan $\Pi=\langle\pi_1,\pi_2,\ldots,\pi_k\rangle$ is a **solution plan** for problem (O,s_0,g) if and only if:
  - each set π_l consists of pairwise independent actions,
  - a set π_1 is applicable to state s_0, π_2 is applicable to state $\gamma(s_0,\pi_1)$ and so on,
  - $g \subseteq \gamma(\ldots\gamma(\gamma(s_0,\pi_1),\pi_2)\ldots\pi_k)$.

Claim:

If Π is solution plan for (O,s_0,g), then any sequence of actions consisting from a permutations of actions from π_1, followed by action permutation from π_2 and so on transfers the state s_0 to a state satisfying g.

Planning with a planning graph

How to do planning with the planning graph?

- **First build a planning graph** in such a way that the last propositional layer satisfies the goal condition.
  - More precisely, we will require that all the goal propositions can be used together in the last propositional layer.
- From the action layers **select subsets of independent actions** in such a way that they cover the goal propositions.
  - This is realised by a **backward run** from the last level, where actions giving the goal are selected and then, in the previous level, we select actions giving preconditions of actions selected from the last level etc.
  - Some goal proposition can be satisfied in the previous level (not the last one).

    We ensure that each proposition is an effect of some action from the previous layer.
    - Using a no-op action for each proposition: **α_p is a no-op action for p**, iff precond(α_p)=effect$^+$(α_p)={p}, effect$^-$(α_p)=$\emptyset$

Propositional mutex

How to find if two propositions can be together in the same layer?

- **All propositions** can be together at the **zero layer** (this is the initial state).
- **Two dependent actions** cannot be used together in the first action layer so their **positive effects cannot appear together** in the next state unless they are given by another pair of independent actions.
- **Two propositions** cannot be together if they are positive and negative effects of a single action (again, unless they are given by another pair of independent actions).
  - *No-op actions are treated as other actions in these conditions (if b deletes p, then α_p and b are dependent).*
- An incompatible pair of propositions is called a propositional **mutex** (**mutual exclusion**).

Action mutes

Level	*Mutex elements*
A_1	{Mr12} × {Lar1}
	{Mq21} × {Lbq2}
P_1	$\{r_2\} \times \{r_1, a_r\}$
	$\{q_1\} \times \{q_2, b_q\}$
	$\{a_r\} \times \{a_1, u_r\}$
	$\{b_q\} \times \{b_2, u_q\}$
A_2	{Mr12} × {Mr21,Lar1,Uar1}
	{Mr21} × {Lbr2,Lar1*,Uar1*}
	{Mq12} × {Mq21,Laq1,Lbq2*,Ubq2*}
	{Mq21} × {Lbq2,Ubq2}
	{Lar1} × {Uar1,Laq1,Lbr2}
	{Lbr2} × {Ubq2,Lbq2,Uar1,Mr12*}
	{Laq1} × {Uar1,Ubq2,Lbq2,Mq21*}
	{Lbq2} × {Ubq2}
P_2	$\{b_r\} \times \{r_1, b_2, u_r, b_q, a_r\}$
	$\{a_q\} \times \{q_2, a_1, u_q, b_q, a_r\}$
	$\{r_1\} \times \{r_2\}$
	$\{q_1\} \times \{q_2\}$
	$\{a_r\} \times \{a_1, u_r\}$
	$\{b_q\} \times \{b_2, u_q\}$
A_3	{Mr12} × {Mr21,Lar1,Uar1,Lbr2*,Uar2*}
	{Mr21} × {Lbr2,Uar2,Ubr2}
	{Mq12} × {Mq21,Laq1,Uaq1,Ubq1,Ubq2*}
	{Mq21} × {Lbq2,Ubq2,Laq1*,Ubq1*}
	{Lar1} × {Uar1,Uaq1,Laq1,Uar2,Ubr2,Lbr2,Mr21*}
	{Lbr2} × {Ubr2,Ubq2,Lbq2,Uar1,Uar2,Ubq1*}
	{Laq1} × {Uar1,Uaq1,Ubq1,Ubq2,Lbq2,Uar2*}
	{Lbq2} × {Ubr2,Ubq2,Uaq1,Ubq1,Mq12*}
	{Uaq1} × {Uar1,Uar2,Ubq1,Ubq2,Mq21}*
	{Ubr2} × {Uar1,Uar2,Ubq1,Ubq2,Mr12}*
	{Uar1} × {Uar2,Mr21*}
	{Ubq1} × {Ubq2}
P_3	$\{a_2\} \times \{a_r, a_1, r_1, a_q, b_r\}$
	$\{b_1\} \times \{b_q, b_2, q_2, a_q, b_r\}$
	$\{a_r\} \times \{u_r, a_1, a_q, b_r\}$
	$\{b_q\} \times \{u_q, b_2, a_q, b_r\}$
	$\{a_q\} \times \{a_1, u_q\}$
	$\{b_r\} \times \{b_2, u_r\}$
	$\{r_1\} \times \{r_2\}$
	$\{q_1\} \times \{q_2\}$

Propositional mutexes can give further incompatibilities between actions in addition to action dependence.

- **Two actions are mutex**, if some of their preconditions are mutex.
- An action whose preconditions are mutex can be removed immeditely.

Mutex – formal definitions

Two actions *a* and *b* **are mutex** at level A_i, if:

- *a* and *b* are dependent, or
- precondition of *a* has a mutex with some precondition of *b* at level P_{i-1}.

The set of action mutexes for level A_i is denoted μA_i.

Two propositions *p* and *q* **are mutex** at level P_i, if:

- each action A_i giving *p* as its positive effect has a mutex with any action giving *q* as a positive effect, and
- there is no action in A_i having both *p* and *q* and as positive effects.

The set of propositional mutexes for level P_i is denoted μP_i.

Mutex is a **symmetric relation.**

Sets of mutexes in the planning graph are **decreasing**:

- if p and q are from P_{i-1} and $(p,q) \notin \mu P_{i-1}$, then $(p,q) \notin \mu P_i$
- if a and b are from A_{i-1} and $(a,b) \notin \mu A_{i-1}$, then $(a,b) \notin \mu A_i$

Proof:

- If two propositions are not mutex then their no-op actions are not mutex and hence they give these propositions in the next layer.
- If two actions are not mutex then they are independent and their preconditions are not mutex. As the preconditions will not become mutex in the next layer (see above) the actions will not be mutex in the next layer.

Note:

Sets of actions and propositions in the planning graph are **increasing** ($P_{i-1} \subseteq P_i$ a $A_{i-1} \subseteq A_i$).

Graphplan is a planning system based on planning graph.

- It repeats graph expansion and plan extraction until a solution plan is found.
- **Expansion:**
  - First construct a planning graph till the layer where there are all goal propositions and no pair of them is mutex (this is a necessary condition for plan existence).
  - If plan extraction fails, add a new level (stop if some final condition holds, then no plan exists).
- **Extraction:**
  - Extract a layered plan from the planning graph in such a way that the plan gives all the goal propositions.

A technical restriction:

we only assume actions with positive preconditions (can be ensured by modifying the planning domain)

Graph expansion

- Planning graph is seen as a sequence of levels and mutex sets

$$G = \langle P_0, A_1, \mu A_1, P_1, \mu P_1, \ldots, A_i, \mu A_i, P_i, \mu P_i \rangle$$

- Planning graph depends only on the planning operators O and on the initial state s_0 (encoded in P_0), but it does not depend on the goal g!
- The procedure Expand(G) adds one level to the graph:

```
Expand(⟨P_0, A_1, μA_1, P_1, μP_1, ..., A_{i-1}, μA_{i-1}, P_{i-1}, μP_{i-1}⟩)
    A_i ← {a ∈ A | precond(a) ⊆ P_{i-1} and precond^2(a) ∩ μP_{i-1} = ∅}
    P_i ← {p | ∃a ∈ A_i : p ∈ effects^+(a)}
    μA_i ← {(a, b) ∈ A_i^2, a ≠ b | effects^-(a) ∩ [precond(b) ∪ effects^+(b)] ≠ ∅
              or effects^-(b) ∩ [precond(a) ∪ effects^+(a)] ≠ ∅
              or ∃(p, q) ∈ μP_{i-1} : p ∈ precond(a), q ∈ precond(b)}
    μP_i ← {(p, q) ∈ P_i^2, p ≠ q | ∀a, b ∈ A_i, a ≠ b :
              p ∈ effects^+(a), q ∈ effects^+(b) ⇒ (a, b) ∈ μA_i}
    for each a ∈ A_i do: link a with precondition arcs to precond(a) in P_{i-1}
        positive arcs to effects^+(a) and negative arcs to effects^-(a) in P_i
    return(⟨P_0, A_1, μA_1, ..., P_{i-1}, μP_{i-1}, A_i, μA_i, P_i, μP_i⟩)
end
```

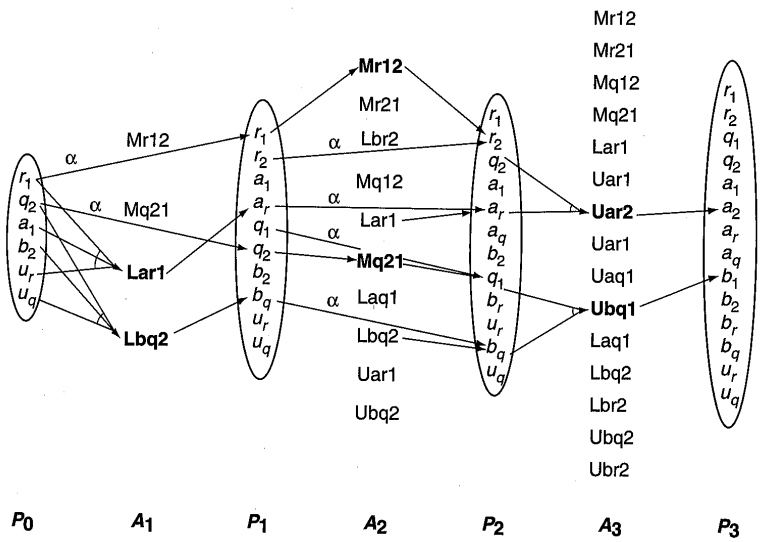
Fixed point

- The number of different levels in the planning graph is restricted. Starting with some level, the graph is not changing – a fixed-point level.
- **A fixed-point level** in the planning graph G is such a level κ that $\forall i$, $i>\kappa$ levels i are identical to it, i.e., $P_i=P_\kappa$, $\mu P_i=\mu P_\kappa$, $A_i=A_\kappa$, $\mu A_i=\mu A_\kappa$.
  - *Each planning graph G has a fixed-point level κ, where κ is the smallest number such that $|P_{k-1}|=|P_k|$ and $|\mu P_{k-1}|=|\mu P_k|$.*

    Proof is based on monotony and finiteness of the levels and mutex sets.
  - This claim also gives an efficient method how to detect the fixed-point level!

- **Plan extraction** is done in a **backward** direction from level P_i containing all the non-mutex goal propositions ($g^2 \cap \mu P_i = \varnothing$).
  - First, find a set of non-mutex actions $\pi_i \subseteq A_i$ that give all the goal propositions.
  - Preconditions of actions from π_i form a new goal for the previous level P_{i-1}.
  - If the goal for a level P_j cannot be covered by actions then go back to level j+1 and explore an alternative set π_{j+1}.
  - If level 0 is reached, the sequence $\langle \pi_1, \pi_2, \ldots, \pi_k \rangle$ is a solution.
- This is basically **AND/OR search**:
  - **OR branches** represent alternative actions giving the goal proposition
  - **AND branches** connect the preconditions with actions

Level	Mutex elements
A_1	{Mr12} × {Lar1}
	{Mq21} × {Lbq2}
P_1	$\{r_2\} \times \{r_1, a_r\}$
	$\{q_1\} \times \{q_2, b_q\}$
	$\{a_r\} \times \{a_1, u_r\}$
	$\{b_q\} \times \{b_2, u_q\}$
A_2	{Mr12} × {Mr21,Lar1,Uar1}
	{Mr21} × {Lbr2,Lar1*,Uar1*}
	{Mq12} × {Mq21,Laq1,Lbq2*,Ubq2*}
	{Mq21} × {Lbq2,Ubq2}
	{Lar1} × {Uar1,Laq1,Lbr2}
	{Lbr2} × {Ubq2,Lbq2,Uar1,Mr12*}
	{Laq1} × {Uar1,Ubq2,Lbq2,Mq21*}
	{Lbq2} × {Ubq2}
P_2	$\{b_r\} \times \{r_1, b_2, u_r, b_q, a_r\}$
	$\{a_q\} \times \{q_2, a_1, u_q, b_q, a_r\}$
	$\{r_1\} \times \{r_2\}$
	$\{q_1\} \times \{q_2\}$
	$\{a_r\} \times \{a_1, u_r\}$
	$\{b_q\} \times \{b_2, u_q\}$
A_3	{Mr12} × {Mr21,Lar1,Uar1,Lbr2*,Uar2*}
	{Mr21} × {Lbr2,Uar2,Ubr2}
	{Mq12} × {Mq21,Laq1,Uaq1,Ubq1,Ubq2*}
	{Mq21} × {Lbq2,Ubq2,Laq1*,Ubq1*}
	{Lar1} × {Uar1,Uaq1,Laq1,Uar2,Ubr2,Lbr2,Mr21*}
	{Lbr2} × {Ubr2,Ubq2,Lbq2,Uar1,Uar2,Ubq1*}
	{Laq1} × {Uar1,Uaq1,Ubq1,Ubq2,Lbq2,Uar2*}
	{Lbq2} × {Ubr2,Ubq2,Uaq1,Ubq1,Mq12*}
	{Uaq1} × {Uar1,Uar2,Ubq1,Ubq2,Mq21}*
	{Ubr2} × {Uar1,Uar2,Ubq1,Ubq2,Mr12}*
	{Uar1} × {Uar2,Mr21*}
	{Ubq1} × {Ubq2}
P_3	$\{a_2\} \times \{a_r, a_1, r_1, a_q, b_r\}$
	$\{b_1\} \times \{b_q, b_2, q_2, a_q, b_r\}$
	$\{a_r\} \times \{u_r, a_1, a_q, b_r\}$
	$\{b_q\} \times \{u_q, b_2, a_q, b_r\}$
	$\{a_q\} \times \{a_1, u_q\}$
	$\{b_r\} \times \{b_2, u_r\}$
	$\{r_1\} \times \{r_2\}$
	$\{q_1\} \times \{q_2\}$



- let $\{a_2, b_1\}$ be the goal atoms
- **bold actions** are selected to the layered **plan**
- We get the layered plan
  $\langle${Lar1,Lbq2}, {Mr12,Mq21}, {Uar2,Ubq1}$\rangle$

Nogoods

- **Mutex** can capture **incompatible pairs**.
- We may find that a given goal cannot be satisfied at a given level. Then the set of propositions forming the goal is together **incompatible**.
  - If we explore the same level later with the same (or larger) goal, then we already know that such a goal cannot be satisfied and the algorithm can immediately backtrack.
- Unsatisfied goals can be remembered for each each level in the form of a **nogood** table ∇.
  - A goal that is nogood immediately fails and the algorithm can backtrack.

Extraction algorithm

```
Extract(G, g, i)
    if i = 0 then return (⟨⟩)
    if g ∈ ∇(i) then return(failure)
    π_i ← GP-Search(G, g, ∅, i)
    if π_i ≠ failure then return(π_i)
    ∇(i) ← ∇(i) ∪ {g}
    return(failure)
end
```

Extract tries to cover a goal at level i

- Uses and updates the nogood table ∇

GPSearch looks for a set π_i of actions covering the goal.

- incrementally adds actions to π_i such that the actions are non-mutex
- after covering the goal continues to the previous level

```
GP-Search(G, g, π_i, i)
    if g = ∅ then do
        Π ← Extract(G, ⋃{precond(a) | ∀a ∈ π_i}, i − 1)
        if Π = failure then return(failure)
        return(Π. ⟨π_i⟩)
    else do
        select any p ∈ g
        resolvers ← {a ∈ A_i | p ∈ effects+(a) and ∀b ∈ π_i : (a, b) ∉ μA_i}
        if resolvers = ∅ then return(failure)
        nondeterministically choose a ∈ resolvers
        return(GP-Search(G, g − effects+(a), π_i ∪ {a}, i))
end
```

Graphplan algorithm

```
Graphplan(A, s0, g)
    i ← 0,    ∇ ← ∅ ,    P0 ← s0
    G ← ⟨P0⟩
    until [g ⊆ Pi and g² ∩ μPi = ∅] or Fixedpoint(G) do
        i ← i + 1
        G ← Expand(G)
    if g ⊈ Pi or g² ∩ μPi ≠ ∅ then return(failure)
    Π ← Extract(G, g, i)
    if Fixedpoint(G) then η ← |∇(κ)|
    else η ← 0
    while Π = failure do
        i ← i + 1
        G ← Expand(G)
        Π ← Extract(G, g, i)
        if Π = failure and Fixedpoint(G) then
            if η = |∇(κ)| then return(failure)
            η ← |∇(κ)|
    return(Π)
end
```

- first find a planning graph satisfying goal condition g
- if it does not exist then stop (no solution)
- otherwise extract a layered plan
- if not successful
- add one more layer
- and again extract a layered plan
- stop when plan is found or no plan exists.

Graphplan - properties

Graphplan is sound, complete, and always finishes.

Proof:

- if the algorithm returns a plan, then it is a solution plan
- if the algorithm fails, then no plan exists
  - Fixedpoint(G) and ($g \not\subset P_i$ or $g^2 \cap \mu P_i \neq \emptyset$)
    - If there is no level satisfying the goal before reaching the fixed point then no other level can cover the goal
  - $|\nabla_{i-1}(\kappa)| = |\nabla_i(\kappa)|$
    - nogood tables only increase – so we get $\nabla_{i-1}(\kappa) = \nabla_i(\kappa)$
    - If the nogood table in the fixed point does not change then the goals from the fixed point cannot be ever satisfied – the nogood table just propagates to next levels in next steps $\nabla_{i-1}(\kappa) = \nabla_i(\kappa+1)$
- The algorithm always stops thanks to monotony and a finite number of atoms and actions (some layers start to repeat and the nogood tables become full).

- **Memory management**
  - The planning graph works with fully instantiated propositions and actions which is memory consuming.
  - Thanks to monotony we do not need to store levels and mutex sets separately – it is enough for each atom/action to remember the first level when it appeared and similarly for mutex pairs.
- **Search**
  - first try to cover propositions with a smaller number of supporting actions or that appeared later in the graph
  - for action selection, prefer no-ops or actions that appeared earlier in the graph
  - after selecting an action, verify that all remaining goal propositions still have some support actions (forward checking))

The idea of planning graph brought a small revolution to planning as it significantly **increased efficiency** of planning techniques and allowed solving **much larger problems**.

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz