

The planning problem

Lecture 8 Planning

Up to now we focus on **entailment**: given a set of sentences which we believes to be true, what other sentences we should also believe in.

We turn to a more difficult problem: **change the world** with a sequence of **actions** such that a **desired sentence becomes true**.

Reference:

- Chapter 11 and 12.3–12.6.

Planning introduces **states**, a KB where some predicates are true.

- In the **initial state**, some **predicates** are known to be true, and some are known to be false.
- There are some **action schemas**, which can be **instantiate to actions**. Actions can be **applied** to a state, provided that some **preconditions** are satisfied in the state. This results into a **new state**, with some predicates **changed** from true to false or vice versa.
 - Instantiation means replacing variables in the schema with actual values.
- Goal: Find a **sequence of actions**, so that after applying them, **some predicates hold** in the resulting state.

We will focus on the situation that there is no **inference** to perform. I.e., everything that change truth value is directly indicated by the action.

It can be done with some tedious work, which we want to avoid during lectures.

AI(0270)

AI(0270)-8.1

A (really simple) planning problem instance

- **Initial state:** $At(P_1, SFO), Plane(P_1), Airport(SFO), Airport(JFK), Airport(HKG)$.
The plane P_1 is at the airport SFO , while there are other airports JFK and HKG .
- **Goal:** $At(P_1, JFK)$.
The plane comes to JFK .
- **Action Fly:** $Fly(p, from, to)$:
 - **Precondition:** $Plane(p), Airport(from), Airport(to), At(p, from)$.
 - **Effect:** $At(p, from)$ becomes false, $At(p, to)$ becomes true.

The shortest solution is just one step: $Fly(P_1, SFO, JFK)$.

Typically, there are many predicates in the preconditions, although there might only be a few in the goal. The number of objects involved is likely to be large, although there might just be a few action schemas.

AI(0270)-8.2

AI(0270)-8.3

STRIPS representation

- **State** representation. It is a conjunction of predicates. The objects in the predicates must be constants.
E.g., $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO)$. Predicates not in the list is assumed false: "**close-world assumption**".
As usual, 0-arity predicates are written like $HaveEgg$ rather than $HaveEgg()$.
- **Goal** representation. It is a conjunction of positive predicates, which should be a partially specified state.
- **Action schema:** with a name and argument list, like $Fly(p, from, to)$. Arguments can appear in the preconditions and effects.
 1. **Precondition:** a conjunction of positive predicates that must be true before the action is applicable.
 2. **Effect:** a conjunction of predicates. If positive, the action add it to the state; if negative, the action delete it from the state.

ADL

Action Description Language (ADL) is a more "convenient" language than STRIPS to specify planning problems. Most important differences:

1. **Open-world assumption**, i.e., things not in state is assumed to be unknown rather than false. So state also contains negated predicates.
In STRIPS, we can emulate $\neg At(P_1, SFO)$ by $KnowNotAt(P_1, SFO)$.
2. **Effects** can be conditional: some of the predicates in the effect list may be conditioned on another predicate (e.g., Action1: Effect $(A \Rightarrow B), C$).
In STRIPS this can be emulate by turning an action to many, e.g., Action 1a: Precond $KnowA$, Effect B, C ; Action 1b: Precond $KnowNotA$, Effect C .
3. **Goals** can have **variables**, which are assumed to be existentially quantified—it is a query for what value of variable works. It can also have **disjunctions**.

In STRIPS, variable is impossible. But we can emulate disjunctive goals by using an extra "disjunction predicate", and use the actions to manage its value.

So most can be emulated by STRIPS, but it makes actions more complicated.

AI(0270)-8.4

AI(0270)-8.5

More example: Spare tire problem

- **Initial state:** a tire of a car is flat, and there's a spare one in the trunk.
 $At(Flat, Axle) \wedge At(Spare, Trunk)$
- **Goal:** having the spare tire installed: $At(Spare, Axle)$.
- **Action 1:** $Remove(Spare, Trunk)$: precond $At(Spare, Trunk)$, effect $\neg At(Spare, Trunk) \wedge At(Spare, Ground)$.
- **Action 2:** $Remove(Flat, Axle)$: precond $At(Flat, Axle)$, effect $\neg At(Flat, Axle) \wedge At(Flat, Ground)$ (We can merge the two $Remove$ actions).
- **Action 3:** $PutOn(Spare, Axle)$: precond $At(Spare, Ground) \wedge \neg At(Flat, Axle)$, effect $\neg At(Spare, Ground) \wedge At(Spare, Axle)$.
The use of \neg in the precond requires ADL. We can turn this to (more natural) STRIPS by using $Clear(Axle)$ instead.
- **Action 4:** $LeaveOvernight$: precond none, effect $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \dots \wedge \neg At(Flat, Trunk)$

Forward state-space search: Progression planning

- Apart from the introduction of terms like **predicates** and **instantiations**, the problem is a **standard searching problem**.
- As long as we don't allow functions in predicates, there is a finite number of objects, and thus a **finite number of actions**. The successor function is thus finite, and can be found by **unification**.
- The direct searching approach is called **progressive planning**.
- We want our algorithm to be able to handle many objects. But for normal searching algorithm, this would create a **lot of actions** from the schema and thus a **large branching factor**.
This makes the problem P-SPACE complete.
- We expect planning problem to be solved **better** than blind searching. Why? Because **the state is open** to the algorithm, we can find **very good heuristics or algorithms**.
So Planning is to Searching as CSP is to Backtracking.

AI(0270)-8.6

Backward state-space search: Regression planning

- If there are only a few ways to establish them, then we can reduce branching factor by **regression**, i.e., search backwards.
This is possible because we have an open state representation...
- The goal is a **partially specified state**, consisting of predicates. So instead of modifying the *truth of each predicate* in every searching step, we modify **what predicates we want to become satisfied**.
 - So the search begin with the **goal** description, like $At(Spare, Axle)$.
 - In each searching step, we find what **action** can be the **last step** to arrive at that goal, and derive **what predicates are needed** so that after applying that action, the goal is satisfied.
 - We then get a **subgoal** that we can apply the strategy again.

AI(0270)-8.7

Regressing through an action

So given a **goal** and an **action**, we want to know what is required **right before** executing action, so that after execution the goal become satisfied.

We call this **regressing the goal** through the action. Procedure:

- **Remove** those predicates that the action **achieves**.
- **Add** those that the action **requires**.

E.g., regress the goal $At(P_1, JFK)$ through the action $Fly(P_1, SFO, JFK)$...

- The action $Fly(P_1, SFO, JFK)$ achieves $At(P_1, JFK)$, so we can **remove** that from the goal.
- The action has a precondition $Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge At(P_1, SFO)$, so all of them must be **added** to the set.

So the result is $Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge At(P_1, SFO)$.

AI(0270)-8.8

Consistence and Relevance

The last step must be **consistent**, i.e., not **remove** a needed predicate.

- E.g., if what we need is $At(Spare, Ground) \wedge \neg At(Flat, Axle)$...
- Then *LeaveOvernight* is not an allowed last action even though it achieves $\neg At(Flat, Axle)$ —because it removes $At(Spare, Ground)$.

Also, the last step must be **relevant**, i.e., **achieve something that is needed**. E.g.,

- The action $Fly(P_2, SFO, JFK)$ produce nothing needed for $At(P_1, JFK)$. We say $Fly(P_2, SFO, JFK)$ is not **relevant**.
- Given a plan with an irrelevant last action, we can **delete** it to get a shorter plan. So we don't need to cater for cases where it is the last step.

Question: can we restrict "relevent" actions to those with an effect not already satisfied in the initial state?

AI(0270)-8.9

In search language...

So we can formulate **regressive planning** as a **state space search**:

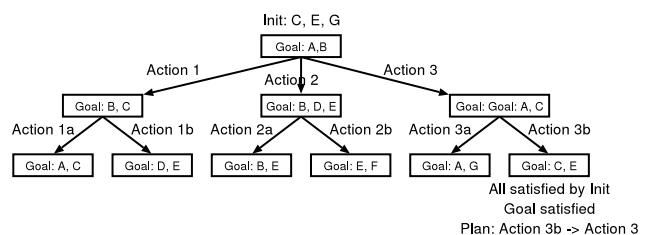
- **State:** each state is a set of predicates that needs to become true.
- **Initial state:** the goal of the planning problem.
- **Goal state:** a state in which every predicate is known to hold in the initial state of the planning problem.
- **Successor function:** given a state, find all **relevant and consistent** action. For each of such action A, **regress** the state through A, which results into another state (set of predicates).

At the end, we will find an action sequence that brings us from the goal to a set of predicates satisfied by the initial states, and the **reverse** of the action sequence would be a solution plan.

AI(0270)-8.10

Illustration: Regressive planner

So the search tree as seen by a regressive planner looks like this:



Of course, one can use any algorithm like BDS, IDS, etc., to search in this state-space.

AI(0270)-8.11

Heuristics in State-space search planners

- **BFS** or **IDS** performs poorly in both progressive and regressive planners, since the branching factor is too high. **BDS** can make this a bit better.
- How about **informed** search algorithms? Can we use A^* search, in particular? To do so we need a **heuristic function** that is **admissible**.
- One simple way: **count** the number of **unsatisfied predicate** (for progressive planners, compare with planning goal state; for regressive planners, compare with the initial state.)
- But it is **not really admissible**: some action may create **multiple effects**, and in this case the heuristic over-estimates.
- We can thus improve the search time, although the resulting plan is sometimes **slightly suboptimal**.
But this is not very useful for progressive planning, since typically we always have one (or just a few) unsatisfied predicate.

AI(0270)-8.12

Better heuristics

Admissible heuristics usually comes from **relaxing** the problem.

- E.g., **remove** (1) all **preconditions** in actions, and (2) all **negated** predicates in **effects**. It is clearly a big relaxation. E.g.,

$$\begin{aligned} \text{Goal}(A \wedge B \wedge C) & \quad \text{Action}(X, \text{EFFECT} : A \wedge P) \\ & \quad \text{Action}(Y, \text{EFFECT} : B \wedge C \wedge Q) \\ & \quad \text{Action}(Z, \text{EFFECT} : A \wedge P \wedge Q) \end{aligned}$$

- We need at least 2 steps to arrive at the goal: X and Y .
- The problem becomes to find a **minimal set of actions** that “covers” all the required goals. It is called the **set cover** problem.
- Set cover is NP-hard, and thus cannot be solved in polynomial time. But **approximation algorithm** exists which usually does reasonably well.
The algorithm: always choose the set covering the most uncovered predicates. It is never $\log_2 n$ times worse than the best solution, usually much better.

AI(0270)-8.13

Empty-delete-list

The heuristic in the last slide is very relaxed and thus **not accurate**. It is possible to **relax less**.

- The **Empty-delete-list** heuristic: **just remove every negated predicate in effects**, without touching the precondition.
- Thus each state still has number of satisfied predicates **increases monotonically**, but we can't apply any actions at will.
- This is **more difficult to compute** than set cover. We need to run a simple planning algorithm to calculate the heuristic function, so it is quite costly.
- But this is the **best state-space search planner** so far: **progressive** planners with **empty-delete-list** as heuristic.

AI(0270)-8.14

Fully-Ordered vs. Partial Order Planners

- So far all the planner we see are **fully-ordered**: the ordering of actions of a plan is exactly specified by the resulting plan.
- This is reasonable for those planning problem that each step is related with another. In such problems, we need the ordering to **make sure the plan can be executed**.
- For most real-world problem, a complete plan consists of **sub-plans** that are independent.
- In such cases we prefer **leaving ordering unspecified**, because:
 - It is **faster**: we don't have to examine every ordering when searching when it does not make a difference.
 - The output is more **generic**: if, when we execute the plan, we find that something prevent some of the ordering, we might be able to use another ordering.

AI(0270)-8.15

Example problem

Consider another very simple planning problem in STRIPS:

Goal(*RightShoeOn* \wedge *LeftShoeOn*)
 Init()
 Action(*RightShoe*, PRECOND : *RightSockOn*, EFFECT : *RightShoeOn*)
 Action(*RightSock*, EFFECT : *RightSockOn*)
 Action(*LeftShoe*, PRECOND : *LeftSockOn*, EFFECT : *LeftShoeOn*)
 Action(*LeftSock*, EFFECT : *LeftSockOn*)

This involve two more or less **independent** tasks: deal with the left sock and shoe, and deal with the right sock and shoe.

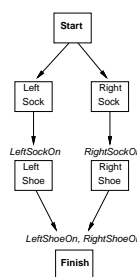
If only asked for a plan for *RightShoeOn*, any planner we have seen will give [*RightSock*, *RightShoe*] as the shortest plan.

But what if we ask for both *RightShoeOn* and *LeftShoeOn*? There are suddenly 6 plans...

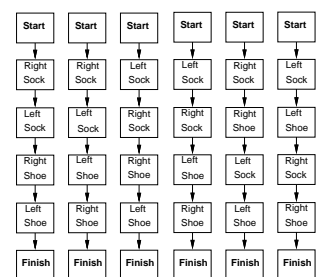
AI(0270)-8.16

Partial Order Plans

Partial Order Plan:



Total Order Plans:



In one partial-ordered plan in the left, we say that **all** fully order on the right are solutions. This is of course desirable.

AI(0270)-8.17

How to execute the plan?

To the agent who want to execute the plan, the most important information provided by a plan consists of two parts:

1. The set of actions required: we need to wear the left sock, wear the right sock, wear the left shoe and wear the right shoe.
2. The set of **ordering constraints**: we must wear the left sock before wearing the left shoe, we must wear the right sock before wearing the right shoe. We write $RightSock \prec RightShoe$

To execute the plan, we perform **linearization**: At each step, we find one of the actions that has **no arrow pointer to it** and execute it.

This is repeated until all the actions required are performed. Note that this leave some freedom for the agent to choose an ordering when executing the plan.

For a correct plan, **any linearization makes the goal satisfied**.

AI(0270)-8.18

How to know a plan is correct?

To verify the correctness of a plan, we need **additional information**: why each step is executed, or why every precondition should be expected to be true. They are shown in the picture a few slides before:

- For each precondition C of each action A' , an arrow, or **causal link** is drawn from an action A , showing what causes the condition to become true. We write $A \xrightarrow{C} A'$.

E.g., a causal link is added between the *LeftShoe* action and the *LeftShoeOn* precondition of the *Finish* action.

To make things more regular, we introduce the *Start* action to be always the first action, and the *Finish* action to be always the last.

If $A \xrightarrow{C} A'$, then $A \prec A'$. To simplify the diagram we won't draw the ordering constraints when there is a causal link.

AI(0270)-8.19

Partial order plans

- Now we can define what is a (partial order) plan.
- A plan consists of three sets: sets of **actions, ordering constrains and causal links**.
- E.g., The plan that we saw before is the following:

Actions: $\{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish\}$

Ordering: $\{RightSock \prec RightShoe, LeftSock \prec LeftShoe\}$

Links: $RightSock \xrightarrow{RightSockOn} RightShoe, LeftSock \xrightarrow{LeftSockOn} LeftShoe,$
 $RightShoe \xrightarrow{RightShoeOn} Finish, LeftShoe \xrightarrow{LeftShoeOn} Finish$

There should be ordering constraints from *Start* to every action and from every action to *Finish*, but since they are always there we won't spell them out.

- A **partial order planner** (POP) will try to come up with such a plan, given a problem specification.

AI(0270)-8.20

Conflicts

There is one important use of the causal link: to handle the complications arising from **one step removing the effect of another**.

- Suppose $A \xrightarrow{C} A'$. If another action B in the plan would remove C , then it must not be done between A and A' .
- Otherwise, when A' is executed, it will find that C is no longer true, and the plan will fail. We say **B conflicts with** the casual link $A \xrightarrow{C} A'$.
- To prevent this, i.e., **to resolve the conflict**, we require that the plan must have either $B \prec A$ or $A' \prec B$. This guarantees that B will be done outside the $[A, A']$.
- Should we add $B \prec A$ or $A \prec B$? We don't know, so we try both.
- If adding an ordering constraint leads to a **cycle**, the resulting plan becomes **invalid**, so we can stop looking for solution in that route.

AI(0270)-8.21

Requirements for correct partial-order plans

The **criteria** that a partial order plan is correct:

- The plan has some **actions**, including the special actions *Start* which has all initial conditions as effects, and *Finish* which has all goal as preconditions.
- The plan has some **ordering constraints**, so that $Start \prec A$ and $A \prec Finish$ for any action A . There must be no cycles formed by the ordering constraints.
- All **preconditions** of all actions are supported by a **causal link**.
- Each **causal link** has the corresponding **ordering constraints**.
- All **conflicts** are resolved by **ordering constraints**.

AI(0270)-8.22

State-space search in plan-space

To find a partial-order plan, we **search** in the **plan space**.

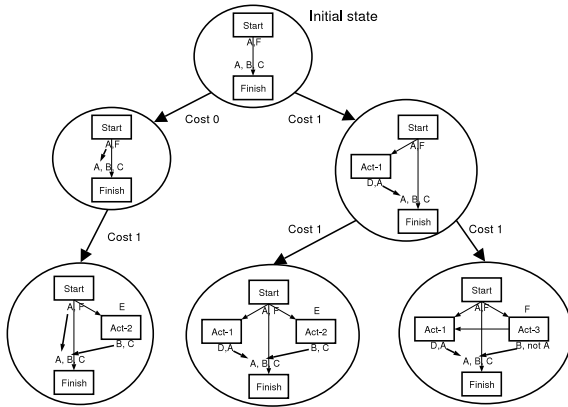
- **States**: a **plan** which is **incorrect**, because **some preconditions** are not supported by casual links, or "**open**".

All other requirements of a correct plan must hold: constraints must be resolved, *Start* and *Finish* must be there, etc. We say it is **consistent**.

- **Goal state**: a correct partial-order plan.
I.e., a consistent plan with no open precondition.
- **Initial state**: the "trivial" consistent plan, with **only** the *Start* and *Finish* symbol and an ordering constraint between them.
- **Successor function**: pick an **open precondition**, add **causal link** to support it (perhaps adding **actions**), and, **resolve all conflicts**.
- **Cost**: if action is added, cost of the step is the action cost.

AI(0270)-8.23

Illustration: the plan space



AI(0270)-8.24

Example: Spare tire problem

We show the problem again for easy reference...

- Init: $At(Flat, Axle) \wedge At(Spare, Trunk)$
- Goal: having the spare tire installed: $At(Spare, Axle)$.
- Action 1: $Remove(Spare, Trunk)$: precondition $At(Spare, Trunk)$, effect $\neg At(Spare, Trunk) \wedge At(Spare, Ground)$.
- Action 2: $Remove(Flat, Axle)$: precondition $At(Flat, Axle)$, effect $\neg At(Flat, Axle) \wedge At(Flat, Ground)$ (We can merge the two $Remove$ actions).
- Action 3: $PutOn(Spare, Axle)$: precondition $At(Spare, Ground) \wedge \neg At(Flat, Axle)$, effect $\neg At(Spare, Ground) \wedge At(Spare, Axle)$.
The use of \neg in the precondition requires ADL. We can turn this to (more natural) STRIPS by using $Clear(Axle)$ instead.
- Action 4: $LeaveOvernight$: precondition none, effect $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \dots \wedge \neg At(Flat, Trunk)$

AI(0270)-8.25

Example POP Planning

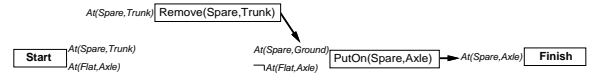
Consider the spare tire problem. Suppose we are doing IDS, have tried all plans with 2 actions, and find them all failed. Now we allow for 3 actions.

- At the beginning, we only have the 2-action trivial plan. The only open condition is $At(Spare, Axle)$.
- There is only 1 way to support $At(Spare, Axle)$: to add an action $PutOn(Spare, Axle)$.
- Thus there is only 1 successor: the plan with 3 actions $Start, Finish$ and $PutOn(Spare, Axle)$, with causal link from $PutOn(Spare, Axle)$ to $At(Spare, Axle)$ of $Finish$.
- Now we have 2 preconditions: $At(Spare, Ground)$ and $\neg At(Flat, Axle)$. We pick one arbitrarily, say we pick the first.
- Again there is only 1 way to support $At(Spare, Ground)$: add $Remove(Spare, Trunk)$. So we have 1 successor.

AI(0270)-8.26

Example POP Planning (2)

We now have this plan...

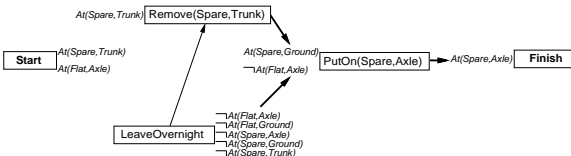


- We now want to pick yet another open precondition to support. There are two, and again we can pick any. Suppose we pick $\neg At(Flat, Axle)$.
- There are two ways to support $\neg At(Flat, Axle)$: either we add $LeaveOvernight$, or we add $Remove(Flat, Axle)$. They give different successors, and must be tried one by one (through backtracking, of course).
- Suppose we try $LeaveOvernight$ first. This causes, among other things, $\neg At(Spare, Ground)$, so it conflict with the causal link provided by $Remove(Spare, Trunk)$...

AI(0270)-8.27

Example POP Planning (3)

There is only one way to resolve the conflict without introducing a cycle: add an ordering constraint $LeaveOvernight \prec Remove(Spare, Trunk)$:

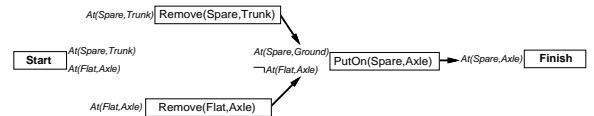


- Now there is only one open precondition: $At(Spare, Trunk)$. We have also used up our cost limit. Now more action can be added now.
- The $Start$ action may provide a casual link for the open condition. But $LeaveOvernight$ conflict with it...
- And both ways to resolve the conflict results in cycles!

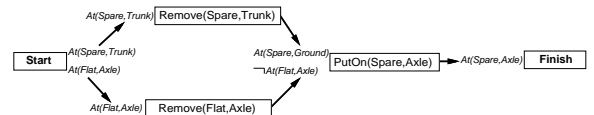
AI(0270)-8.28

Example POP Planning (4)

Now we have to **backtrack**. There is indeed only one "continuation": to support $\neg At(Flat, Axle)$ using $Remove(Flat, Axle)$.



Again we used up our cost quota, but the remaining open preconditions can be supported without additional actions. So we arrive at a solution.



AI(0270)-8.29

Variables in subgoals

- During the planning process, it is possible that we have actions that can be **partially** specified.
- E.g., if we want $\neg At(P_1, SFO)$, we need $Fly(P_1, SFO, x)$ for any x which is an airport, but for what x ?
- It is possible to **instantiate the action completely** whenever we add an action, by instantiating the action immediately.
This is the solution that is normally used on a regression planner.
- If there are many airport, this will **add a lot to the branching factor**.
- Another possibility: **leave it unspecified**.
Just like we want to leave plan order unspecified. In general, we want to avoid committing to a choice whenever we can—minimum commitment principle.
- In this case our sub-goal will have a variable in it.

AI(0270)-8.30

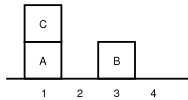
Some complications

- When the sub-goal can have variables waiting to be binded, **actions and causal links may also have variables waiting to be binded**.
- There is one complication: $Fly(P_1, SFO, SFO)$ won't work!
In particular, it will not establish $\neg At(P_1, SFO)$.
- Another problem: when causal links and effects have variables, we **don't know whether a causal link is causing a conflict!**
E.g., The action $Fly(P_1, SFO, x)$ conflicts with a causal link with condition $\neg At(P_1, JFK)$ only if $x = JFK$.
- One solution: check it when x is binded. But this makes the successor function complicated.
- Another solution: **allow (in)equality in preconditions** as well. E.g., the precondition might say $At(P_1, x) \wedge x \neq SFO \wedge x \neq JFK$.
This allows potential conflicts to be resolved immediately.

AI(0270)-8.31

Class exercise: the block world

The most interesting toy problem in planning is the block world: it easily give a huge branching factor even with a simple problem with very few action schema.



- **Start state:**
- **Goal:** $On(A, B) \wedge On(B, C)$
- **Action schema:** only one— $Move(block, here, there)$:
Pre-condition: $On(block, here) \wedge Clear(block) \wedge Clear(there)$
Note: this is slightly incorrect: what if $here = there$?
Effect: $On(block, there) \wedge Clear(there) \wedge \neg Clear(there)$

AI(0270)-8.32

Heuristic for POP planning

- Avoiding to specify an ordering can sometimes lead to a substantial saving of time, but that won't remove the need for a good **heuristic**.
- But heuristic in POP is **more difficult to find**, since we don't have an explicit state.
- Simple possibilities: **count number of open preconditions** that is not satisfied by an action that its action doesn't precede.
- Like the counting strategies for total-ordered planners, it is not admissible, but can reduce the search space significantly.
- But we have one more interesting choice to make...
- At each step, we need to **pick an arbitrary open-precondition**. Which to pick? We can do something similar to MRV in CSPs.
I.e., choose the open precondition that has fewest ways to support.

AI(0270)-8.33

Heuristic capturing negative interactions

- So far our heuristics does not capture **negative interaction**. I.e., we ignore the fact that the effect of one action may **undo** the precondition achieved by another.
- It is **too time consuming** to consider **all** negative interactions.
The problem then becomes as difficult as the original planning problem.
- But capturing negative interactions **partially** can still improve our heuristic to much better than even empty-delete-list.
- The techniques work with **propositions**. FOL sentences must be propositionalized—whether it is worthwhile depends on problems.
- What we will do: do some **preprocessing** of an initial state so that more accurate heuristic can be derived.
Benefits regressive planners much more than progressive planners: for regressive planner we have only one initial state (and only one pre-processing to do).

AI(0270)-8.34

Mutex: a simple way to capture some negative interactions

The idea: keep track of **what is possible before** each step i :

- What **propositions** (positive or negated) can be true.
 - What **pair** of propositions (positive or negated) can be true **together**.
How about which 3 or more can be true together? It is too slow to compute.
- E.g., if A can be true in step 1, and the only action applicable is to establish B and C but removes A , then at or before step 2,
- B and C can be true, and can be true together.
 - A can be true (in particular, at step 1, A is true).
 - But A and B cannot be true together, since the action establishing the previously impossible B would delete A . We say they are **mutex** (mutually exclusive). Similar for A and C .
 - An action requiring $A \wedge B$ thus can't be executed yet.

AI(0270)-8.35

The preprocessing to do

- At the **initial state** (step 0), some propositions (perhaps negated) are true. All pairs occur together (i.e., no mutex).
- An action requiring some pre-conditions cannot be executed if either...
 - A precondition cannot happen before a step.
 - A pair of preconditions cannot happen at the same time.
- Each action allows some further **propositions**, and also some further **pairs of propositions** to happen together.
- So given what is possible up to step i , we can derive what is possible up to step $i + 1$.
- This is repeated until **no more proposition or pairs of propositions can be added**.
This always happens, as there are finitely many propositions.

AI(0270)-8.36

Example

We use an extremely simple example to illustrate the idea...
Bigger example causes the planning graph to be exceedingly complex.

```

Init(HaveCake ∧ NotEatenCake)
Goal(HaveCake ∧ EatenCake)
Action(EatCake,
  PRECOND: HaveCake
  EFFECT: ¬HaveCake ∧ NotHaveCake
        ∧ EatenCake ∧ ¬NotEatenCake)
Action(BakeCake,
  PRECOND: NotHaveCake
  EFFECT: HaveCake ∧ ¬NotHaveCake)
    
```

This is written in STRIPS, so close-world assumption holds.

AI(0270)-8.37

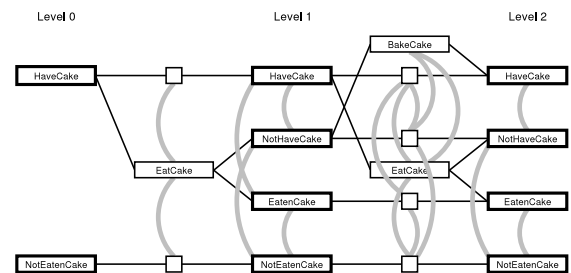
What we would know

- Up to step 0**, *HaveCake* and *NotEatenCake* can be true, and they can be true together.
- This allows the *EatCake* action to be taken, but not *BakeCake*.
- So this adds *NotHaveCake* and *EatenCake* to the possible true proposition **up to step 1**.
- But only the pairs (*HaveCake*, *NotEatenCake*) and (*NotHaveCake*, *EatenCake*) can occur together. The other 4 pairs cannot (so we have 4 mutexes).
- This allows both *EatCake* and *BakeCake* to execute at a point up to step 1.
- They don't add new propositions **up to step 2**, but now the pair (*HaveCake*, *EatenCake*) can happen together as well, so **one mutex is removed**.

AI(0270)-8.38

Planning graph

We usually represent these information with a "planning graph":



The proposition that can be true up to a step is put into a "level", and the possible actions take us from one level to the next.

AI(0270)-8.39

Interpreting the planning graph

- The thick boxes: the **propositions** that can be true up to a step.
- The thin boxes: the **actions** that can be taken.
- The small boxes: **persistent actions**, i.e., "actions" that allows propositions of the last step to be carried forward to the next step.
By definition all proposition has their persistent actions.
- The **lines** with propositions on left and actions on right: the **preconditions** that allow the action to be taken.
- The **lines** with actions on left and propositions on right: the **effects** that is produced by the action.
Negated effects are captured by the action mutexes.
- The **grey arcs**: the **mutex** that two actions or two proposition cannot happen at the same time.

AI(0270)-8.40

Creating the planning graph, part 1

After the initial step (which just contains all propositions known to be true and no mutex), we must construct subsequent steps.

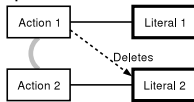
Adding actions from step i :

- Persistent actions are added for each proposition.
- Each action is examined to see whether
 - All its preconditions are in step i .
 - None of its preconditions have a mutex in step i .
- All **non-persistent** actions are **mutex** to each other.
Unless we are building parallel plan that in a step you can take many actions. Our graph is called a "serial planning graph" as we disallow them.
- An action is also mutex to another action if they have **inconsistent effect** or **competing needs**...

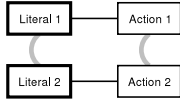
AI(0270)-8.41

Inconsistent effect and competing needs

- Two actions have **inconsistent effects** if one of the actions **remove** a proposition in the next step and the other **establishes** it.



- Two actions have **competing needs** if one of their preconditions are **mutually exclusive**.



We can expect that one of the actions (say, action 2) is a persistent action, since non-persistent actions have mutexes anyway.

AI(0270)-8.42

Creating the planning graph, part 2

Once the possible actions in step i (and their mutexes) are known, we can compute the possible propositions in step $i + 1$.

- There is a literal in step $i + 1$ if and only if an **action** in step i have that literal as a **effect**.
- Two literals are negation of each other if...
 - Inconsistent support:** All actions that produces them are mutex to each other.
 - They are negation of each other.
 - This case shouldn't be needed actually, as the actions should negate that create L should also delete $NotL$.
- If two literals have no mutex in step i , then both have persistent actions that make them in step $i + 1$, and the persistent actions have no mutex. So the two literals will still be there without mutex in step $i + 1$.
 - So mutexes is really monotonically being removed.

AI(0270)-8.43

Using a planning graph in regressive planners and POP

- In both the regressive planner and POP, we have a **single initial state** for the search.
- So we can process the initial state at the beginning to make a planning graph at the beginning, and use it to give heuristic values...
 - If a subgoal has a literal that is found only at step i onwards, then the heuristic value of the subgoal would be at least i .
 - If a subgoal has two literals that is both found and not mutex only at step i onwards, then the heuristic value of the subgoal is at least i .
 - The estimate is admissible: there is really no way to establish that subgoal before doing i actions.
- For POP, we might find the first level in which all open preconditions are satisfied, trace the actions leading to it, and count how many of them are not already in the plan. This is slower, though.

AI(0270)-8.44

Extracting solutions from a planning graph directly

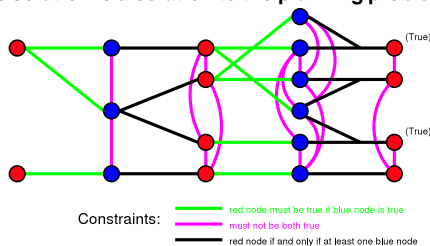
- There is another way to use the planning graph: to "extract" a solution from the planning graph directly!
- The algorithm (called **GraphPlan**) looks like this:


```
def GraphPlan(problem):
    graph = InitialPlanningGraph(problem)
    goals = problem.goals
    while True:
        if goals are all non-mutex in last level of graph:
            solution = ExtractSolution(graph, goals, graph.length)
            if solution != failure
                return solution
        graph = ExpandGraph(graph, problem)
```
- So the planning graph serves as a **filter**: if it tells us that a solution is impossible, we don't bother to try finding solutions.

AI(0270)-8.45

Extract a solution from the planning graph

The planning graph serves another purpose: it can be seen as a **boolean CSP**, and its **solution is a solution to the planning problem** as well!



So planning is as easy as a CSP with binary domains!

The CSP generated is in general easy to solve, since most constraints are made very explicit. E.g., the above problem can be done completely without backtracking using just forward checking!

AI(0270)-8.46

How well they work?

- All the algorithms we see are **complete** as long as a complete state-space search algorithm is used, and can be made optimal if admissible heuristic is used.
- POP** gives more **flexible** plans, but is also more difficult to write and make heuristic so that the algorithm scale.
- For efficiency, there is **no clear winner**—each type of planner is best for different kinds of problems.
- State-of-art: a **block world problem** with 20 steps and several dozens of blocks is the best that an optimal planner can do.
- Giving up optimality, one can solve problem a bit larger by using a more aggressive heuristic that is **not admissible**.
- For huge real-world problems, we have to **decompose** the problem into smaller ones.

AI(0270)-8.47