# Lecture 8

# Planning

The last chapter establish **logic** as an expressive language that we can use for reasoning. We also see how automatic reasoning be done using logic.

In this chapter we turn to another way to use the expressive power: as a language that we can use to express **searching** problems.

**Reference**:

- Chapter 11 and 12.3–12.6.



## The planning problem

The planning problem is different from the entailment problem by having a state that is expected to be **changing**.

- Initially, we are at a state, at which certain **predicates** are known to be true, some are known to be false.
- There are some **action schemas** that can be **instantiated** and **applied** to a state, provided that some **preconditions** are satisfied. An action can make some predicate to become true or become false.
- Goal: We want to find a sequence of operators that ends up in a state where **some predicate holds**.

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.

Different planning language has different properties in this general framework. We will see some of them.



## A (really simple) planning problem instance

- Initially true: $At(P_1, SFO)$, $Plane(P_1)$, $Airport(SFO)$, $Airport(JFK)$, $Airport(HKG)$.
  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.



## STRIPS representation

- **State** representation. It is a conjunction of predicates. The objects in the predicates must be constants.
  E.g., $At(P_1, SFO) \land Plane(P_1) \land Airport(SFO)$. Predicates not in the list is assumed false: "**close-world assumption**".
  We will abbreviate *HaveEgg*() as *HaveEgg*, like a proposition.
- **Goal** representation. It is a set 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

ADL (Action Description Language) is a more expressive language than STRIPS, used for planning. 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, one has to write $KnowAt(P_1, SFO)$ to emulate this. In ADL, there is no way for actions to create a new unknown predicate.
2. **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, a query of variable is not possible. In STRIPS, one can use an extra symbol to represent the disjunction, but it is difficult to manage its truth value.
3. **Effects** can be conditional: some of the predicates in the effect list may be conditioned on another predicate.
   This can be emulated in STRIPS by turning an action schema into many, one for each condition. ADL makes it more natural.



## More example: Spare tire problem

- Initially, we have a tire in a car gone flat, and a spare tire in the trunk.

  $At(Flat, Axle) \land At(Spare, Trunk)$
- Goal: having the spare tire installed: $At(Spare, Axle)$.
- Action 1: $Remove(Spare, Trunk)$: precond $At(Spare, Trunk)$, effect $\neg At(Spare, Trunk) \land At(Spare, Ground)$.
- Action 2: $Remove(Flat, Axle)$: precond $At(Flat, Axle)$, effect $\neg At(Flat, Axle) \land At(Flat, Ground)$ (We can merge the two *Remove* actions).
- Action 3: $PutOn(Spare, Axle)$: precond $At(Spare, Ground) \land \neg At(Flat, Axle)$, effect $\neg At(Spare, Ground) \land At(Spare, Axle)$.
  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 $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \ldots \land \neg At(Flat, Trunk)$

## 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 goal might only consist of partial information, but standard searching allow for a function as goal test. So standard searching techniques can be used to solve the problem completely.
- There is one problem, though: there can be a **huge** number of operators, since every instantiation of every object gives an operator.
- Unless there is a very good heuristic function, the whole search algorithm would break down by just a moderate size question.

  But then, that's not too surprising: planning is PSPACE-complete. Most real world planning problems are simpler, though.

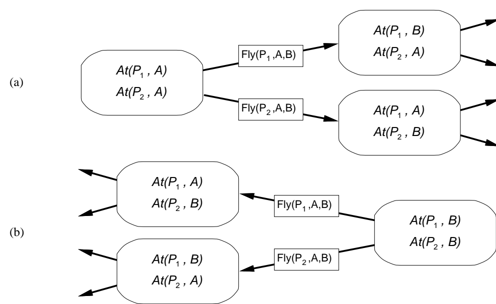


## Backward state-space search: Regression planning

- If our goal contains only a few predicates that **can be independently established**, then we can reduce branching factor by **regression**, i.e., search backwards.
- But **how** to know what is the state before the goal? If we were having a general search problem, we can't. But in planning…
- The goal is a **partially specified state**, consisting of predicates. So instead of keeping *what is true in the current state*, we will keep **what is still needed before we know how to arrive at the goal**.
- **How to work backwards**? I.e., given "what is needed before we know how to get to a goal" and an action as the *last* step, **what is needed so that we can execute that action and achieve the effect**?
- Simple: remove those needed predicates that the action achieves, and add those that the action requires.



## Comparison




(a) is progressive planning, (b) is regressive planning.



## Relevance of actions

E.g., if our final goal is $At(P_1, JFK)$, and our action is $Fly(P_1, SFO, JFK)$…

- 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 list of needed predicates.

Clearly, the last step must achieve something that is needed. E.g.,

- The action $Fly(P_2, SFO, JFK)$ produce nothing needed.
- If a plan $[P|Fly(P_2, SFO, JFK)]$ succeed in turning the initial state to a goal state, then $P$ can also do so.
- We say $Fly(P_2, SFO, JFK)$ is not **relevant**.



## Consistence

- Also, the last step must not **remove** some predicates that is needed.
- E.g., if what we need is $At(Spare, Ground) \wedge \neg At(Flat, Axle))$…

  This is the next step after one determine that to achieve $At(Spare, Axle)$, the only way is to execute $PutOn(Spare, Axle)$.
- Then *LeaveOvernight* is not an allowed last action even though it achieves $\neg At(Flat, Axle)$—because it removes $At(Spare, Ground)$.
- We say the *LeaveOvernight* action is not **consistent** with the needed state.
- In regression planning, we search only **relevant** and **consistent** actions. Each such action is processes as we mentioned before: add predicates in precondition, delete predicates in effects.
- **Question**: can we restrict "relevent" actions to those with an effect not already satisfied in the initial state?



## In search language…

Regressive planning as a state space search:

- **State**: a tuple containing a partially specified state and a plan that go from that partially specified state to the goal.
- **Initial state**: the tuple $(PlanningGoal, [])$.

  I.e., to go from the planning goal to the planning goal, the empty plan suffices .
- **Goal state**: a state in which every predicate in the partially specified state is known to hold initially in the planning problem.
- **Successor function** of a state $(Needed, Plan)$: for each relevant and consistent action $A$ instantiated from any action schema, a tuple $(Needed', [A|Plan])$, where…
  - $Needed'$ is the created by adding predicates in precondition of $A$ to $Needed$ and then removing the predicates in effects of $A$.

    This is said to be "regressing through the action $A$".

## Heuristics in State-space search planners

- Both progressive and regressive planners are unusable without good heuristic, since the number of possible actions overwhelm all *uninformed* search algorithms.
- How about **informed** search algorithms? Can we use A* search, in particular? To do so we need a **heuristic function** that is **admissible**, or nearly admissible.
- One simple way: count the number of unsatisfied predicate (for progressive planners, compare with 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.
- Even then, this can be used to improve the search time, in expense of a sometimes slightly suboptimal plan.



## Better heuristics

- Is it possible to remove the over-estimate?
- Most admissible heuristic comes from **relaxing** the problem, i.e., removing some constraints. We can do the same for planning.
- We can remove all preconditions of all actions, and remove all negated predicates in every effect. The never lengthen the shortest path, but make the problem much easier. E.g., if the relaxed problem is…

  $Goal(A \wedge B \wedge C)$
  $Action(X, EFFECT : A \wedge P)$
  $Action(Y, EFFECT : B \wedge C \wedge Q)$
  $Action(Z, EFFECT : A \wedge P \wedge Q)$
- We need at least 2 steps to arrive the goal: $X$ and $Y$.
- The problem is to find a minimal set of actions that "covers" all the required goals. It is the well known problem "set cover".



## Better heuristics (cont'd)

- 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.
- The relaxation done is quite large. It is possible to relax less: we can instead **just remove every negated predicate in the effect list**, without touching the precondition—"**empty-delete-list**".
- The result: we can easily decompose the problem into many subproblems, since everything established will never be removed.
- We need to run a simple planning algorithm to calculate the heuristic function, so it is quite costly.
- Best state-space search planner so far: progressive planners with empty-delete-list as heuristic.



## 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 case we prefer **leaving ordering unspecified**, because:
  - It is **faster**, because 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.



## Example problem

Consider another very simple planning problem:

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

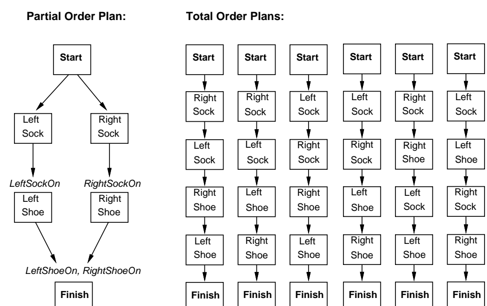
This involve two more or less **independent** tasks: deal with the left socks and shoe, and deal with the right socks and shoe.

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…



## Partial Order Plans




If we specify a plan as in the left, we probably don't need any backtracking. If we want a plan on the right, we probably need to consider all plans of 1–5 step and conclude they all won't work.

## 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.**



## 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 $A \stackrel{C}{\rightarrow} A'$.

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

We consider Start and Finish as actions, to make things simpler. In particular, we can say “all preconditions are caused” rather than “all preconditions and the goal predicates are caused”.

If $A \stackrel{C}{\rightarrow} A'$, then $A \prec A'$. So we won’t draw the ordering constraints when there is a causal link.



## 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: $\{RightSock \prec RightShoe, LeftSockt \prec LeftShoe\}$
  Links: $RightSock \stackrel{RightSockOn}{\rightarrow} RightShoe$, $LeftSock \stackrel{LeftSockOn}{\rightarrow} LeftShoe$,
  $RightShoe \stackrel{RightShoeOn}{\rightarrow} Finish$, $LeftShoe \stackrel{LeftShoeOn}{\rightarrow} Finish$

  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.



## Conflicts

- There is one important use of the causal link: to handle the complications arising from one step removing the effect of another.
- We show this by an example: suppose $A \stackrel{C}{\rightarrow} A'$. 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 $A \stackrel{C}{\rightarrow} A'$.
- To prevent this, we require that either $B \prec A$, or $A' \prec B$ must be in the plan. This guarantees that $B$ will be done outside the **protected interval** $[A, A']$. If so, we say the conflict is **resolved**.



## Requirements for correct partial-order plans

Now we sum up the requirement of a correct partial order plan.

- The plan should have some **actions**, including the special actions *Start* which has all initial conditions as effects, and *Finish* which has all goal as preconditions.
- The plan should have some **ordering constraints**. In particular, $Start \prec A$ and $A \prec Finish$ for any action $A$. There must be no cycles formed by the ordering constraints.
- All preconditions of all actions must be supported by a **causal link**. Each causal link has the corresponding ordering constraints.
- All conflicts must be resolved by having ordering constraints preventing conflicting actions from taking place during the protection interval.

It should be clear that the execution of any linearization of such a plan will result in all predicates of the goal becoming true.



## State-space search in plan-space

- It might seems that to come up with such a plan, we have to invent a new algorithm. But in fact it’s not the case.
- **We can do state-space search** in a special state-space: the **plan space**. Any standard algorithm can be used to carry out the search.
- The target is to find a correct plan for the problem. So this is the goal state of our state-space search.
- What are the intermediate **states**? Each state is a **plan** which is **incorrect**. But we restrict the “incorrectness”: it can only be incorrect because **some preconditions are not supported** by casual links. All other requirements of a correct plan must hold.
- We say a plan is **consistent** if it satisfies the above requirement. Each of them has a set of **open preconditions**, i.e., preconditions that are not supported. If the set is empty, the plan is a **solution**.

## State-space search formulation of POP

- **Initial state**: a plan with two actions *Start* and *Finish*, ordering constraint $Start \prec Finish$. Every precondition of *Finish* is an open precondition.
- **Goal state**: any solution.
- **Successor function**: Given any consistent plan, the succossor is formed as follows:
  1. Pick any open precondition. (Any precondition is okay. We don't need to try them all—it will have to be resolved sooner or later.)
  2. Every way to add a causal link to support it (while maintaining consistency) is a successor. This includes adding a causal link from an existing action, or adding a new action.
  3. Each way to resolve any conflict is a different successor.
- **Cost function**: each action costs 1, everything else is free.



## Example: Spare tire problem

- We show the problem again for easy reference…

  $At(Flat, Axle) \wedge At(Spare, Trunk)$

- 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 $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \ldots \wedge \neg At(Flat, Trunk)$



## Example POP Planning

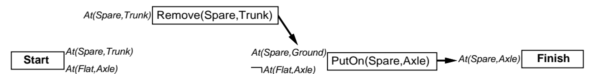
Consider the spare tire problem. Suppose we are doing IDS, have tried all plans with cost <= 2, and find them all failed. Now we allow for cost 3.

- 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.



## Example POP Planning (2)

We now have this plan…

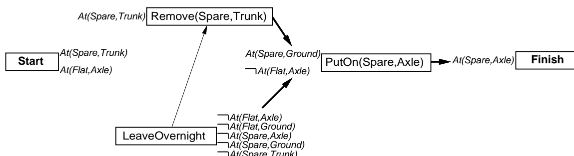



- We now want to pick yet another open precondition to support. There are two, and again we can pick any. Suppose we pick $\neg At(Flat, Axle)$.
- There are two ways to support $\neg At(Flat, Axle)$: either we add *LeaveOvernight*, or we add $Remove(Flat, Axle)$. They give different successors, and must 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 $Remove(Spare, Trunk)$…



## Example POP Planning (3)

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

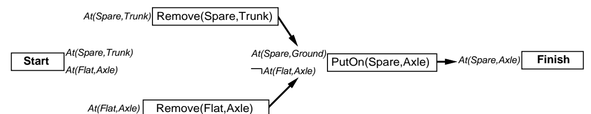



- Now there is only one open precondition: $At(Spare, Trunk)$.
  We have also used up our cost limit. Now more action can be added now.
- The *Start* action may provide a casual link for the open condition. But *LeaveOvernight* conflict with it…
- And both ways to resolve the conflict results in cycles!

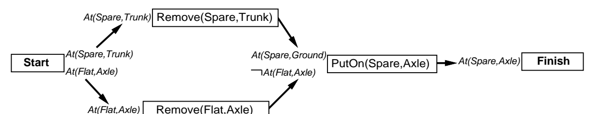


## Example POP Planning (4)

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.



## 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.



## Some complications

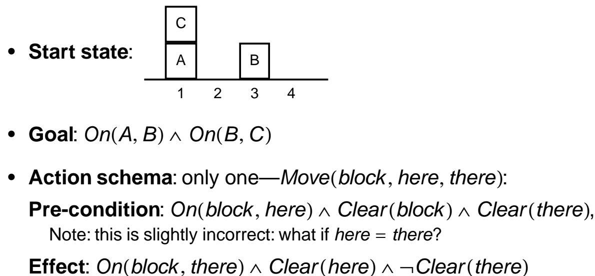
- When the sub-goal can have variables waiting to be binded, **actions and causal links may also have variables waiting to be binded**.
- There is one complication: $Fly(P_1, SFO, SFO)$ won't work!
  In particular, it will not establish $\neg At(P_1, SFO)$.
- Another problem: when causal links and effects 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 complicates the successor function.
- Another solution: **allow 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.



## 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**: $On(block, there) \wedge Clear(here) \wedge \neg Clear(there)$



## 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**, subtract the number of them that is satisfied by the *Start* state.
- 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.



## Even better heuristic

- The heuristics that we mentioned in the last slide is not accurate, because actions may have **negative interaction**.
- I.e., the effect of one action may undo the precondition achieved by another, and vice versa.
- In this case the two actions actually cannot provide all the required preconditions, although the heuristic function is ignorant about it.
- It is **impossible to consider all negative interactions**: it would spend too much time.
  The problem then becomes as difficult as the original planning problem.
- But if we can capture **some** of the negative interactions, then our heuristic can be much more accurate than those that just count perdicates.
- The techniques work with **propositions**. FOL sentences must be propositionalized—whether it is worthwhile depends on problems.



## The idea of a planning graph

- The idea: only capture "trivial" negative interactions, that **two things cannot occurs at the same time**.
- The "things": **actions** or **literals** (positive or negative predicates).
- To find the heuristic, we create a "planning graph", which capture what can be true at a particular step, as well as what cannot happen together: **mutex** ("mutual exclusion").
- By definition, the first step has all the literals in the initial state (or current state in case of progressive planners). There is no mutex.
- Using the information of the possible literals at a step, the actions that can be done are found. We also calculate the mutex between the actions.
- Using the information about the possible actions, the possible literals of the next step is calculated, with the mutex.

## Example

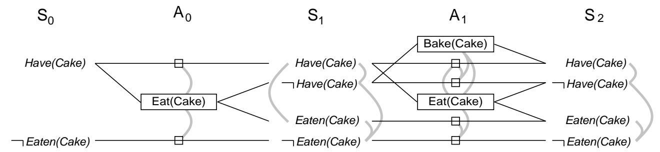
We use an extremely simple example to illustrate the idea…

*Init*(*Have*(*Cake*))
*Goal*(*Have*(*Cake*) ∧ *Eaten*(*Cake*))
*Action*(*Eat*(*Cake*),
PRECOND: *Have*(*Cake*)
EFFECT: ¬*Have*(*Cake*) ∧ *Eaten*(*Cake*))
*Action*(*Bake*(*Cake*),
PRECOND: ¬*Have*(*Cake*)
EFFECT: *Have*(*Cake*))

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



## Example planning graph




- At step 0, both *Have*(*Cake*) and ¬*Eaten*(*Cake*) are known to be true.
- One can only take the *Eat*(*Cake*) action, or the "persistence actions" represented by the empty boxes to leave a predicate unchanged.
- One cannot eat the cake and leave *Have*(*Cake*) unchanged at the same time, so there is a mutex between them. Similar for ¬*Eaten*(*Cake*).
- At step 1, any possible literal may be true. But of course, *Have*(*Cake*) and ¬*Have*(*Cake*) cannot happen at the same time…



## Example planning graph (2)

- At step 1, *Have*(*Cake*) and *Eaten*(*Cake*) cannot appear at the same time:
  - *Have*(*Cake*) can only be produced by the persistence action.
  - *Eaten*(*Cake*) can only be produced by the *Eat*(*Cake*) action.
  - But they are mutex to each other!

  So we have a mutex between *Have*(*Cake*) and *Eaten*(*Cake*).
- Similarly, ¬*Have*(*Cake*) and ¬*Eaten*(*Cake*) cannot be both true at step 1, and have a mutex between them.

Note that a counting-only heuristic would **assume that it is possible**, since they ignore the interactions by removing the delete list.

So even such simple example shows the strength of planning graph.



## Example planning graph (3)

- At step 1, both *Bake*(*Cake*) and *Eat*(*Cake*) can be done, although they are mutex to each other (i.e., cannot be both done) because of various reasons.
  When a planning graph is used for heuristic computation, any two actions that are not persistence actions should be mutex to each other, anyway.
- Or is it? The first step only has one possible action: *Eat*(*Cake*). So at step 1 the only possibility is that the cake is no longer there, and cannot be eaten again.
- But when drawing planning graph, **we always allow inaction**. So step 1 can be "do nothing".
  This makes sure that the set of doable action will always grow, and thus guarantees that any planning graph would eventually level off.
- At step 2, the mutex between *Have*(*Cake*) and *Eaten*(*Cake*) is removed. There is no change in the next step, so we say the graph **levels off**.



## Definition of mutex

Formally, two literals $L_1$ and $L_2$ in a step (or "level") are mutex if:

- For any action $A_1$ that results in $L_1$, and for any action $A_2$ that results in $L_2$, $A_1$ and $A_2$ are mutex to each other.
  This is trivially true if $L_1 = \neg L_2$.

Two actions $A_1$ and $A_2$ in a step are mutex if:

- The actions produce opposite effects—inconsistent effects.
  E.g., *Bake*(*Cake*) and *Eat*(*Cake*) produces *Have*(*Cake*) and ¬*Have*(*Cake*).
- $A_1$ produce an effect that negates a literal needed by $A_2$—interference.
  E.g., *Eat*(*Cake*) removes the precondition *Have*(*Cake*) of the presistence action for *Have*(*Cake*).
- A literal $L_1$ of $A_1$ and a literal $L_2$ of $A_2$ are mutex—competing needs.
  E.g., *Bake*(*Cake*) requires ¬*Have*(*Cake*) but *Eat*(*Cake*) requires *Have*(*Cake*).



## Using a planning graph for heuristic

- Whenever we use a planning graph for heuristic computation, there is a set of literals that are known to be true, and a set of literals that we want to become true:
  - Progressive planner: current state literals known to be true, desire goal predicates to be true.
  - Regressive planner: initial state known to be true, desire literals in the current state to become true.
  - POP: initial state known to be true, desire open preconditions to become true.
    There is a flaw here to make this heuristic become inadmissible in POP. Do you see it?
- How to estimate? we follow the planning graph to find the first state that all the desired literals are true **and not mutex** to each other. The step number is the estimate.

## Extracting solutions from a planning graph directly

- There is another way to use the planning graph: to produce a solution from the planning graph directly.
- So instead of a regular search algorithm, the algorithm 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)
```

- When using such an algorithm, we will usually allow actions in a planning graph step to be done in parallel.



## Extract a solution from the planning graph

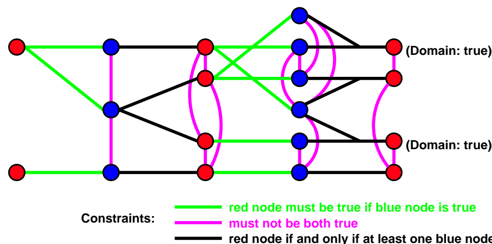
How can we try to extract a solution from the planning graph? It turns out that the planning graph specifies a **boolean CSP** that we can solve using a CSP algorithm instead of a state-space search algorithm!

- Variables: each action and literal in each step of the planning graph.
- Domains: either true or false.
  - An action variable is true iff the action is taken at that step.
  - A literal variable is true iff it is true at a planning step.
- Constraints: follow the edges of the planning graph.
  - From literal L variables to actions A: if A, then L.
  - From action variables A1, A2, … to literals L: L iff A1 $\vee$ A2 $\vee$ …
  - Mutex: must not be both true.



## Extract a solution from the planning graph (cont'd)

Adding the constraints that everything in goal must be true at last step, we get the CSP instance that any solution represents a plan…




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

The above problem can be done completely without backtracking using just forward checking!



## How well they work?

- Clearly, progressive, regressive and POP planners are all complete and optimal as long as the state-space search algorithm used is complete. I.e., if there is a solution, it is guaranteed that it can be found.
- GraphPlan can be made to be complete and optimal algorithm if all (non-persistence) actions are made to be mutex.

  There is a variant which turn the problem into satisfiability and use DPLL to solve the problem. It behaves similarly.
- So the difference is basically efficiency. Unluckily, 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 a planner can do.
- Larger than that size, we have to decompose the problem into smaller ones, and settle for sub-optimal solution.



## HTN Planning

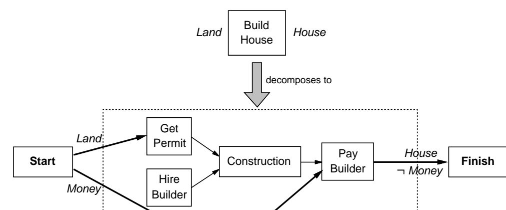
- How to "decompose the problem"? The idea is the same for basically all fields: do it in a **hierarchy**.

  We actually seen this once before, when we talk about CSP.
- In planning, the method is known as **hierarchical task networks**, or HTNs.
- The initial trivial plan is viewed as a **very high level plan** which describes how to solve the problem.
- Plans are **refined** by **action decompositions**, until all actions becomes **primitive** actions that can be readily performed.
- Each high level action may have **many different ways** to be decomposed.
- The planner **chooses among all decompositions** to find one which can build up a full plan.



## Plan library: how to decompose

The planner should have a **plan library**, which lists the possible decompositions. Each decomposition is something like this:




One interesting thing to note: the decomposed plan can have **more** precondition than the high-level action.

So a complete plan can become incomplete after decomposition.

## The rule stored…

The plan library thus have the following decomposition that can be chosen by a HTN planner when decomposing *BuildHouse*.

$$
\begin{aligned}
&\mathit{Decompose}(\mathit{BuildHouse},\\
&\quad \mathit{Plan}(\text{STEPS}:\{S_1:\mathit{GetPermit}, S_2:\mathit{HireBuilder},\\
&\qquad\qquad S_3:\mathit{Construction}, S_4:\mathit{PayBuilder}\},\\
&\qquad \text{ORDERINGS}:\{\mathit{Start} \prec S_1 \prec S_3 \prec S_4 \prec \mathit{Finish}, \mathit{Start} \prec S_2 \prec S_3\},\\
&\qquad \text{LINKS}:\{\mathit{Start} \xrightarrow{Land} S_1, \mathit{Start} \xrightarrow{Money} S_4,\\
&\qquad\qquad S_1 \xrightarrow{Permit} S_3, S_2 \xrightarrow{Contract} S_3, S_3 \xrightarrow{HouseBuilt} S_4,\\
&\qquad\qquad S_4 \xrightarrow{House} \mathit{Finish}, S_4 \xrightarrow{\neg Money} \mathit{Finish}\}))
\end{aligned}
$$

It also has the following high-level action:

$\mathit{Action}(\mathit{BuildHouse}, \text{PRECOND:}\mathit{Money}, \text{EFFECTS:}\mathit{Land} \wedge \neg \mathit{Money})$



## Information hiding

The major incentive of hierarchical planning is to **delay dealing with details** until the "big picture" is planned.

So intrinsically, HTN hides information:

1. Some preconditions of the plan is not in the high-level actions. Indeed, the preconditions of a high-level action should be the **intersection** of the preconditions of all plans that it can decompose into.
2. There are **internal** preconditions that is not the primary aim of the full plan, and is thus not shown. E.g., *Permit*, *Contract*, etc.
3. High level description does not tell when, **within the action**, the precondition must be true and the effect become true.
   All that is known is that the preconditions are needed before the plan completes, and the effect will become true after the action.



## HTN planners

How to take advantage of action decomposition?

- POP is the best choice to start with: others won't work well…
  - Progressive and regressive state-space planners require planning in a particular order, while decompositions introduce problems in any ordering.
  - It is difficult to introduce decomposition into GraphPlan.
- To deal with HTN, the POP successor function can be modify to **allow results of decompositions to be a possible successor**.
- In a pure HTN planner, everything is done by decomposition. In a **hybrid** planner, both decomposition and adding causal link may be done.
  We prefer a hybrid approach. It is not really more powerful (adding causal link can be expressed into decompositions), but is more natural.



## Subtask sharing

- How decomposition take place? Just put the decomposed plan in place of the original high-level action?
- Sometimes we want to be smarter. E.g., if we also want *MakeWine* which require *BuyLand* in the decomposition, it is desirable to share the *BuyLand* step (or actually, the *Land* effect).
- Each action in the decomposition can be added in two ways:
  - **Reuse** an existing action.
  - Create a **new** action for the decomposition.
- **Internal** ordering constraints and causal links are **copied over**. There may be some **new conflicts** that needs to be resolved, and we may have new **open precondition**, after decomposition.



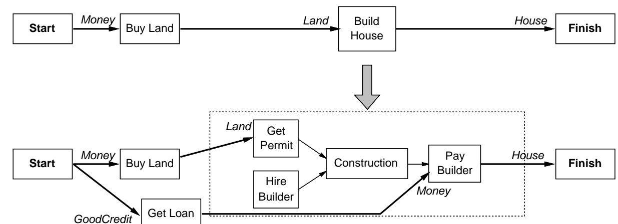
## Hooking up the causal links and ordering constraints

- What about those constraints and links of the **high-level plan**?
- Causal links are simpler: if the original plan has $A \xrightarrow{C} B$ where $B$ is decomposed, then we can just **find where** $C$ **points to** from *Start*, and add a link there. The other direction (to *Finish*) is similar.
- What for ordering constraints of high-level plan $A \prec B$ where $B$ is decomposed? A simple solution: just add $A \prec B'$ for every action in $B$.
- But this is too restrictive. It is probably better to remember *why* the ordering constraint is there.

And… we finish in making a HTN planner!



## Results



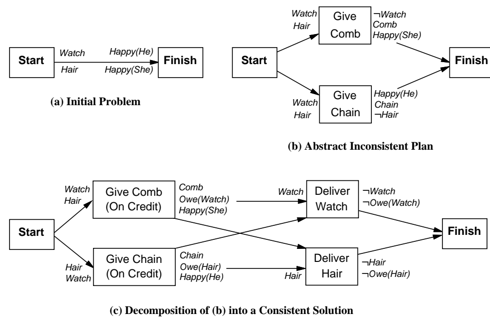

Here, there is no subtask sharing possibility. Ordering constraint of *HireBuilder* is not from *BuyLand*, but instead from *Start*. *Money* in *PayBuilder* is made satisfied by a further action.

It cannot be done by a causal link from *Start*, for *BuyLand* has an effect which negates *Money*.

The plan will then be further refined if any of the steps is not primitive.

## Problem: hidden information can prohibit plans!

Sometimes seemingly impossible high-level action can actually be done!




The problem: the high-level action hides "when" an effect is achieved.



## Hybrid approach

- Luckily, there is usually not many cases of such problems. One possible strategy is to just ignore them.
- Alternatively, one can add all the component action in a decomposition to the planner as a normal action as well.
- If the plan library fails to find decompositions, then regular causal link adding step will try adding them.
  Although slowly, probably after most of the decompositions are tried.
- One can view the plan library as something that **provides a strong heuristic**, so strong that using them is considered "cheaper" than using regular steps.
  This results in suboptimality, but we have to accept it.



## How much better?

A highly idealized situation…

- If each precondition has 3 valid way to satisfy, and there are 50 preconditions to add in best plan, then regular POP needs to examine $3^{50}$ "nodes" in the state-space search engine (pretty much undoable).
- If each high-level action decompose to 10 preconditions, and at any time there are 100 possible ways to do decomposition each time…
  Including the number of ways to do subtask sharing, deal with new open preconditions, etc…
  - We can expect the number of levels in the decomposition tree to be around $50/10 = 5$ (since each adds that many preconditions).
  - So the number of internal nodes in the state-space search engine is $100^5$, which is much more managable.



## Downsides

- Regular decomposition is not always possible.
- The result can be suboptimal.
- Good decompositions can be difficult to find.

But… virtually all large scale planners in use is a HTN planner.