# Planning

Planning consists of finding a plan to achieve a certain goal. A plan is a sequence of actions that make it possible to achieve a goal. As we can see, Planning looks similar to the Search-problems previously analysed but while it shares certain characteristics, it uses a **factored representation** for representing the states rather than an atomic one.

Planning problems are defined through a specific language called PDDL (Planning Domain Definition Language).

The similarity of Planning to Search problems is that in Planning problems we need to define the following as well:

1. Initial state
2. Actions that are possible to perform in a state
3. The result of each action in a given state
4. Goal state

## State representation

In PDDL each state is represented as a conjunction of fluents that are ground, functionless atoms. For example, Poor ∧ Unknown might represent the state of a hapless agent, and a state in a package delivery problem might be At(Truck 1, Melbourne) ∧ At(Truck 2, Sydney).

Database semantics is used: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that Truck 1 and Truck 2 are distinct. The following fluents are not allowed in a state: At(x, y) (because it is non-ground), ¬Poor (because it is a negation), and At(Father (Fred), Sydney) (because it uses a function symbol). The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a set of fluents, which can be manipulated with set operations. The set semantics is sometimes easier to deal with.

In other words, each state in PDDL is defined as a conjunction of fluents (literals that change over time) which are positive, ground (variables do not appear) and functionless (no functions inside).

Given that each state is represented as a conjunction and all fluents not mentioned are false then it is easy to represents each state as a set of fluents. This makes it easier to deal with states when implementing planning through some programming language.

The database semantics consists of the following assumptions:

* Closed-word assumption => the fluents not mentioned in a state are considered to be false
* Unique name assumption => each different constant symbol in First-Order logic refers to a different object in the DOMAIN of the model

## Actions representation

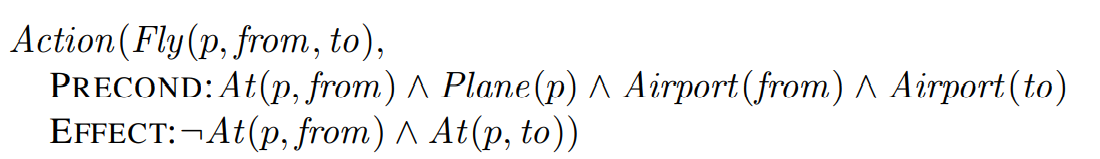
Classical planning concentrates on problems where most actions leave most things unchanged. PDDL does that by specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned.

In PDDL, actions are represented through action schemas.

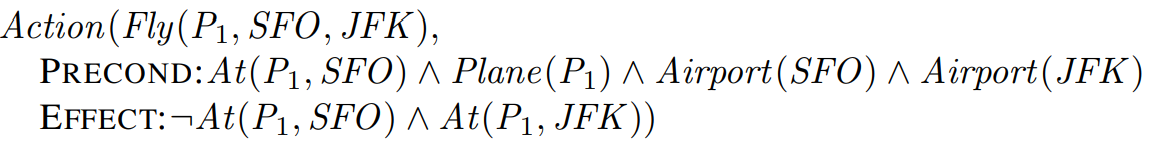
The schema consists of the action name, a list of all the variables used in the schema, a precondition and an effect.

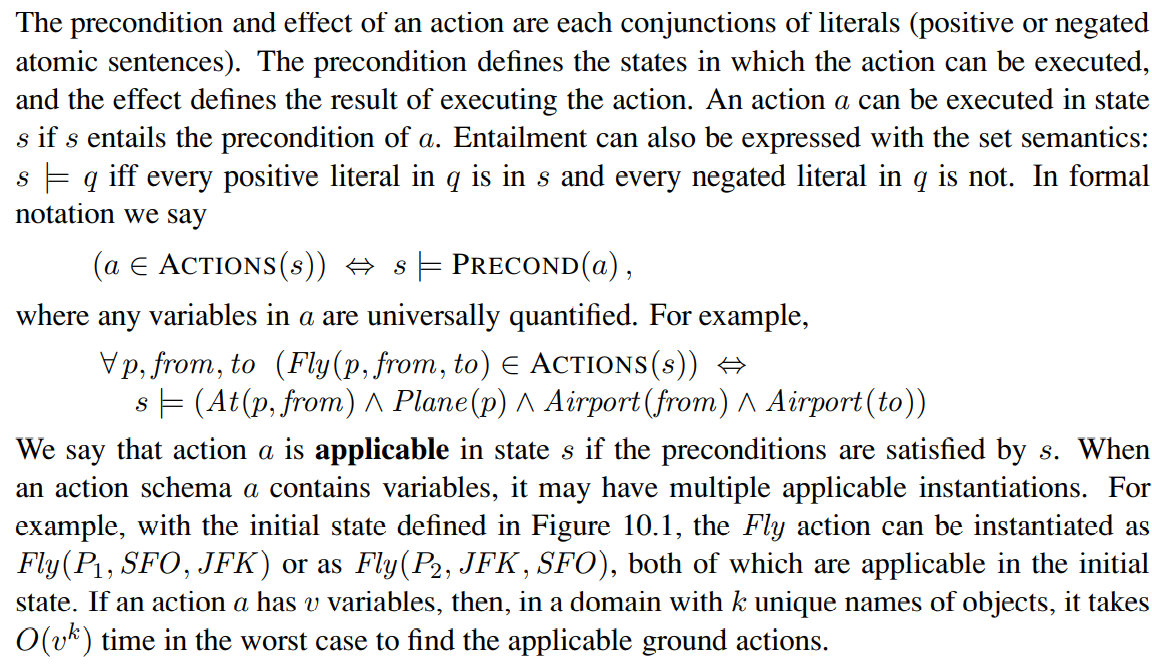
Therefore, action schemas are non-ground as they contain variables. However, this is a positive thing because through a single action schema it is possible to instantiate different ground actions by replacing the variables with some objects.

As an example given the following action schema:



It is possible to create the following ground action by replacing the variables with some objects:



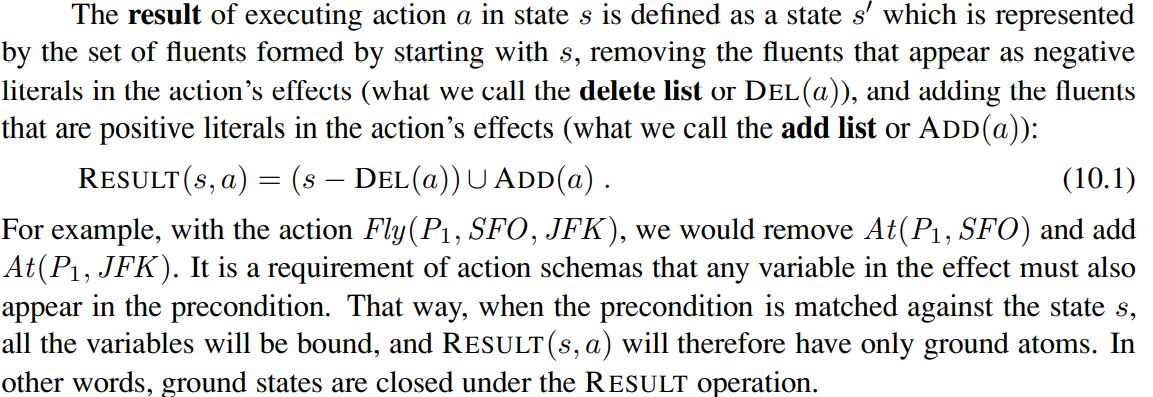


In other words, a ground action a is applicable in a state s if the precondition of action a is satisfied by state s. A state s satisfies the precondition of an action if s semantically entails the precondition.

By using a set semantics, we can define entailment in the following way (considering that PDDL uses the database semantics):

**s ⊨ q iff every positive literal in q is in s and every negative literal in q is not in s**

where q = precondition of a



In other words, given that an action’s effect outputs only what changes and not what has been left unchanged then given that an action a has been performed on state s then the resulting state is the following:

s’ = Result (s,a) = (s – Del(a)) ∪ Add(a)

In other words, from state s we need to remove all negative literals that appear in the effects of action a and we need to add al the positive literals that appear in the effects of action a.

## Initial state representation

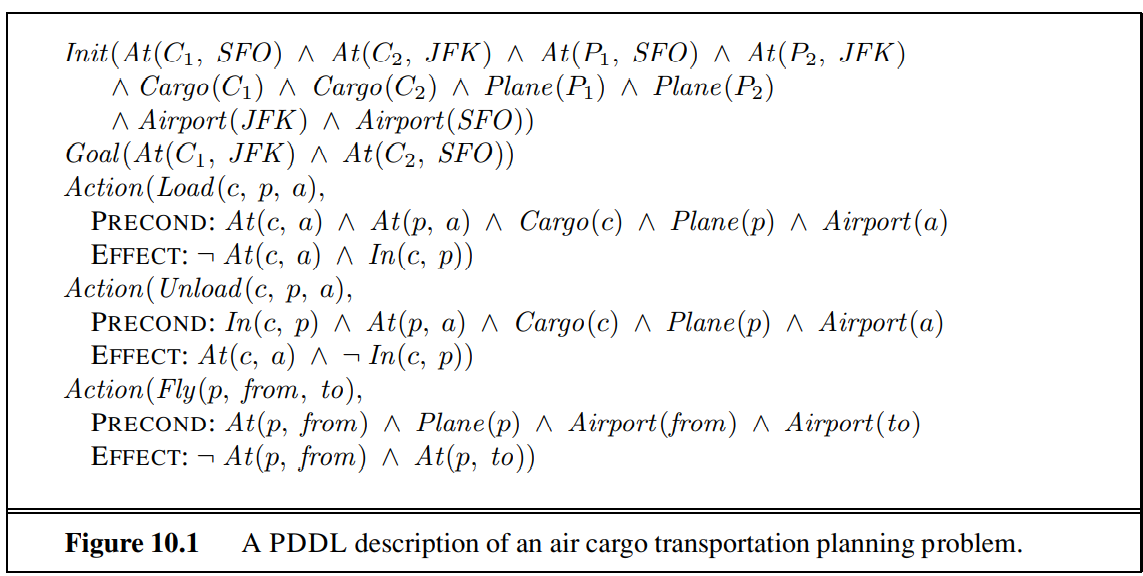
The initial state representation is equal to the state’s representation mentioned above. Thus, the database semantics is used and it is a conjunction of ground, positive, functionless fluents.

## Goal representation

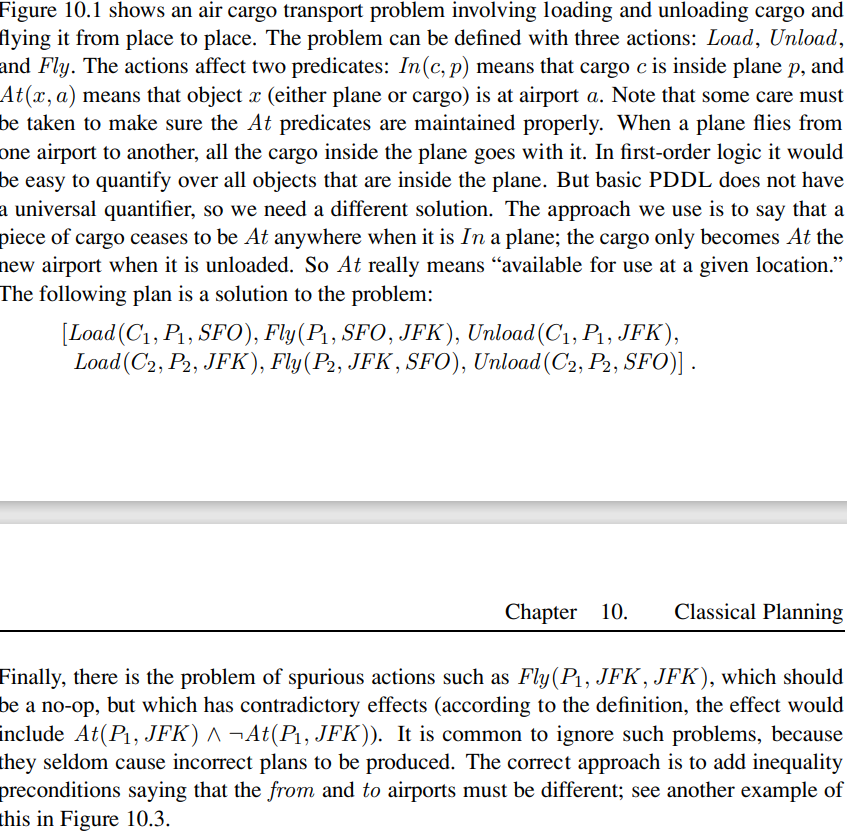
The goal is just like a precondition: a conjunction of literals (positive or negative) that may contain variables, such as At(p, SFO) ∧ Plane(p). Any variables are treated as existentially quantified, so this goal is to have any plane at SFO. The problem is solved when we can find a sequence of actions that end in a state s that entails the goal. For example, the state Rich ∧ Famous ∧ Miserable entails the goal Rich ∧ Famous, and the state Plane(Plane1) ∧ At(Plane1, SFO) entails the goal At(p, SFO) ∧ Plane(p).

Therefore, a planning problem consists of finding a sequence of actions from the initial state that lead to a state s’ which semantically entails the goal.

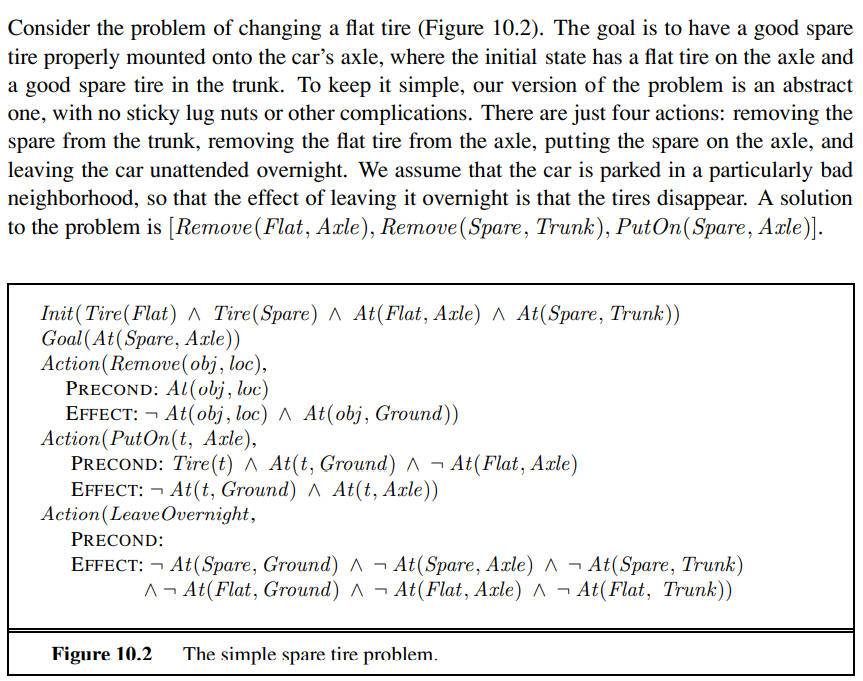
In the case of a goal with variables then that goal is treated as an existentially quantified conjunction.



## Air cargo transportation



## The spare tire problem



# Planning algorithms

There are two algorithms for solving planning problems:

1. Forward (progression) state space search
2. Backward (regression) state space search

## Progression state space search

Progression state space search consists of starting from the initial state and searching for a sequence of actions that reach a state s’ which semantically entails the goal. Thus, progression state space search can be implemented using any state space search algorithm previously studied.

The basic idea of progression state space search is to try all the applicable actions in a state s.

However, forward search is highly inefficient and not practical.

Consider the noble task of buying a copy of AI: A Modern Approach from an online bookseller. Suppose there is an action schema Buy(isbn) with effect Own(isbn). ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal. Second, planning problems often have large state spaces. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let’s say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 200041 nodes.

## Regression state space search

Regression state space search instead consists of starting with the goal and find a sequence of actions from the goal that reach a state s which is the initial state. However, we need to remember that a goal is not a state but it may represent a set of states which all semantically entail the goal. For example, the goal ¬Poor ∧ Famous describes those states in which Poor is false, Famous is true, and any other fluent can have any value.

Thus, the root node in the regression state space search is the goal state and the first level of the tree search consists of the states that semantically entail the goal.

Once all states that semantically entail the goal have been inserted in the first level of the tree space search then we need to go back (regress) from them so that to reach the initial state.

However, how is it possible to find the state that precedes another states?

In PDDL this is an easy task as all we have to do in order to regress from a state g is to find an action a which is **relevant** to the state g. In other words, the positive literals [ADD LIST] in the effect of a must be present in the state g and the negative literals [DELETE LIST] in the effect of a must not be present in g. Once we have found such an action a we can easily find the previous state, that is the state g’ where action a has been performed which has resulted in state g, with the following formula:

**g’ = ( g − ADD(a) ) ∪ Precond(a)**

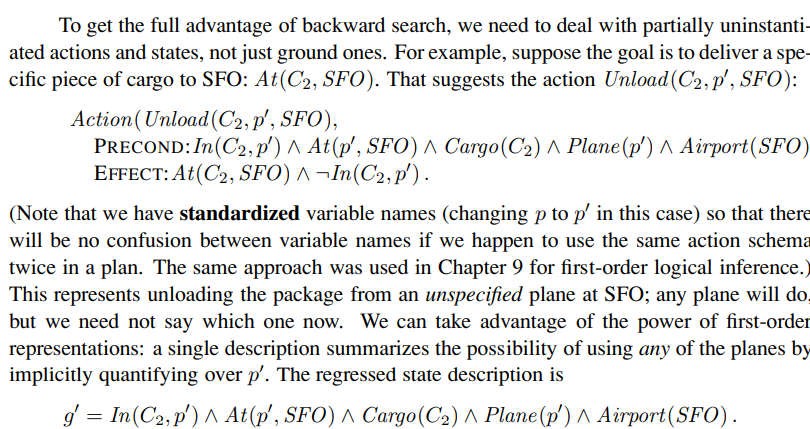
The above formula says that the state g’ that precedes state g given that action a has been performed in g’ is the state g removed with the ADD LIST of the effects of a (as previously to the action a such fluents had not to be in g’) and added with the precondition of action a as action a could have been formed on state g’ only if g’ satisfied the preconditions of action a.

Thus the algorithm to regress from a state g is the following:

* Find all actions a that are relevant to state g
* For each relevant action a regress to state **g’ = ( g − ADD(a) ) ∪ Precond(a)**

As we can see in the formula for finding g’ we do not use the [DELETE LIST] in the precondition of action a. We do so because we do not really know if the fluents in the DELETE LIST were true or false prior to the execution of action a. However, this is not a problem.

Thus, as we can see while in forward search space planning at each state s we try all possible applicable actions a, in regression search space planning at each state s we try all possible relevant actions a.



Given the goal At(C2, SFO), several instantiations of Unload are relevant: we could choose any specific plane to unload from, or we could leave the plane unspecified by using the action Unload(C2, p’ , SFO). We can reduce the branching factor without ruling out any solutions by always using the action formed by substituting the most general unifier into the (standardized) action schema.

Thus, as we can see, in regression state space search we can further decrease the branching factor furtherly by using general placeholders whenever possible rather than using fully ground actions as we can see in the example above. Once, we arrive at a certain level in the tree we will form a state with this placeholders which will be equivalent to the initial state and so all we have to do is to replace the placeholders that we have accumulated during the space search with the corresponding objects present in the initial state.

In order to find the sequence of actions all we have to do is to go up from the initial node to the goal and collect these actions in a list replacing the placeholders with the correct objects present in the initial state.

## Heuristic for planning

Neither forward nor backward search is efficient without a good heuristic function. Indeed, the problem of finding a plan is NP-HARD. Recall from Chapter 3 that a heuristic function h(s) estimates the distance from a state s to the goal and that if we can derive an admissible heuristic for this distance—one that does not overestimate—then we can use A\* search to find optimal solutions. An admissible heuristic can be derived by defining a relaxed problem that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

In planning problems it is possible to derive relaxed versions by using one of the following two strategies:

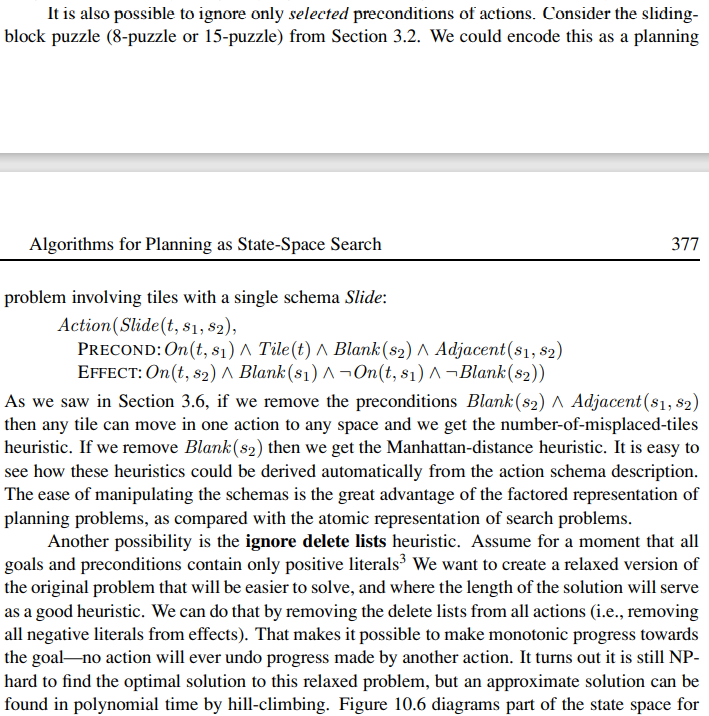
* Add more edges to the state space search
* Group some nodes together
* Decomposition

## Add more edges to the state space search

We look first at heuristics that add edges to the graph. For example, the **ignore preconditions heuristic** drops all preconditions from actions. Every action becomes applicable in every state, and any single goal fluent can be achieved in one step (if there is an applicable action—if not, the problem is impossible). This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others. For many problems an accurate heuristic is obtained by considering (1) and ignoring (2). First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal. Then, we count the minimum number of actions required such that the union of those actions’ effects satisfies the goal. This is an instance of the set-cover problem. There is one minor irritation: the set-cover problem is NP-hard. SET-COVER PROBLEM Fortunately a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of log n of the true minimum covering, where n is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

In other words with the ignore precondition heuristic all preconditions are dropped from every action.

This almost implies that the number of steps to reach a goal is equal to the number of fluents yet to be achieved. However, some actions can have as effects multiple goal fluents and other actions can lead to the removal of some fluents of the goal. This is why when using this heuristic usually the negative effects of actions are removed as well. However, solving this relaxed problem turns out to be NP-HARD as it is an instance of the SET-COVER problem.



Thus, the main ways of adding edges to a plan problem are:

1. Remove preconditions heuristic => remove all preconditions or only selected ones
2. Ignore delete list => allows to relax a planning problem by allowing monotonic progress towards the goal condition as no previously acquired fluents are removed.

## Decomposition

A key idea in defining heuristics is decomposition: dividing a problem into parts, solving each part independently, and then combining the parts. The subgoal independence assumption is that the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal independently. The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each subgoal—for example, when an action in one subplan deletes a goal achieved by another subplan. It is pessimistic, and therefore inadmissible, when subplans contain redundant actions—for instance, two actions that could be replaced by a single action in the merged plan. Suppose the goal is a set of fluents G, which we divide into disjoint subsets G1,...,Gn. We then find plans P1,...,Pn that solve the respective subgoals. What is an estimate of the cost of the plan for achieving all of G? We can think of each Cost(Pi) as a heuristic estimate, and we know that if we combine estimates by taking their maximum value, we always get an admissible heuristic. So maxi COST(Pi) is admissible, and sometimes it is exactly correct: it could be that P1 serendipitously achieves all the Gi. But in most cases, in practice the estimate is too low. Could we sum the costs instead? For many problems that is a reasonable estimate, but it is not admissible. The best case is when we can determine that Gi and Gj are independent. If the effects of Pi leave all the preconditions and goals of Pj unchanged, then the estimate COST(Pi) + COST(Pj ) is admissible, and more accurate than the max estimate.

Therefore, decomposition is based on the subgoal assumption which says that the cost of achieving a conjunction of fluents is approximately equal to the sum of the costs of achieving each fluent separately. However, such a sum could be either optimistic or pessimistic. The latter makes it an inadmissible heuristic. However, if we are granted that all fluents are independent of each other than the subgoal independence assumption leads to an admissible heuristic.

# Partially ordered plans

All the approaches we have seen so far construct totally ordered plans consisting of a strictly linear sequences of actions. This representation ignores the fact that many subproblems are independent. A solution to an air cargo problem consists of a totally ordered sequence of actions, yet if 30 packages are being loaded onto one plane in one airport and 50 packages are being loaded onto another at another airport, it seems pointless to come up with a strict linear ordering of 80 load actions; the two subsets of actions should be thought of independently. An alternative is to represent plans as partially ordered structures.

In the 1980s and 90s, partial-order planning was seen as the best way to handle planning problems with independent subproblems—after all, it was the only approach that explicitly represents independent branches of a plan. On the other hand, it has the disadvantage of not having an explicit representation of states in the state-transition model. That makes some computations cumbersome. By 2000, forward-search planners had developed excellent heuristics that allowed them to efficiently discover the independent subproblems that partial order planning was designed for. As a result, partial-order planners are not competitive on fully automated classical planning problems. However, partial-order planning remains an important part of the field. For some specific tasks, such as operations scheduling, partial-order planning with domain specific heuristics is the technology of choice. Many of these systems use libraries of high-level plans, as described in Section 11.2. Partial-order planning is also often used in domains where it is important for humans to understand the plans. Operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and verify that they are correct.

Now, we are going to see what are the elements that are necessary to describe a partially-ordered plan and the algorithm to find it.

A partially-ordered plan is a mathematical model consisting of the following elements:

1. A set of actions (called steps as well and denoted as S)
2. A set of ordering constraints (Si < Sj denotes that step i must occur prior to step j)
3. A set of variable bindings
4. A set of causal links

## Ordering constraints

The set of ordering constraints denotes which steps must occur before others. However, such an ordering constraint is partially-ordered which means that there is some freedom for some actions to be performed prior or after others.

## Variable bindings

The variable bindings set is a set of variables which are bound to some constant value or to some other variable. Variable are introduced during the recursive construction of partially ordered plans. Indeed, when some action Si is chosen to satisfy the precondition to an already added action Sj, some arguments of the action Si can assume any value at the moment since they do not influence the satisfaction of the precondition of Sj. Therefore, we use variables for such arguments. As the recursive algorithm continues eventually such variables will be bound to some constant or other variable.

So, for example, if you're trying to make a cake, and you have to have the eggs and the flour in the same bowl. You’d have an operator that says, "Put the eggs in [something]” and another that says “Put the flour in [something]." Let's say, you don't want to commit to which bowl you’re going to use yet. Then, you might say, "Well, whatever the bowl is that I put the eggs into, it has to be the same as the bowl that I put the flour into, but it's not yet Bowl32." So, that's the case where you would end up having a partial plan where you had a variable constraint, that this variable equal that variable.

## Causal links

A causal link is denoted as Si → cSj

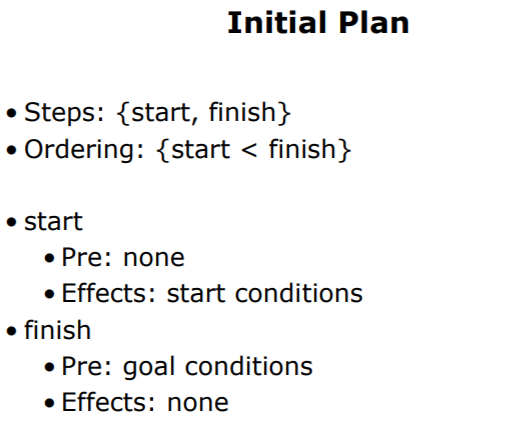
This means that action i satisfies the precondition c for the action j.

The set of causal links contains all such links in a partially-ordered plan and as we will see it is more of a bookkeeping thing as it will help us on performing the recursive algorithm for finding partially ordered plans.

## Brief description of the recursive algorithm

Let Init and Goal be the initial state and the goal condition respectively of an instance of planning problem which must be solved using the partially-ordered plan strategy.

The recursive algorithm consists of creating an initial plan that looks like this:



In other words, the initial plan consists of a set of two steps called Start and Finish.

A set of ordered constraints that tells that Start must occur prior to Finish.

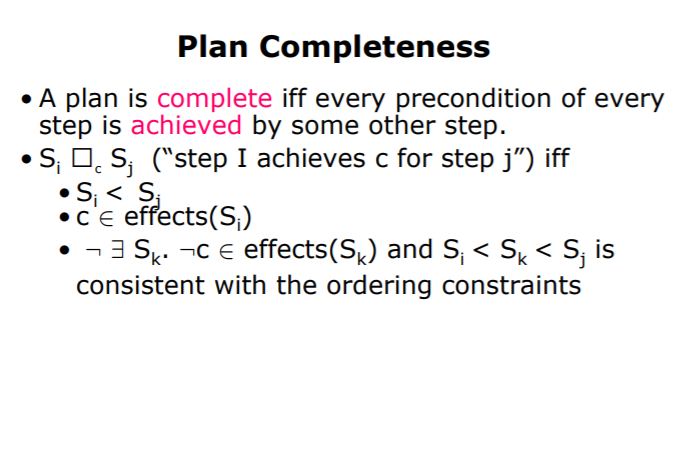
The step Start is an action that has as effect the conjunction of literals contained in the Init state of the partially ordered plan being solved. It has no preconditions.

The step Finish, instead, has no effect but has preconditions that are the conjunction of literals of the goal condition of the partially ordered plan being solved.

Starting from the initial plan, the recursive algorithm will recursively add steps, ordering constraints, variable binding and causal links until the partially-ordered plan is satisfactory.

A partially-ordered plan is satisfactory when it is complete and consistent.

## Complete partially-ordered plan



Thus, a partially-ordered plan is complete when every condition of every step is satisfied by some other step. The satisfaction of a precondition c of an action j is denoted through the causal links.

The full definition of Si → cSj is the following:

* Si must occur prior to Sj as otherwise cannot satisfy c for Sj
* The effects of Si must contain c
* There must not exist any step Sk with ⌐c as effects which according to the current ordering constraints can be placed between Si and Sj, otherwise it is not possible to satisfy c for Sj.

## Consistent partially-ordered plan

A plan is consistent iff the ordering constraints are consistent and the variable binding constraints are consistent. For a plan to be consistent, it is enough for the temporal ordering constraints to be consistent (we can’t have I before j and j before I) and for the variable binding constraints to be consistent (we can’t require two constants to be equal).

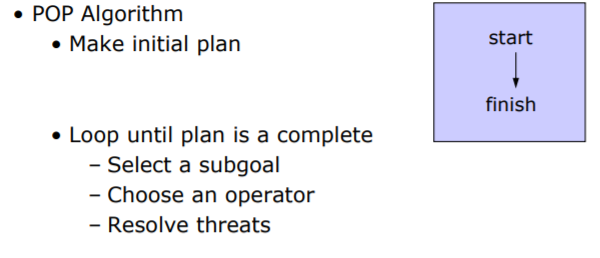
Given that the algorithm will always produce a complete partially-ordered plan (whenever it is possible to do so) then any totally-ordered plan which is consistent with the partially-ordered constraints of the found solution will be a correct solution.

The algorithm is based on the least **commitment strategy.** This means that the algorithm will add the least number of ordering constraints so that to make the algorithm complete and consistent.

The algorithm adds the least number of ordering constraints because at each iteration when adds a step it adds the necessary ordering constraints only in order to make the current partial plan partially complete and consistent.

* Look at the example in the slides Part 1 of this folder

## Deep description of the algorithm

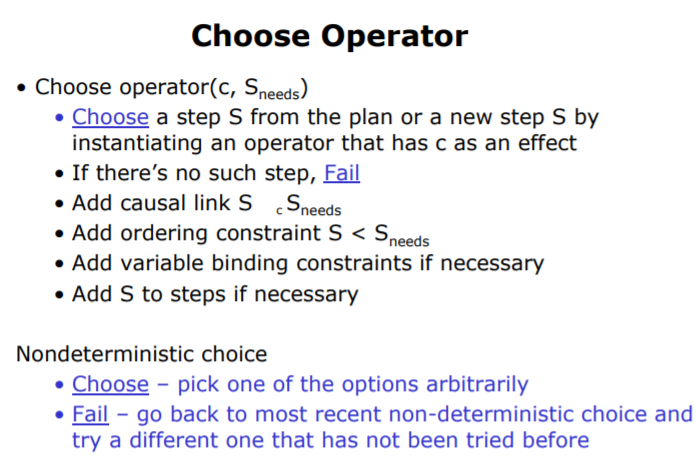


In other words, after creating the initial plan as previously described, the algorithm will loop (or call itself recursively) until the plan is complete and consistent. At each iteration, it selects a pre-condition c which has not been satisfied yet and tries to find an operator (step) S which satisfies c. After S has been added, eventual threats are resolved.

## Select subgoal

The phase of selecting a subgoal consists of selecting a precondition c which has not been satisfied yet. The precondition can be a whole pre-condition of a step or a subset of a pre-condition of a step.

## Choose an operator



This phase consists of selecting a step S from the plan or add a new one which satisfies the previously chosen pre-condition c. If such a step S does not exists and so no subset of a pre-condition c can be satisfied we need to backtrack to the most recent iteration of the algorithm trying adding another step rather than the one previously chosen.

If such a step S exists then we add a causal link S → cSneeds and the ordering constraint S < Sneeds and also Start < S and S < Finish.

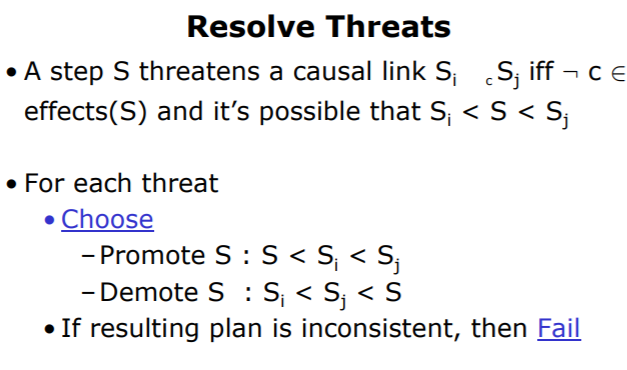
Eventual variable bindings are added and then S can be finally added to the plan.

Afterwards, eventual threats are resolved and then the algorithm will call itself recursively until the plan is complete.

## Resolve threats

There are two kinds of threats that can occur when adding a step S.

The first one is the following:



The first threat can occur is that the new added step S can occur (according to the current ordering constraints) between Si and Sj where there exists the following causal link Si → cSj and ⌐c is an effect of S. Thus, S cannot occur in between them.

There are two options to solve this threat:

* Promote S : Add an ordering constraint that S must occur before Si
* Demote S: Add an ordering constraint that S must occur after Sj

If neither of them is consistent with the current ordering constraint then we must backtrack because S cannot be added and so we try to add another step.

## Variable threats

Another threat that can occur is the fact that when adding step S some variables in the current partially ordered plan may be bound to some constants or other variables. This bounding can cause inconsistency in the variable bindings or a threat of the kind we have illustrated above.

* Look at the example in slides Part 2 in this folder