Module 3: Planning

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Learning Objectives

To exhibit intelligent (rather than random or else rigid) behavior, agents must plan.

By the end of this module, you will be able to observe states as the mode for a plan, distinguish the creation of a plan by beginning with the initial state, or else with a final desired state.

The ideas are applicable to programs, which are a kind of plan.

After successfully completing this module, you will be able to do the following:

- Identify actions applicable in states.
- Plan forward or backward.
- Apply to programming.

Module 3 Study Guide and Deliverables

Module Theme:	Planning; Uncertainty and Bayesian Reasoning
Readings:	• Module 3 online content • Russell & Norvig Chapter 11 (Automated Planning), concentrate on Section 11.1 • Russell & Norvig Chapter 12 (Quantifying Uncertainty), concentrate on Section 12.1
Assignments:	• Lab 3 due Sunday, September 26, at 6:00 AM ET • Assignment 3, due Wednesday, September 29, at 6:00 AM ET
Live Classrooms:	• Wednesday, September 22, from 8:00 PM to 9:00 PM ET • Thursday, September 23, from 8:00 PM to 9:00 PM ET • Live Office: Wednesday and Thursday after Live Classroom, for as long as there are questions

Introduction

In this section we discover what's involved in planning.

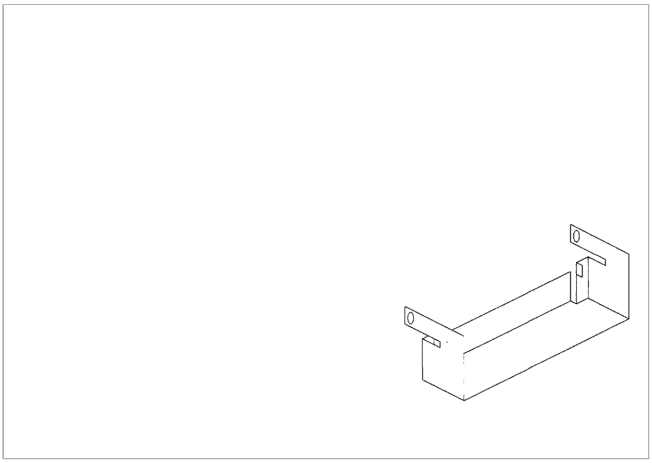
Example: Plan …

- a wedding
- a trip
- a project
- a move
    - between apartments
    - between houses
    - from one business location to another
- a robot's course of action

Robots, and agents in general, are much more effective if they have a plan rather that following a purely greedy algorithm.

Example: Outcome

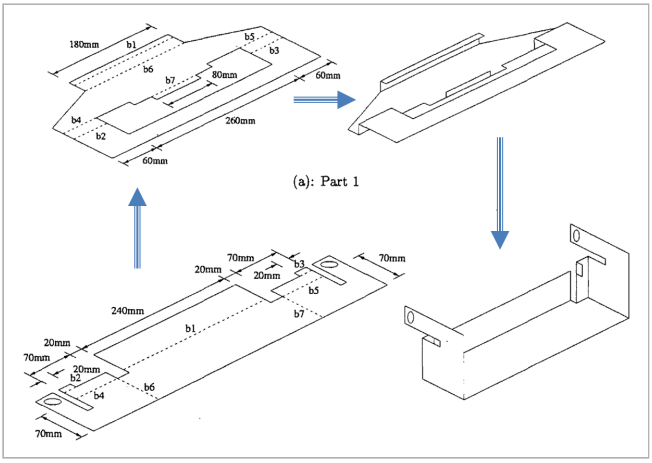
→



Consider the problem of finding a way to manufacture an item, such as the one shown made from a single flat piece of metal. This is ***backward*** planning.

The problem of assembling items (creating molecules, packing a box etc.) is similar.

Source: Gupta, S.K & Bourne, D.A. (1999). Shhet Metal Bending: Generating Shared Setups. Journal of Manufacturing Science and Engineering. November 1999, Vol. 121.



The figure shows a way to do this. The question is: can this plan be arrived at through an (AI planning) algorithm. The advantages include faster manufacturing, less waste, and labor saving.

Source: Gupta, S.K & Bourne, D.A. (1999). Shhet Metal Bending: Generating Shared Setups. Journal of Manufacturing Science and Engineering. November 1999, Vol. 121.

Planning is also required for the increasing autonomy of vehicles, just as humans exercise as much foresight as they can in operating them. This is especially true of extraterrestrial vehicles, which cannot be controlled in real time.

What is a Plan?

A ***plan***, formally speaking, is a sequence of actions that accomplish a goal, e.g., to get to NYC. Each action changes the state of the agent, agents, or system. The set of available actions depends on the statev (Obtain from action inventory indexed by state).

- the initial state
  - e.g., (agent is) in JP, Boston
  - or agent1 in … and agent2 in …
- the actions available in a state
- the result of applying an action
- the goal test
  - (agent is) in NYC?

States

In general, a state is defined by values of variables. For example, the state “Late for Work” is defined by \(t\gt{K}\) where \(t\) is the variable *time to get to work* and \(K\) is *minimum timeto get to work without being late* . In this example, the variable (\(t\)) is continuous.

Russell and Norvig simplify this by considering only a discrete variable (sometimes called ***state***)—that can only take a set of values from a finite set. These values are called ***fluents***, and are given names. An example is

\(state = {Poor, Unknown}\).

Each state is represented as a conjunction of **fluents**: ground, functionless atoms.

For example,

\(Poor \land{unknown}\)

\(In_JP\land{Have_car}\)

The simplified version of states are often expressed in planning as predicates, such as those in the example below. This is like a variable \(At\), whose values are pairs.

A state in a package delivery problem might be

\(At(Truck_1, Melbourne)\land{At(Truck_2, Sydney)}\)

Closed World Assumption (CWA)

In AI, we often choose to make the ***closed world assumption***—Anything not provable is assumed to be false. Think of it as ultra-science-based: assume that anything unprovable within a system is actually false within that system.

The CWA is by no means foolproof, as the following example suggests.

CWA: Anything not provable is assumed to be false.

Example: Plan a driving vacation between Denver and LA. If “steep roads are en route” can’t be established from the given environments, assume that there are none.

Action Schemas

So far, our discussion of planning has involved simple states (e.g., \(At(Truck, Melbourne)\)) and actions. This is addressed with ***schemas***&mdashd;set of actions.

Lifts level of reasoning from propositional logic to … first-order logic.

Schemas tell what the effects (the postconditions) are for an action—under what preconditions.

For example, an action schema for flying a plane from one location to another:

```
Action(
        Fly(p, from, to),
        PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
```

```
            EFFECT: ¬At(p, from) ∧ At(p, to)
    )
```

Plan Example: Spare Tire

Consider the goal \(At(Spare, Axle)\) (the spare tire is installed), given \(Tire(Flat)\) (the regular tire is flat) etc. What plan (sequence of schemas) accomplishes this?

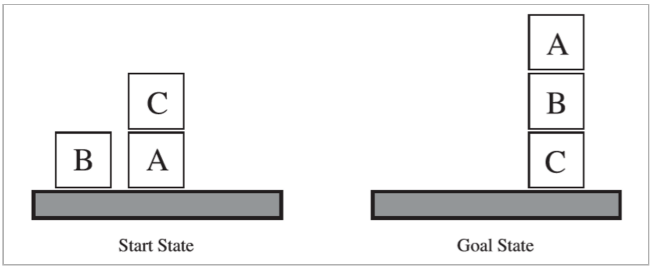
We show the beginning of the first schema of such as sequence: \(Action(PutOn(t, Axle)\) …

The sequence of actions is \(Remove(Tire, Axle)\), \(PutOn(Spare, Axle)\), \(LeaveOvernight\), and each is defined above with their pre- and postconditions. Notice that we can't assume any common sense: for example, the logic has to be explicit, in the last Action, that the Spare is *not* at specific locations—other than the axle.

\(Init(Tire(Flat)\land{Tire(Spare)}\land{At(Flat,Axle)}\land{At(Spare, Trunk)})\)
\(Goal(At(Spare,Axle))\)
\(Action(Remove(obj,loc)),\)
\(\quad{\text{PRECOND}: At(obj,loc)}\)
\(\quad{\text{EFFECT}: \lnot{At(obj,loc)}\land{At(obj,Ground)}}\)
\(Action(PutOn(t,Axle),\)
\(\quad{\text{PRECOND}: Tire(t)\land{At(t,Ground)}\land{\lnot{At(Flat,Axle)}}}\)
\(\quad{\text{EFFECT}: \lnot{At(t,Ground)}\land{At(t,Axle)}}\)
\(Action(LeaveOvernight,\)
\(\quad{\text{PRECOND}}:\)
\(\quad{\text{EFFECT}: \lnot{At(Spare,Ground)}\land{\lnot{At(Spare,Axle)}}\land{\lnot{At(Spare,Trunk)}}}\)
\(\qquad\qquad\qquad{\land{\lnot{At(Falt,Ground)}}\land{\lnot{At(Flat,Axle)}}\land{\lnot{At(Flat,Trunk)}}}\)

Plan Example: Famous Blocks World

Figure: A Planning Problem in the Block World



Source: Russell and Norvig

Let's see a planning problem in the blocks world: building a three-block tower.

This simple example shows the principles: such is the purpose of the blocks shown in the figure.

The code shows the totality of initial conditions.

$Init(On(A,Table)\land{On(B,Table)}\land{On(C,A)})$
$\quad\land\,{Block(A)}\land{Block(B)}\land{Block(C)}\land{Clear(B)}\land{Clear(B)}$
$Goal(On(A,B)\land{On(B,C)})$

One solution is the sequence $[MoveToTable(C,A),\,Move(B, Table, C),\, Move(A,Table, B)]$

The Action needed to accomplish the goal are instances of **Move**, as defined in the code.

$Init(On(A,Table)\land{On(B,Table)}\land{On(C,A)})$
$\quad\land\,{Block(A)}\land{Block(B)}\land{Block(C)}\land{Clear(B)}\land{Clear(B)}$
$Goal(On(A,B)\land{On(B,C)})$
$Action(Move(b,x,y),$
$\quad\text{PRECOND}: On(b,x)\land{Clear(y)}\land{Block(b)}\land{Block(y)}\land$
$\qquad\qquad (b\ne{x})\land{(b\ne{y})}\land{(x\ne{y})},$
$\quad\text{EFFECT}: On(b,y)\land{Clear(x)}\land\lnot{On(b,x)}\land\lnot{Clear(y)})$
$Action(MoveToTable(b,x),$
$\quad\text{PRECOND}: On(b,x)\land{Clear(b)}\land{Block(b)}\land{(b\ne{x})},$
$\quad\text{EFFECT}: On(b,Table)\land{Clear(x)}\land\lnot{On(b,x)})$

Planning Domain Definition Language (PDDL)

Practical Plan Example

There is cargo at San Francisco, which we want at JFK.

There is cargo at JFK, which we want at San Francisco.

We have a plane at San Francisco and one at JFK. Develop a plan (at the level of load/unload/fly).

This poses a practical planning example. It refers to the ***Planning Domain Definition Language (PDDL)***, which Russell and Norvig translate into logic notation below.

The code below specifies the initial conditions, the goal, and the first action of the plan.

$Init(At(C_1,SFO)\land{At(C_2,JFK)}\land{At(P_1,SFO)}\land{At(P_2,JFK)})$
$\qquad\land\,{Cargo(C_1)}\land{Cargo(C_2)}\land{Plane(P_1)}\land{Plane(P_2)}$
$\qquad\land\,{Airport(JFK)}\land{Airport(SFO)})$
$t(C_2,SFO)})$

\(Action(Load(c,p,a),\)
\(\quad\text{PRECOND}: At(c,a)\land{At(p,a)}\land{Cargo(c)}\land{Plane(p)}\land{Airport(a)}\)
\(\quad\text{EFFECT}: \lnot{At(c,a)}\land{In(c,p)})\)

Then the plan is completed below.

A PDDL Description of AN Air Cargo Transportation Planning Problem

\(Init(At(C_1,SFO)\land{At(C_2,JFK)}\land{At(P_1,SFO)}\land{At(P_2,JFK)}\)
\(\qquad\land\,{Cargo(C_1)}\land{Cargo(C_2)}\land{Plane(P_1)}\land{Plane(P_2)}\)
\(\qquad\land\,{Airport(JFK)}\land{Airport(SFO)})\)
\(Goal(At(C_1,JFK)\land{At(C_2,SFO)})\)
\(Action(Load(c,p,a),\)
\(\quad\text{PRECOND}: At(c,a)\land{At(p,a)}\land{Cargo(c)}\land{Plane(p)}\land{Airport(a)}\)
\(\quad\text{EFFECT}: \lnot{At(c,a)}\land{In(c,p)})\)
\(Action(Unload(c,p,a),\)
\(\quad\text{PRECOND}: In(c,p)\land{At(p,a)}\land{Cargo(c)}\land{Plane(p)}\land{Airport(a)}\)
\(\quad\text{EFFECT}: At(c,a)\land\lnot\,{In(c,p)})\)
\(Action(Fly(p,from,to),\)
\(\quad\text{PRECOND}: At(p,from)\land{Plane(p)}\land{Airport(from)}\land{Airport(to)}\)
\(\quad\text{EFFECT}: \lnot\,{At(p,from)}\land{At(p,to)})\)

Source: Russell and Norvig

PDDL

The Planning Domain Definition Language (PDDL) is an attempt to standardize Artificial Intelligence (AI) planning languages (Wikipedia).

PDDL can specify the following:

- the initial state
- the actions available in a state
- the result of applying an action
- the goal test

Source: https://github.com/pellierd/pddl4j/wiki/A-tutorial-to-start-with-PDDL

PDDL (Code) Examples

Check out the following list including a a PDDL demo, code base, and (essentially) a marketing example. The syntax of PDDL is rooted in LISP.

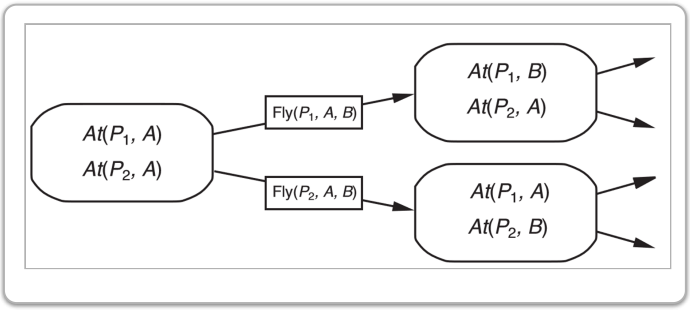
- SQUIRREL Demo (demo)
- ning Example (logistic application)

Forward vs. Backward Planning

In this section we compare the two principal poles of planning: forwards from the initial conditions, and backward from the goal.

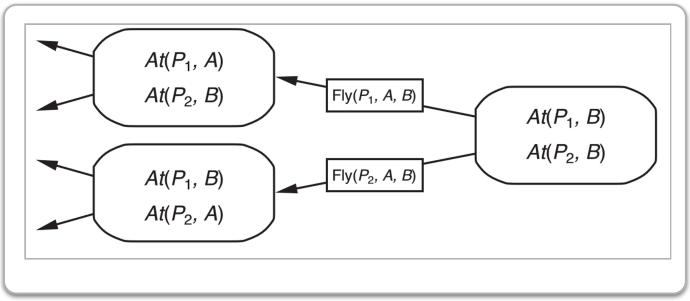
Forward Planning

In (pure, un-pruned) forward planning, we generate all possible permissible actions until the goal is encountered. A beginning is shown in the figure.



Backwards Planning

In backward planning, we begin with the goal, and generate all actions that result in that goal, noting their preconditions, which become a new set of goals on which to (recursively) apply the same process. In principle, this is performed until the given initial conditions are obtained.



Cumulative Goal Fulfillment

A computer program is analogous to a plan. In the ***Cumulative Goal Fulfillment*** approach to programming, we fulfill a set of goals \((CG_1, CG_2, \ldots {CG_n})\), which are at least as strong as the set of postconditions. We write code that fulfills the first goal; then code that fulfills the second, but which maintains the validity of the first etc.

We write code that fulfills the first goal; then code that fulfills the second, but which maintains the validity of the first

1. **Start with overall goals** $OG_1, OG_2, \ldots, OG_n$
2. **Identify a sufficient sequence of goals** $CG_1, CG_2, \ldots, CG_n$

   i.e., such that $(CG_1 \land CG_2 \land \ldots \land CG_n \Rightarrow OG_1 \land OG_2 \land \ldots \land OG_n)$

   which …

Often … $CG_1 = OG_1, CG_2 = OG_2, ldots., CG_n = OG_n$

Weakest Preconditions and Cumulative Subgoal Fulfillment, by Eric J. Braude, *Science of Computer Programming* Volume 89, Part C, 1 September 2014, pps. 223-234.

The property of fulfilling goals while maintaining prior ones is “cumulative.” Cumulative goals are helpful in being specific about the purpose of each block of code.

1. **Start with overall goals** $OG_1, OG_2, \ldots, OG_n$
2. **Identify a sufficient sequence of goals** $CG_1, CG_2, \ldots, CG_n$
   i.e., such that $(CG_1\land{CG_2}\land\ldots\land{CG_n}\Rightarrow{OG_1\land{OG_2}\land\ldots\land{OG_n}})$
   which are cumulative i.e.,
   fulfilling CG_i maintains $CG_1\land{CG_2}\land\ldots\land{CG_{i-1}}$

- Example: Towers of Hanoi: CG_1 = “??”
- Example: Repair flat: CG_1 = “??”

Weakest Preconditions and Cumulative Subgoal Fulfillment, by Eric J. Braude, *Science of Computer Programming* Volume 89, Part C, 1 September 2014, pps. 223-234.

There are techniques for identifying goals, and the code (the *actions* in planning lingo) that fulfills them.

But they are arrived at by practice and selected from a very wide set of legal programming constructs whereas in planning, the set of actions is very limited, and we can often use heuristics to identify potentially successful ones.

1. **Start with overall goals** $OG_1, OG_2, \ldots, OG_n$
2. **Identify a sufficient sequence of goals** $CG_1, CG_2, \ldots, CG_n$
   i.e., such that $(CG_1\land{CG_2}\land\ldots\land{CG_n}\Rightarrow{OG_1\land{OG_2}\land\ldots\land{OG_n}})$
   which are cumulative i.e.,
   fulfilling CG_i maintains $CG_1\land{CG_2}\land\ldots\land{CG_{i-1}}$

- Example: Towers of Hanoi: CG_1 = “Largest on target”
- Example: Repair flat: CG_1 = “New inner tube on rim”

Weakest Preconditions and Cumulative Subgoal Fulfillment, by Eric J. Braude, *Science of Computer Programming* Volume 89, Part C, 1 September 2014, pps. 223-234.

Examples

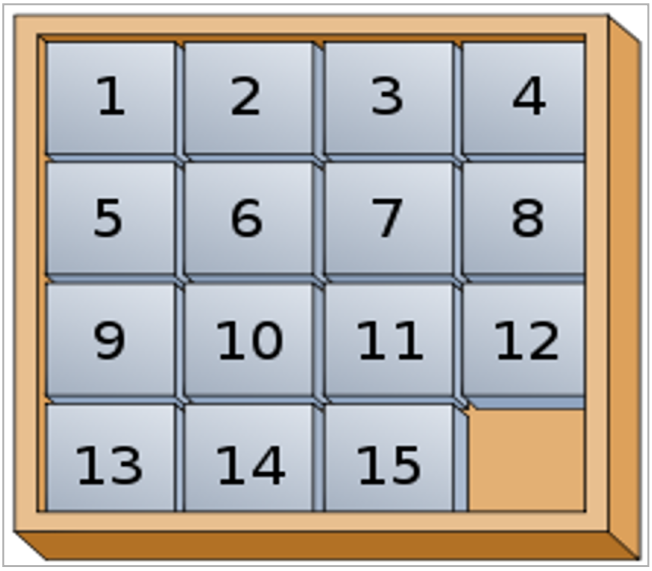
15-Puzzle Postcondition(s)

15-Puzzle

To solve the famous **Fifteen Puzzle**, for example, what would be a useful cumulative goal for the plan? What fluents or predicates?

A conjunction of fluents (maybe location_of_1, location_of_2, …)

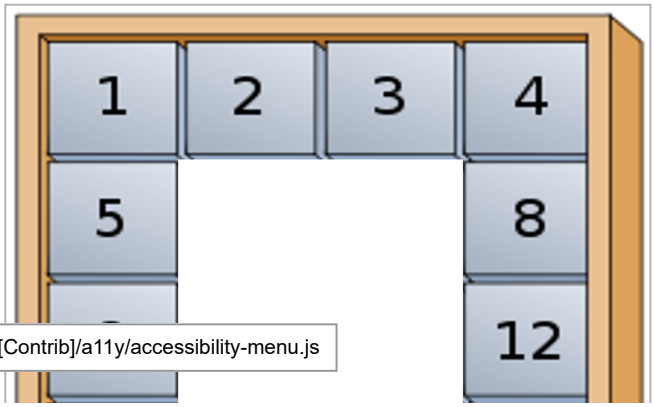
Which selection is useful?



15-Puzzle Cumulative Goal # 1 (A useful fluent to take as a goal)

A useful—and, it turns out—practical goal is to have the required outer ring in its required order.

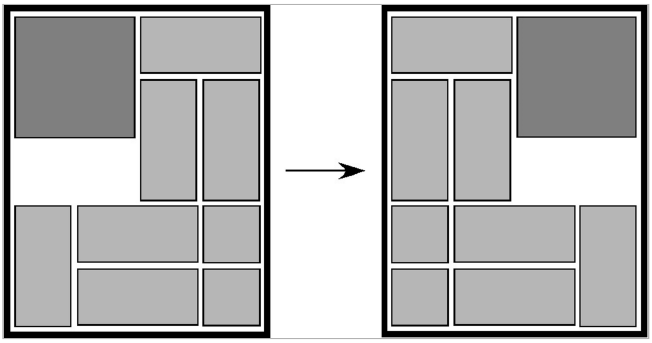
The point here is that there is apparently no limit on how imaginative useful goal fluents and predicates can be.



Dad's Puzzle

A more difficult problem is **Dad's Puzzle**, shown in the figure. Goal predicates can be identified such as **Square One Move From Destination**, but not very useful ones.

Unsolved … (non-final) Goal: Dad's Puzzle
from the start position shown on the left, slide pieces (without picking them up) to form the end position shown on the right.

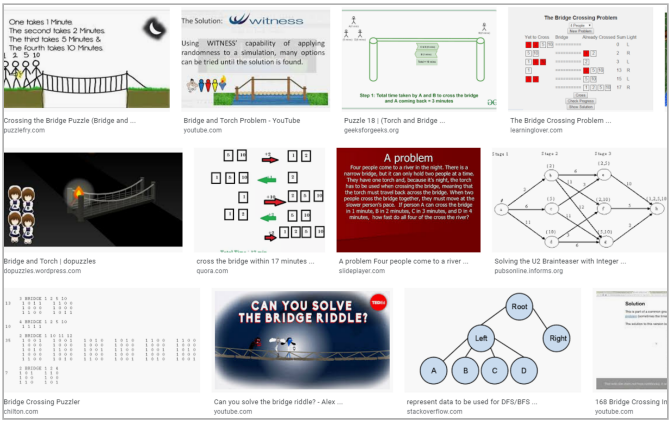


Source: https://www.youtube.com/watch?v=bzgYMIJQiE8

Bridge-Torch Problem

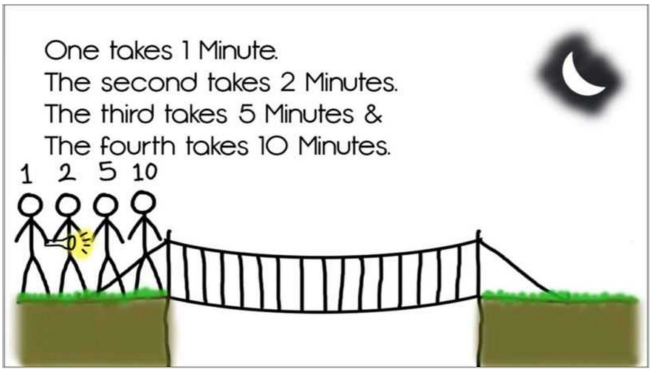
Puzzles provide a fertile ground for planning methods. An example is the **Bridge-Torch problem** in which a group of people, each with their own bridge-crossing time, must cross a bridge at night. The bridge can hold at most two people. A flashlight is necessary for each crossing. There is only one.

Bridge-Torch Problem



The figure below shows a particular instance of the problem.

Bridge-Torch Problem



Heuristics and Plan Graphs

In this section we consider how to use heuristics to develop a plan.

Heuristics to Prune the Tree

Viewing the possible courses of action as a tree of actions, the task of finding a good plan amounts to either generating all possible plans—which is typically impractical—or else pruning the tree using heuristics.

- Ignore preconditions
  – e.g., in 15-puzzle: ignore “must be a blank”
- Ignore “delete lists”
  – remove negative literals from effects
  – no action undoes progress made by another action
  – e.g., *Take car* does not remove *bus available*
- Abstract states
  – e.g., Instead of “Cargo at Atlanta Airport”, use “Cargo at Airport”
  – e.g., Instead of “Place dining table”, use “Sketch dining space”
- Decompose goal (artificially?)
  – e.g., At Waldorf Astoria \(\rightarrow\) At a NYC hotel \(\land\) On Park Avenue

The figure first lists two heuristics.

- **Ignore preconditions** loosens the circumstances of the problem, allowing for more flexibility in finding a plan. For the 15-puzzle, we could ignore the need to move only to a blank position. The idea is to develop a plan with this omission, swapping tiles, and then use the (illegal) result to create a legal plan.

- The **delete** lists heuristic is similar in that it ignores ill effects of actions.

  In everyday life, we plan like this all the time. For example, in planning a celebrations, we may first ignore preconditions such as venue availability, then use the resulting rough plan as the basis to move forward, modifying it as realistic venue availability is applied.

- **Abstract states** means to replace concrete courses of actions with general ones, e.g., replace *Select hotel* with *Select venue*. This makes it easier make progress with the plan. You

ulting rough plan.

- **Decompose goal** is similar in that it abstracts aspect of a goal as shown in the figure.

Managing Heuristics: Planning Graphs

We will discuss ***Planning Graphs***, which apply only to goals in simple propositional form (e.g., *not* of the form "every guest is familiar with the venue").

- Polynomial estimation of whole forward plan building
- Only for propositional planning problems
- Compare with A* (admissibility)

Plan Graph Example: Have your cake & …

Consider the problem with the initial condition, goal, and allowable actions shown in the figure. We'll generate a plan—a sequence of actions that, assuming *Init*, accomplish *Goal*.

$Init(Have(Cake))$
$Goal(Have(Cake)\land{Eaten(Cake)})$
$Action(Eat(Cake)$
$\quad\text{PRECOND}: Have(Cake)$
$\quad\text{EFFECT}: \lnot{Have(Cake)}\land{Eaten(Cake)})$
$Action(Bake(Cake)$
$\quad\text{PRECOND}: \lnot{Have(Cake)}$
$\quad\text{EFFECT}: Have(Cake))$

Source: Russell and Norvig

Basic *Planning Graph* Technique

To solve the problem, we think in terms of fluent sets that actions transform into new fluent sets. When a set of actions are such as to not affect each other, we note that because it makes matters easier. For example, the actions *selectVenue* and *selectWeddingDress* can be grouped.

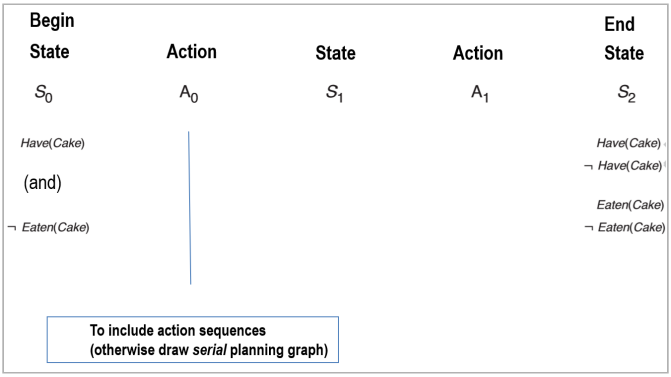
1. Organize as …

   Possible fluents → possible actions → new possible fluents

2. Group actions that can be taken in any order

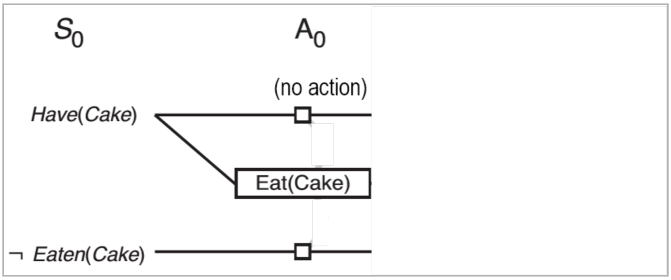
Planning Graph

The figure shows a framework for carrying this out.

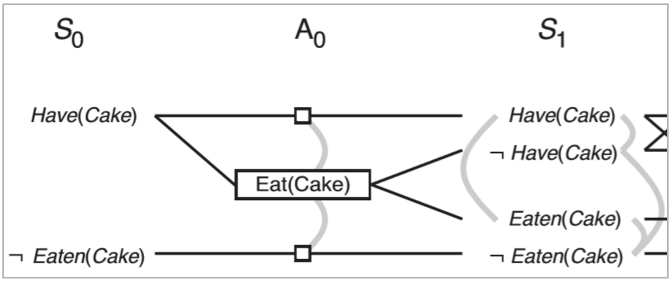


We then list all possible first actions (\(A_0\)) i.e., whose preconditions are included in the initial conditions. This would be *No action* (as we often say, doing nothing is a real choice) and \(Eat(Cake)\). *No action* activates on any precondition; \(Eat(Cake)\) is possible only when \(Have(Cake)\).

Note that \(Bake(Cake)\) is not an option because its precondition is not valid.



Now we collect all the possible states resulting from all possible actions, on all applicable precondition sets. The gray arcs at right (*mutex*'s) denote mutually exclusivity (you can't have both \(X\) and \(\lnot{X}\).



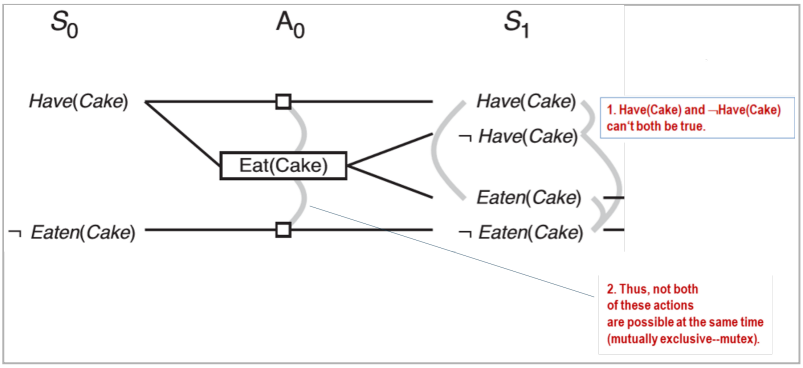
Note: \(Have(Cake)\) and \(\lnot{Have(Cake)} can't both be true\)

A mutex relation holds between two actions at a given level if any of:

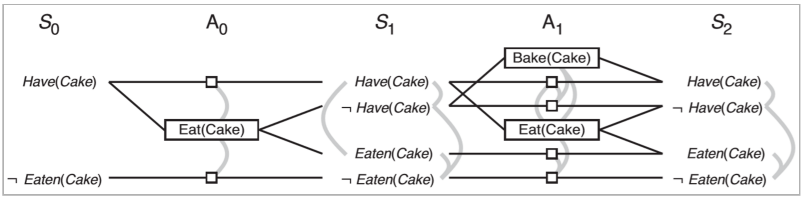
- **Inconsistent effects**
- **Interference:** one of the effects of one action is the negation of a precondition of the other. For example, \(Eat(Cake)\) interferes with the persistence of \(Have(Cake)\) by negating its precondition.

The process shows that *no action* and \(Eat(Cake)\) cannot be executed in any order here because their effects are not consistent (contain contradictions). Hence the first set of gray arcs (from the left).

In total, there are two possible resulting subsets—\(\{Have(Cake), \lnot{Eaten(Cake)}\}\), from executing *no action*—and \(\{\lnot{Have(Cake)}, Eaten(Cake)\}\), from executing \(Eat(Cake)\). These define the possible states \(S_1\).



We continue the process for the next possible actions. Instead of the preconditions, we consider the (2) possible states \(S_1\). This time, the action \(Bake(Cake)\) is an option as well.



A mutex relation holds between two actions at a given level if any of:

- **Inconsistent effects**
- **Interference:** one of the effects of one action is the negation of a precondition of the other. For example \(Eat(Cake)\) interferes with the persistence of \(Have(Cake)\) by negating its precondition.
- **Competing needs:** one of the preconditions of one action is mutually exclusive with a precondition of the other. For

example, \(Bake(Cake)\) and \(Eat(Cake)\) are mutex because they compete on the value of the \(Have(Cake)\) precondition.

Using a Planning Graph

- To estimate the number of actions required to get from a state to a goal. There are several variations (see p. 382, Russell and Norvig).
- To assess whether there is a plan that accomplishes desired goals.

Extracting Plan from Planning Graph

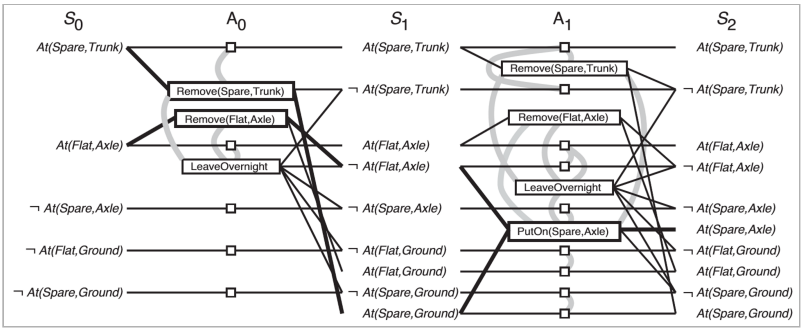
The pseudocode shown formalizes the extraction of plans using this approach.

```
function GRAPHPLAN(problem) returns solution or failure

   graph ← INITIAL-PLANNING-GRAPH(problem)
   goals ← CONJUNCTION(problem.GOAL)
   nogoods ← an empty hash table
   for tl=0 to ∞ do
       if goals all non-mutex in S_t of graph then
           solution &lefarrow; EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph),nogoods)
           if solution ≠ failure then return solution
        if graph and nogoods have both leveled off then return failure
        graph ← EXPAND-GRAPH(graph,problem)
```

Spare Tire Example

The figure shows the result of installing a spare tire (from Russell and Norvig). The process does not use any heuristics—which are needed because the exponential nature of its growth. However, many planning problems require a manageable number of steps, and may not require pruning.



Conclusion

Summary of Planning

- Specified by actions applicable in some states
- Forward and backward
- Serious pruning needed for forward

Boston University Metropolitan College