Lecture 10: Planning and Acting in the Real World
CS 580 (001) - Spring 2016

Amarda Shehu

Department of Computer Science
George Mason University, Fairfax, VA, USA

Apr 06, 2016
Outline of Today’s Class

2 Classical Planning
   ■ Search vs. Planning
   ■ STRIPS Operators
   ■ Planning Domain Definition Language (PDDL)
   ■ Forward (Progression) Planning: (Valid) State-space Search
   ■ Backward (Regression) Planning: Relevant-states Search
   ■ Heuristics for Efficient Forward and Backward Planning
     ■ Planning Graph
   ■ Classical Planning Summary: Complexity, Top-Performers
   ■ STRIPS Planning Algorithm and the Sussman Anomaly
   ■ Partial-order Planning

3 Planning and Acting in the Real World
   ■ Assumptions Invalidated in the Real World
   ■ Planning that Scales: Hierarchical Task Network Planning
   ■ Conformant Planning
   ■ Contingent/Conditional Planning
   ■ Online Planning: Monitoring and Replacing
Planning is the process of computing several steps of a problem-solving procedure before executing any of them.

This problem can be solved by search.

The main difference between search and planning is the representation of states.

In search, states are represented as a single/atomic entity (which may be quite a complex object, but its internal structure is not used by the search algorithm).

In planning, states have structured/factored representations (collections of properties/attributes) which are used by the planning algorithm.
Consider the task *get milk, bananas, and a cordless drill*

Standard search algorithms seem to fail miserably:

After-the-fact heuristic/goal test inadequate
Planning systems do the following:

1) open up action and goal representation to allow selection

2) divide-and-conquer by subgoaling

3) relax requirement for sequential construction of solutions

<table>
<thead>
<tr>
<th></th>
<th><strong>Search</strong></th>
<th><strong>Planning</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Lisp data structures</td>
<td>Logical sentences</td>
</tr>
<tr>
<td>Actions</td>
<td>Lisp code</td>
<td>Preconditions/outcomes</td>
</tr>
<tr>
<td>Goal</td>
<td>Lisp code</td>
<td>Logical sentence (conjunction)</td>
</tr>
<tr>
<td>Plan</td>
<td>Sequence from $S_0$</td>
<td>Constraints on actions</td>
</tr>
</tbody>
</table>
Classical Planning

- Atomic time: Each action is indivisible
- No concurrent actions allowed
- Deterministic actions: Result of each action is completely determined by the definition of the action, and there is no uncertainty in performing it in the world
- Agent is the sole cause of change in the world (environment is static and deterministic)
- Agent is omniscient – has complete knowledge of the state of the world (environment is fully-observable)
- **Closed World** assumption – everything known to be true in the world is included in a state description; what not listed is false
STRIPS planning language (Fikes and Nilsson, 1971)

Tidily arranged actions descriptions, restricted language

**Action:** Buy($x$)

**Precondition:** At($p$), Sells($p$, $x$)

**Effect:** Have($x$)

[Note: this abstracts away many important details!]

Restricted language $\Rightarrow$ efficient algorithm

Precondition: conjunction of **positive** literals

Effect: conjunction of literals

Complete set of STRIPS operators can be translated into set of successor-state axioms
Planning Domain Definition Language

A bit more relaxed than STRIPS

Preconditions and goals can contain **negative** literals

**Action:** Buy(x)  
**Precondition:** At(p), Sells(p, x)  
**Effect:** Have(x)

is called an **action schema**
Planning Domain

States are sets of **fluents** (ground, function-less atoms)

Fluents which are not mentioned are false (this is the **closed world** assumption)

\[ a \in \text{Actions}(s) \iff s \models \text{Precond}(a) \]

\[ \text{Result}(s, a) = (s - \text{Del}(a)) \cup \text{Add}(a) \]

where:
- Del(a) is the list of literals which appear **negatively** in the effect of a
- Add(a) is the list of **positive** literals in the effect of a
Action: Buy(x)
Precondition: At(p), Sells(p, x), Have(Money)
Effect: Have(x), ~Have(Money)

Del(Buy(Jaguar)) = Have(Money)
Add(Buy(Jaguar)) = Have(Jaguar)

If s = \{At(JDealer), Sells(JDealer, Jaguar), Blue(Sky), Have(Money)\}

Is Buy(Jaguar) ∈ Actions(s)? is it a valid action?
Yes: s |= Precond(a)
Example

Action: Buy(x)
Precondition: At(p), Sells(p, x), Have(Money)
Effect: Have(x), ¬Have(Money)

\[\text{Del}(\text{Buy}(\text{Jaguar})) = \text{Have}(\text{Money})\]
\[\text{Add}(\text{Buy}(\text{Jaguar})) = \text{Have}(\text{Jaguar})\]

If \( s = \{\text{At(JDealer)}, \text{Sells(JDealer, Jaguar)}, \text{Blue(Sky)}, \text{Have(Money)}\} \)

Is \( \text{Buy(Jaguar)} \in \text{Actions}(s) \)? is it a valid action?
Yes: \( s \models \text{Precond(a)} \)
Action: Buy(x)
Precondition: At(p), Sells(p, x), Have(Money)
Effect: Have(x), ¬Have(Money)

Del(Buy(Jaguar)) = Have(Money)
Add(Buy(Jaguar)) = Have(Jaguar)

If \( s = \{At(JDealer), Sells(JDealer, Jaguar), Blue(Sky), Have(Money)\} \)

Is Buy(Jaguar) ∈ Actions(s)?
Yes: \( s \models Precond(a) \)  

Result(s, Buy(Jaguar)) = (s - Have(Money)) \cup \{Have(Jaguar)\}  
= \{At(JDealer), Sells(JDealer, Jaguar), Blue(Sky), Have(Jaguar)\}  

is it a valid action? How?
Action: Buy(x)
Precondition: At(p), Sells(p, x), Have(Money)
Effect: Have(x), ¬Have(Money)

\[ \text{Del(Buy(Jaguar))} = \text{Have(Money)} \]
\[ \text{Add(Buy(Jaguar))} = \text{Have(Jaguar)} \]

If \( s = \{ \text{At(JDealer), Sells(JDealer, Jaguar), Blue(Sky), Have(Money)} \} \)

Is Buy(Jaguar) \( \in \) Actions(s)? is it a valid action?
Yes: \( s \models \text{Precond(a)} \) How?

Result(s, Buy(Jaguar)) = (s - Have(Money)) \( \cup \) \{Have(Jaguar)\}
= \{At(JDealer), Sells(JDealer, Jaguar), Blue(Sky), Have(Jaguar)\}
Planning problem = **planning domain** + **initial state** + **goal**

Goal is a **conjunction** of literals: Have(Jaguar) ∧ ¬ At(Jail)

Can solve planning problem using **search**

How?
Planning problem = planning domain + initial state + goal

Goal is a conjunction of literals: Have(Jaguar) ∧ ¬ At(Jail)

Can solve planning problem using search

How?
Forward Search or Backward Search

We mainly look at search as forward search: from initial to goal state

Nothing prevents us from searching from goal to initial state

Sometimes, branching factor makes searching backwards more reasonable

Motivating example: imagine trying to figure out how to get to some small place with few traffic connections from somewhere with a lot of traffic connections
Planning problem $= \text{planning domain} + \text{initial state} + \text{goal}$

Goal is a conjunction of literals: $\text{Have(Jaguar)} \land \neg \text{At(Jail)}$

Can solve planning problem using search

How? 

**Forward Search** or **Backward Search**

We mainly look at search as forward search: from initial to goal state

Nothing prevents us from searching from goal to initial state

Sometimes, branching factor makes searching backwards more reasonable

Motivating example: imagine trying to figure out how to get to some small place with few traffic connections from somewhere with a lot of traffic connections
Example: Forward Search Tree
Example: Backward Search Tree
Backward Search

Can use any search method, BFS, DFS, IDS, Dijkstra, A*, etc.

If there are several goal states, search backwards from each in turn

Planning can use both forward and backward search (progression and regression planning)
Example of Forward/Progression Planning

Planning domain:

Predicates: At, Sells, Have

Two action schemas:

Action: Buy(x)
Precondition: At(p), Sells(p, x), Have(Money)
Effect: Have(x), ¬Have(Money)

Action: Go(x, y)
Precondition: At(x)
Effect: At(y), ¬At(x)
Planning problem: planning domain above plus

Objects: Money, J (for Jaguar), Home, G (for garage)
Initial state: At(Home) ∧ Have(Money) ∧ Sells(G, J)
Goal state: Have(J)

Note: state descriptions are always ground (no variables).

Goal description may have variables: At(x) ∧ Have(y).

An atomic ground formula At(Home) is true iff it is in the state description.

A negation of a ground atom ¬At(G) is true iff the atom At(G) is not in the state description.

A property with a variable, such as At(x), is satisfied at a state if there is a way of substituting an object for x so that the resulting formula is true in the state.
Note:

At initial state:
  Go(Home, Home) is applicable/valid and does not change the state
  Buy(x) not available for any x (dont have Sells(Home, x))

At intermediate state:
  Go(Garage, Home) and Go(Garage, Garage) both applicable/valid
Backward (Regression) Planning: Relevant-states Search

Also called relevant-states search

Start at the goal state(s) and do regression (go back)

To be precise, we start with a ground goal description \( g \) which describes a set of states (all those where \( \text{Have}(J) \) holds but \( \text{Have}(\text{Money}) \) may or may not hold, for example)
Given a goal description $g$ and a ground action $a$, the regression from $g$ over $a$ gives a state description $g'$:

$$g' = (g - \text{Add}(a)) \cup \{\text{Precond}(a)\}$$

For example, if the goal is $\text{Have}(J)$ and ground action is $\text{Buy}(J)$:

$$g' = (\{\text{Have}(J)\} - \{\text{Have}(J)\}) \cup \{\text{At}(p), \text{Sells}(p, J), \text{Have(Money)}\}$$

$$= \{\text{At}(p), \text{Sells}(p, J), \text{Have(Money)}\}$$

Note that $g'$ is partially uninstantiated ($p$ is a free variable)

In this example, there is only one match for $p$, namely $G(\text{arage})$, but in general there may be several.
Which actions to regress over?

**Relevant** actions: have an effect which is in the set of goal elements and no effect which negates an element of the goal

For example, Buy(Jaguar) is a relevant action

In summary: backward planning searches backwards from g, remembering the actions and checking whether it reached an expression applicable to the initial state
Example of Backward/Regression Planning (continued)

Note:

- goal state matches initial state with y/Home and x/G
- intermediate state does not match initial state yet

Let \( g = \text{Have(Jaguar)} \land \neg \text{At(Jail)} \):

Buy(Jaguar) is a relevant action:

- (has an effect which is in \( g \) and no effect which negates an element of \( g \))

\[
g' = (\{ \text{Have(Jaguar)}, \neg \text{At(Jail)} \} - \{ \text{Have(Jaguar)} \}) \cup \\
\{ \text{At(p)}, \text{Sells(p, Jaguar)}, \text{Have(Money)} \} = \\
\{ \neg \text{At(Jail)}, \text{At(p)}, \text{Sells(p, Jaguar)}, \text{Have(Money)} \}
\]

If we had an extra action Steal(Jaguar), which also resulted in Have(Jaguar) but had an additional effect of At(Jail), Buy(Jaguar) would not be a relevant action.
Example of Backward/Regression Planning (continued)

Note:
- goal state matches initial state with y/Home and x/G
- intermediate state does not match initial state yet

Let $g = \text{Have}(\text{Jaguar}) \land \neg \text{At}(\text{Jail})$:

$\text{Buy}(\text{Jaguar})$ is a relevant action:
- (has an effect which is in $g$ and no effect which negates an element of $g$)

$$g' = (\{\text{Have}(\text{Jaguar}), \neg \text{At}(\text{Jail})\} - \{\text{Have}(\text{Jaguar})\}) \cup \{\text{At}(p), \text{Sells}(p, \text{Jaguar}), \text{Have(Money)}\} = \{\neg \text{At}(\text{Jail}), \text{At}(p), \text{Sells}(p, \text{Jaguar}), \text{Have(Money)}\}$$

If we had an extra action $\text{Steal}(\text{Jaguar})$, which also resulted in $\text{Have}(\text{Jaguar})$ but had an additional effect of $\text{At}(\text{Jail})$, $\text{Buy}(\text{Jaguar})$ would not be a relevant action
If there are lots of actions, searching for a solution starting from the initial state looks hopeless.
Heuristics to Tame Forward and Backward Planning

One can derive good heuristics (for A*) - two basic approaches to deriving heuristics:

1) relax the problem - effectively adding more edges to the graph
   Strategies: remove (some) preconditions, ignore delete lists, etc.

   Action: Slide(t, s1, s2)
   Precond: On(t, s1) ∧ Tile(t) ∧ Blank(s2) ∧ Adjacent(s1, s2)
   Effect: On(t, s2) ∧ Blank(s1) ∧ ¬On(t, s1) ∧ ¬Blank(s2)

   removing preconditions Blank(s2) ∧ Adjacent(s1, s2) \implies \text{number-of-misplaced-tiles heuristic}
   removing Blank(s2) allows tiles to move to occupied places \implies \text{Manhattan-distance heuristic}

2) abstract the problem (group nodes together, make the search space smaller)

   Backward planning considers a lot fewer actions/relevant states than forward search, but uses sets of states (g, g’) - harder to come up with good heuristics:
   \text{planning graph} can be used to derive better heuristics
Planning graphs are also used as a source of heuristics (an estimate of how many steps it takes to reach the goal)

Planning graph is an approximation of a complete tree of all possible actions and their results

Organized into **levels**:

Level S0: initial state, consisting of nodes representing each fluent that holds in S0
Level A0: each ground action that might be applicable in S0
Then alternate Si and Ai

Si contains fluents which could hold at time i, (may be both P and ¬P)
   literals may show up too early but never too late

Ai contains actions which could have their preconditions satisfied at i
Initial state: Have(Cake)

Goal: Have(Cake) \land Eaten(Cake)

Eat(Cake):
Precond: Have(Cake)
Effect: \neg Have(Cake) \land Eaten(Cake)

Bake(Cake):
Precond: \neg Have(Cake)
Effect: Have(Cake)
Incomplete Planning Graph

\[ S_0 \quad A_0 \]

- Have(Cake)
  - \neg\text{Have(Cake)}
  - \text{Eaten(Cake)}
- \neg\text{Eaten(Cake)}
- \text{Eat(Cake)}
  - \neg\text{Eat(Cake)}
  - \text{Have(Cake)}
In addition to 'normal' action, persistence action or no-op (no operation): one for each fluent, preserves the fluents truth

Mutex or mutual exclusion links depicted by red semicircles mean that actions cannot occur together

Similarly there are mutex links between fluents

Build the graph until two consecutive levels are identical until the graph levels off
Incomplete Planning Graph (continued)

\[ S_0 \quad A_0 \quad S_1 \]

- Have(Cake)
- no-op
- Eat(Cake)
- \neg Have(Cake)
- Eaten(Cake)
- \neg Eaten(Cake)

\neg Eaten(Cake) \quad \neg Eaten(Cake)
Mutex relation holds between two actions at the same level if any of the following three conditions holds:

◊ Inconsistent effects: one action negates an effect of another
  For example, Eat(Cake) and persistence for Have(Cake) have inconsistent effects (¬Have(Cake) and Have(Cake))

◊ 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, Eat(Cake) has precondition Have(Cake) and Bake(Cake) has precondition of ¬Have(Cake).
Mutex between Fluents

Mutex holds between fluents if:

◊ they are negations of each other, like Have(Cake) and \neg \text{Have(Cake)}

◊ each possible pair of actions that could achieve the two literals is mutually exclusive, for example Have(Cake) and Eaten(Cake) in S1 can only be achieved by persistence for Have(Cake) and by Eat(Cake) respectively. (In S2 can use persistence for Eaten(Cake) and Bake(Cake) which are not mutex).
**Complete Planning Graph (and Size)**

Size: is polynomial in the size of the problem (unlike a complete search tree, which is exponential!)

If we have \( n \) literals and \( a \) actions:
- each \( S_i \) has no more than \( n \) nodes and \( n^2 \) mutex links,
- each \( A_i \) has no more than \( a + n \) nodes (\( n \) because of no-ops), \( (a + n)^2 \) mutex links, and \( 2(an + n) \) precondition and effect links.

Hence, a graph with \( k \) levels has \( O(k(a + n)^2) \) size.
Using Planning Graph for Heuristic Estimation

If some goal literal does not appear in the final level of the graph, the goal is not achievable.

The cost of achieving any goal literal can be estimated by counting the number of levels before it appears.

This heuristic never overestimates.

It underestimates because planning graph allows application of actions (including incompatible actions) in parallel.

Conjunctive goals:
- **max level heuristic**: max level for any goal conjunct (admissible but inaccurate)
- **set level heuristic**: which level they all occur on without mutex links (better, also admissible)
GraphPlan repeatedly adds a level to a planning graph with Expand-Graph.

Once all the goals show up as non-mutex in the graph, calls Extract-Solution on the graph to search for a plan.

If that fails, extracts another level.

```
function GraphPlan(problem) returns solution or failure
    graph ← Initial-Planning-Graph(problem)
    goals ← Conjuncts(problem.Goal)
    loop do
        if goals all non-mutex in last level of graph then do
            solution ← Extract-Solution(graph,goals, NumLevels(graph)
            if solution ≠ failure then return solution
        else if No-solution-possible(graph) then return failure
        graph ← Expand-Graph(graph,problem)
    end loop

Solution can be then extracted via backward search.

Search may still degenerate to exponential exploration, but heuristics exist.
Example of a planning system using planning graphs for heuristics

FF or FastForward system (Hoffman 2005)

Forward space searcher

Ignore-delete-lists heuristic estimated using planning graph

Uses hill-climbing search with this heuristic to find solution

When hits a plateau or local maximum uses iterative deepening to find a better state or gives up and restarts hill-climbing
GraphPlan seeks optimal plan

SATPlan reduces planning problem to classical propositional SAT problem (section 10.4 in book)

SAT problem: is this propositional formula satisfiable? (is there an assignment that makes it true?)

Can only find plans of fixed maximal length

To use SATPlan, PDDL planning problem description needs first to be translated to a suitable form
PlanSAT is the question whether there exists any plan that solves a given planning problem.

Bounded PlanSAT is the question whether there exists a plan of length $k$ or less.

PlanSAT is about satisficing (want any solution, not necessarily the cheapest or the shortest).

Bounded PlanSAT can be used to ask for the optimal solution.

If in the PDDL language we do not allow functional symbols, both problems are decidable.

Complexity of both problems is PSPACE (can be solved by a Turing machine which uses polynomial amount of space).

NP is a subset of PSPACE (PSPACE is even harder than NP).
<table>
<thead>
<tr>
<th>Year</th>
<th>Track</th>
<th>Winning systems (approaches)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>Optimal</td>
<td>Gamer (symbolic bi-directional search)</td>
</tr>
<tr>
<td>2008</td>
<td>Satisficing</td>
<td>LAMA (fast forward search with FF heuristic)</td>
</tr>
<tr>
<td>2006</td>
<td>Optimal</td>
<td>SATPlan, MAXPlan (boolean satisfiability)</td>
</tr>
<tr>
<td>2006</td>
<td>Satisficing</td>
<td>SGPlan (forward search, partition into independent subproblems)</td>
</tr>
<tr>
<td>2004</td>
<td>Optimal</td>
<td>SATPlan (boolean satisfiability)</td>
</tr>
<tr>
<td>2004</td>
<td>Satisficing</td>
<td>Fast Diagonally Forward (forward search with causal graph)</td>
</tr>
<tr>
<td>2002</td>
<td>Automated</td>
<td>LPG (local search, constraint satisfaction)</td>
</tr>
<tr>
<td>2002</td>
<td>Hand-coded</td>
<td>TLPLan (temporal action logic with control rules for forward search)</td>
</tr>
<tr>
<td>2000</td>
<td>Automated</td>
<td>FF (forward search)</td>
</tr>
<tr>
<td>2000</td>
<td>Hand-coded</td>
<td>TalPlanner (temporal action logic with control rules for forward search)</td>
</tr>
<tr>
<td>1998</td>
<td>Automated</td>
<td>IPP (planning graphs); HSP (forward search)</td>
</tr>
</tbody>
</table>
Classical planning algorithms typically do not scale well

Should subgoals be reached serially?

What about leveraging goal decomposition?

Does order matter?

Next: Partial-order Planning
Classical planning algorithms typically do not scale well

Should subgoals be reached serially?

What about leveraging goal decomposition?

Does order matter?

Next: Partial-order Planning

Then: Partial-order planning that scales: hierarchical task network Search may still degenerate to exponential exploration (HTN) planning
Classical planning algorithms typically do not scale well

Should subgoals be reached serially?

What about leveraging goal decomposition?

Does order matter?

Next: Partial-order Planning

Then: Partial-order planning that scales: hierarchical task network Search may still degenerate to exponential exploration (HTN) planning

But first: an instructive exercise
Classical planning algorithms typically do not scale well

Should subgoals be reached serially?

What about leveraging goal decomposition?

Does order matter?

Next: Partial-order Planning

Then: Partial-order planning that scales: hierarchical task network Search may still degenerate to exponential exploration (HTN) planning

But first: an instructive exercise
The domain consists of:

1. a table, a set of cubic blocks, and a robot arm

2. each block is either on the table or stacked on top of another block

3. the arm can pick up a block and move it to another position either on the table or on top of another block

4. the arm can only pick up one block at a time, so it cannot pick up a block which has another block on top

A goal is a request to build one or more stacks of blocks specified in terms of which blocks are on top of which other blocks.
State Descriptions

Blocks are represented by constants A, B, C . . . etc.

An additional constant Table represents the table.

The following predicates are used to describe states:

On(b, x): block b is on x, where x is either another block or the table.

Clear(x): there is a clear space on x to hold a block.
MOVE(b, x, y):
Precond: On(b, x) ∧ Clear(b) ∧ Clear(y)
Effect: On(b, y) ∧ Clear(x) ∧
        ¬On(b, x) ∧ ¬Clear(y)

MOVE-TO-TABLE(b, x):
Precond: On(b, x) ∧ Clear(b)
Effect: On(b, Table) ∧ Clear(x) ∧
        ¬On(b, x)
There is nothing to stop the planner using the 'wrong' operator to put a block on the table, i.e. MOVE(b, x, Table) rather than MOVE-TO-TABLE(b, x)

which results in a larger-than-necessary search space, and

Some of the operator applications, such as MOVE(B,C,C), which should do nothing, have inconsistent effects

To solve the first one, we could introduce predicate Block and make Block(x) and Block(y) preconditions of MOVE(x, y, z)

To solve the second we could add \( \neg(y = z) \) as a precondition for MOVE(x, y, z)
Example Problem: Sussman Anomaly

Initial state:
On(C,A) ∧ OnTable(A) ∧ OnTable(B) ∧ Clear(B) ∧ Clear(C)

Goal state:
On(A,B) ∧ On(B,C) ∧ OnTable(C)
**Goal Stack Planning:** One of the earlier planning algorithms, used by STRIPS

Work backwards from the goal, looking for an operator which has one or more of the goal literals as one of its effects and then trying to satisfy the preconditions of the operator.

The preconditions of the operator become subgoals that must be satisfied.

Keep doing this until initial state is reached.

Goal stack planning uses a stack to hold goals and actions to satisfy the goals, and a knowledge base to hold the current state, action schemas and domain axioms.

Goal stack is like a node in a search tree.

If there is a choice of action, create branches.
Goal Stack Planning Pseudocode

Push the original goal on the stack.

Repeat until the stack is empty:

If stack top is a compound goal, push its unsatisfied subgoals on the stack.

If stack top is a single unsatisfied goal, replace it by an action that makes it satisfied and push the actions precondition on the stack.

If stack top is an action, pop it from the stack, execute it and change the knowledge base by the action’s effects.

If stack top is a satisfied goal, pop it from the stack.
(Below does pushing of subgoals at the same step as the compound goal.)  
The order of subgoals is arbitrary; could have put $\text{On}(B,C)$ on top

$\text{On}(A,B)$
$\text{On}(B,C)$
$\text{OnTable}(C)$
$\text{On}(A,B) \land \text{On}(B,C) \land \text{OnTable}(C)$

$\text{KB} = \{\text{On}(C,A), \text{OnTable}(A), \text{OnTable}(B), \text{Clear}(B), \text{Clear}(C)\}$

$\text{plan} = [ \ ]$

The top of the stack is a single unsatisfied goal. So push the action that would achieve it, and its preconditions.
Clear(B)
Clear(A)
On(A, x)
Clear(A) ∧ Clear(x)
MOVE(A, x, B)
On(A,B)
On(B,C)
OnTable(C)
On(A,B) ∧ On(B,C) ∧ OnTable(C)

KB = \{On(C,A), OnTable(A), OnTable(B), Clear(B), Clear(C)\}

plan = [ ]

The top of the stack is a satisfied goal. We pop the stack.
Clear(A)
On(A, x)
Clear(A) ∧ Clear(x)
MOVE(A, x, B)
On(A,B)
On(B,C)
OnTable(C)
On(A,B) ∧ On(B,C) ∧ OnTable(C)

KB = \{On(C,A), OnTable(A), OnTable(B), Clear(B), Clear(C)\}

plan = [ ]

The top of the stack is an unsatisfied goal. We push the action which would achieve it, and its preconditions.
On(C, A)
Clear(C)
On(C, A) ∧ Clear(C)
MOVE-TO-TABLE(C, A)
Clear(A)
On(A, x)
Clear(A) ∧ Clear(x)
MOVE(A, x, B)
On(A,B)
On(B,C)
OnTable(C)
On(A,B) ∧ On(B,C) ∧ OnTable(C)

KB = \{ On(C,A), OnTable(A), OnTable(B), Clear(B), Clear(C) \}

plan = [ ]
The top of the stack is a satisfied goal. We pop the stack (three times).
MOVE-TO-TABLE(C, A)
Clear(A)
On(A, x)
Clear(A) \land Clear(x)
MOVE(A, x, B)
On(A,B)
On(B,C)
OnTable(C)
On(A,B) \land On(B,C) \land OnTable(C)

KB = \{On(C,A),OnTable(A),OnTable(B),Clear(B),Clear(C)\}

plan = []

The top of the stack is an action. We execute it, update the KB with its effects, and add it to the plan.
Clear(A)
On(A, Table)
Clear(A) \land Clear(x)
MOVE(A, x, B)
On(A,B)
On(B,C)
OnTable(C)
On(A,B) \land On(B,C) \land OnTable(C)

KB = \{ OnTable(C), OnTable(A), OnTable(B), Clear(A), Clear(B), Clear(C) \}
plan = [ MOVE-TO-TABLE(C,A) ]

The top of the stack is a satisfied goal. We pop the stack (thrice).
MOVE(A, x, B)  
On(A,B)  
On(B,C)  
OnTable(C)  
On(A,B) ∧ On(B,C) ∧ OnTable(C)  

KB = \{OnTable(C), OnTable(A), OnTable(B), Clear(A), Clear(B), Clear(C)\}  

plan = [MOVE-TO-TABLE(C,A)]  

The top of the stack is an action. We execute it, update the KB with its effects, and add it to the plan.
On(A,B)  
On(B,C) 
OnTable(C) 
On(A,B) \land On(B,C) \land OnTable(C) 

KB = \{On(A,B), OnTable(C), OnTable(B), Clear(A), Clear(C)\} 

plan = [MOVE-TO-TABLE(C,A), MOVE(A, Table,B)] 

the current state is:

```
 A
 B
 C
```
If we follow the same process for the $\text{On}(B,C)$ goal, we end up in the state:

\[
\text{On}(B,C) \land \text{OnTable}(A) \land \text{OnTable}(C)
\]

The remaining goal on the stack $\text{On}(A,B) \land \text{On}(B,C) \land \text{OnTable}(C)$ is not satisfied.

So $\text{On}(A,B)$ will be pushed on the stack again!
Now we finally can move A on top of B, but the resulting plan is redundant:

MOVE-TO-TABLE(C,A)
MOVE(A, Table,B)
MOVE-TO-TABLE(A,B)
MOVE(B, Table,C)
MOVE(A, Table,B)

There are techniques for 'fixing' inefficient plans (where something is done and then undone), but it is difficult in general (when it is not straight one after another)
It seemed to make sense to break up a conjunctive goal into subgoals and achieve them, separately, in some order.

Sussman anomaly is instructive, because achieving one goal (On(A,B)) destroys preconditions of an action which is necessary to achieve the other goal (On(B,C)), namely Clear(B).

Such interaction between actions is called clobbering.

This is addressed in partial order planning.
Totally vs. Partially-ordered Plans

So far we produced a linear sequence of actions (totally ordered plan)

Often it does not matter in which order some of the actions are executed

For problems with independent subproblems, often easier to find a partially-ordered plan: a plan which is a set of actions and a set of constraints Before(ai, aj)

Partially ordered plans are created by a search through a space of plans (rather than the state space)
**Partially-ordered** collection of steps with:

- **Start step** has the initial state description as its effect
- **Finish step** has the goal description as its precondition
- **Causal links** from outcome of one step to precondition of another
- **Temporal ordering** between pairs of steps

**Open condition** = precondition of a step not yet causally linked

A plan is **complete** iff every precondition is achieved

A precondition is **achieved** iff it is the effect of an earlier step and no **possibly intervening** step undoes it
Example

At(Home)  Sells(HWS,Drill)  Sells(SM,Milk)  Sells(SM,Ban)

Have(Milk)  At(Home)  Have(Ban.)  Have( Drill)

Finish
Example

```
Start
At(Home)  Sells(HWS,Drill)  Sells(SM,Milk)  Sells(SM,Ban.)

At(HWS)  Sells(HWS,Drill)
Buy(Drill)

At(x)  
Go(SM)

At(SM)  Sells(SM,Milk)
Buy(Milk)

Have(Milk)  At(Home)  Have(Ban.)  Have(Drill)
Finish
```
Example

![Diagram showing a classical planning problem]

Diagram showing a classical planning problem with states and actions including:
- Start
- At(Home)
- Go(HWS)
- At(HWS)
- Sells(HWS, Drill)
- Buy(Drill)
- At(HWS)
- Go(SM)
- At(SM)
- Sells(SM, Milk)
- Buy(Milk)
- At(SM)
- Sells(SM, Ban.)
- Buy(Ban.)
- At(SM)
- Go(Home)
- Have(Milk)
- At(Home)
- Have(Ban.)
- Have(Home)
- Finish
Operators on partial plans:

- **add a link** from an existing action to an open condition
- **add a step** to fulfill an open condition
- **order** one step with respect to another to remove possible conflicts

Gradually move from incomplete/vague plans to complete, correct plans.

Backtrack if an open condition is unachievable or if a conflict is unresolvable.
function POP(initial, goal, operators) returns plan

plan ← MAKE-MINIMAL-PLAN(initial, goal)
loop do
    if SOLUTION?(plan) then return plan
    S_{need}, c ← SELECT-SUBGOAL(plan)
    CHOOSE-OPERATOR(plan, operators, S_{need}, c)
    RESOLVE-THREATS(plan)
end

function SELECT-SUBGOAL(plan) returns S_{need}, c

pick a plan step S_{need} from STEPS(plan)
    with a precondition c that has not been achieved
return S_{need}, c
procedure \textsc{Choose-Operator}(plan, operators, S_{need}, c)

choose a step $S_{add}$ from operators or \textsc{Steps}(plan) that has $c$ as an effect

if there is no such step then fail
add the causal link $S_{add} \xrightarrow{c} S_{need}$ to \textsc{Links}(plan)
add the ordering constraint $S_{add} \prec S_{need}$ to \textsc{Orderings}(plan)

if $S_{add}$ is a newly added step from operators then
add $S_{add}$ to \textsc{Steps}(plan)
add $Start \prec S_{add} \prec Finish$ to \textsc{Orderings}(plan)


procedure \textsc{Resolve-Threats}(plan)

for each $S_{threat}$ that threatens a link $S_i \xrightarrow{c} S_j$ in \textsc{Links}(plan) do
choose either

Demotion: Add $S_{threat} \prec S_i$ to \textsc{Orderings}(plan)
Promotion: Add $S_j \prec S_{threat}$ to \textsc{Orderings}(plan)

if not \textsc{Consistent}(plan) then fail
end
A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. E.g., \texttt{Go(Home)} clobbers \texttt{At(Supermarket)}:

**Demotion:** put before \texttt{Go(Supermarket)}

**Promotion:** put after \texttt{Buy(Milk)}
Properties of POP

Nondeterministic algorithm: backtracks at choice points on failure:

– choice of $S_{add}$ to achieve $S_{need}$

– choice of demotion or promotion for clobberer

– selection of $S_{need}$ is irrevocable

POP is sound, complete, and systematic (no repetition)

Extensions for disjunction, universals, negation, conditionals

Can be made efficient with good heuristics derived from problem description

Particularly good for problems with many loosely related subgoals
Example: Blocks World

"Sussman anomaly" problem

Start State

Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)

Clear(z) On(x,y)

Goal State

Clear(x) On(x,z)

PutOnTable(x)

~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints
Example Continued

\[\text{On}(C,A) \ \text{On}(A,\text{Table}) \ \text{Cl}(B) \ \text{On}(B,\text{Table}) \ \text{Cl}(C)\]

\[\text{On}(A,B) \ \text{On}(B,C)\]

\[\text{START}\]

\[\text{FINISH}\]
Example Continued

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

Cl(B) On(B,z) Cl(C)

PutOn(B,C)

On(A,B) On(B,C)

FINISH
Example Continued

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

Amarda Shehu (580) Classical Planning 70
Example Continued

START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

On(C,z) Cl(C)

PutOnTable(C)

Cl(A) On(A,z) Cl(B)

PutOn(A,B)

Cl(B) On(B,z) Cl(C)

PutOn(B,C)

On(A,B), On(B,C)

FINISH

PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

PutOn(B,C) clobbers Cl(C) => order after PutOnTable(C)
Planning in the Real World

Planning in the presence of limited resources:
Actions consume resources, time being one of them
Specialized scheduling algorithms or algorithms that integrate scheduling with planning address time

Semi-autonomous Planning:
Hierarchical task network planning (HTN) allows planning agent to incorporate advice from domain expert in the form of high-level actions (HLAs)

Assumptions of classic planning often violated in the real world:
Contingent plans allow agent to sense in a partially-observed environment
Online planning agent can address nondeterministic actions, exogeneous events, or incorrect models of the environment
Multiagent planning allows cooperation or competition with other agents in the environment

Markov decision processes and game theory allow agent to plan in stochastic environments
Partial-order planning does not scale well in the real world.

Some planning tasks involve millions of actions.

Example: planning to invade a country.

There is, however, hierarchical structure that humans employ when putting together plans.

HTN planning: plan at a higher level, then refine if necessary.

Important concept: high-level actions (HLAs).

More in class, on the board.
The Real World: Planning and Acting in Nondeterministic Domains

\[
\begin{align*}
\text{START} & : \sim\text{Flat(Spare)} \land \text{Intact(Spare)} \land \text{Off(Spare)} \\
& \land \text{On(Tire1)} \land \text{Flat(Tire1)}
\end{align*}
\]

\[
\begin{align*}
\text{On(x)} & \land \sim\text{Flat(x)}
\end{align*}
\]

\[
\begin{align*}
\text{FINISH} & : \\
\text{Remove(x)} & : \sim\text{Flat(x)} \\
\text{Puton(x)} & : \text{On(x)} \land \sim\text{ClearHub} \\
\text{Inflate(x)} & : \text{Flat(x)}
\end{align*}
\]
Things Go Wrong

**Incomplete information**
Unknown preconditions, e.g., $\text{Intact}(\text{Spare})$?
Disjunctive effects, e.g., $\text{Inflate}(x)$ causes
$\text{Inflated}(x) \lor \text{SlowHiss}(x) \lor \text{Burst}(x) \lor \text{BrokenPump} \lor \ldots$

**Incorrect information**
Current state incorrect, e.g., spare NOT intact
Missing/incorrect postconditions in operators

**Qualification problem:**
can never finish listing all the required preconditions and possible conditional outcomes of actions
Conformant or sensorless planning
Devise a plan that works regardless of state or outcome
*Sach plans may not exist*

Conditional planning
Plan to obtain information (observation actions)
Subplan for each contingency, e.g.,
\[ \text{Check}(Tire1), \text{if } \text{Intact}(Tire1) \text{ then } \text{Inflate}(Tire1) \text{ else } \text{CallAAA} \]

*Expensive because it plans for many unlikely cases*

Monitoring/Replanning
Assume normal states, outcomes
Check progress *during execution*, replan if necessary
*Unanticipated outcomes may lead to failure (e.g., no AAA card)*

(Really need a combination; plan for likely/serious eventualities, deal with others when they arise, as they must eventually)
Conformant Planning

Search in space of belief states (sets of possible actual states)
If the world is nondeterministic or partially observable
then percepts usually provide information,
i.e., split up the belief state.
Conditional plans check (any consequence of KB +) percept
[... , if C then Plan_A else Plan_B, ...]

Execution: check C against current KB, execute “then” or “else”

Need some plan for every possible percept
(Cf. game playing: some response for every opponent move)
(Cf. backward chaining: some rule such that every premise satisfied

AND–OR tree search (very similar to backward chaining algorithm)
**AND-OR-GRAPH-SEARCH for Conditional Planning**

```plaintext
function `AND-OR-GRAPH-SEARCH`(*problem*) returns a conditional plan, or failure
    `OR-SEARCH`(problem.Initial-State, problem, [])

function `OR-SEARCH`(*state*, problem, *path*) returns a conditional plan, or failure
    if problem.Goal-Test(state) then return the empty plan
    if state is on path then return failure
    for each action in problem.Actions(state) do
        plan ← `AND-SEARCH`(Results(state, action), problem, [state | path])
        if plan≠ failure then return [action | plan]
    return failure

function `AND-SEARCH`(*states*, problem, *path*) returns a conditional plan, or failure
    for each $s_i$ in states do
        plan$_i$ ← `OR-SEARCH`($s_i$, problem, path)
        if plan$_i$ = failure then return failure
    return [if $s_1$ then plan$_1$ else if $s_2$ then plan$_2$ else ... if $s_{n-1}$ then plan$_{n-1}$ else plan$_n$]
```
Nondeterministic Vacuum Cleaner

Double Murphy: sucking or arriving may dirty a clean square
Nondeterministic Vacuum Cleaner

Triple Murphy: also sometimes stays put instead of moving

\[ L_1 : \text{Left, if AtR then } L_1 \text{ else [if CleanL then [] else Suck]} \]

or \[ \text{while AtR do [Left, if CleanL then [] else Suck]} \]

“Infinite loop” but will eventually work unless action always fails
“Failure” = preconditions of *remaining plan* not met

Preconditions of remaining plan
= all preconditions of remaining steps not achieved by remaining steps
= all causal links *crossing* current time point

On failure, resume POP to achieve open conditions from current state

IPEM (Integrated Planning, Execution, and Monitoring):
keep updating *Start* to match current state
links from actions replaced by links from *Start* when done
Example

```
Start
  ↓
  At(Home)

Go(HWS)
  ↓
  At(HWS)  Sells(HWS,Drill)

Buy(Drill)
  ↓
  At(HWS)

Go(SM)
  ↓
  At(SM)  Sells(SM,Milk)

Buy(Milk)
  ↓
  At(SM)

Go(Home)
  ↓
  Have(Milk)  At(Home)  Have(Ban.)  Have(Drill)

Finish
```
Example
Example
Emergent Behavior

**Preconditions**

START

\[ \text{Color}(Chair, Blue) \land \neg \text{Have}(Red) \]

Get(Red)

\[ \text{Have}(Red) \]

Paint(Red)

\[ \text{Color}(Chair, Red) \]

**Failure Response**

Fetch more red
“Loop until success” behavior emerges from interaction between monitor/replan agent design and uncooperative environment.
“Loop until success” behavior \textit{emerges} from interaction between monitor/replan agent design and uncooperative environment.