Chapter 11
Planning

Dr. Daisy Tang

IT WAS ALREADY TOO LATE WHEN JIMMY REALIZED HE HAD FORGOTTEN TO CONVERT HIS UNITS BACK INTO THE METRIC SYSTEM.
Real-World Problems

- **Planning**: the task of coming up with a sequence of actions that will achieve a goal
  - Search-based problem-solving agent
  - Logical planning agent
  - Complex/large scale problems?

- For the discussion, we consider **classical planning environments** that are fully observable, deterministic, finite, static and discrete (in time, action, objects and effects)
Problems with Standard Search

- Overwhelmed by irrelevant actions
- Finding a good heuristic function is difficult
- Cannot take advantage of problem decomposition
Example

Consider the task: get milk, bananas, and a cordless drill
- Standard search algorithms seem to fail miserably
- Why? Huge branching factor & heuristics
Problem Decomposition

- Perfectly decomposable problems are delicious but rare
  - Partial-order planner is based on the assumption that most real-world problems are nearly decomposable

- Be careful, working on some subgoal may undo another subgoal
Planning vs. Problem Solving

- Planning agent is very similar to problem solving agent
  - Constructs plans to achieve goals, then executes them

- Planning agent is different from problem solving agent in:
  - Representation of goals, states, actions
  - Use of explicit, logical representations
  - Way it searches for solutions
Planning vs. Problem Solving

Planning systems do the following:
- divide-and-conquer
- relax requirement for sequential construction of solutions

<table>
<thead>
<tr>
<th>Problem Sol.</th>
<th>Planning</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>data structures</td>
</tr>
<tr>
<td>Actions</td>
<td>code</td>
</tr>
<tr>
<td>Goal</td>
<td>code</td>
</tr>
<tr>
<td>Plan</td>
<td>sequence from $s_0$</td>
</tr>
</tbody>
</table>
Famous Problem Solver Task: “Missionaries and Cannibals”

(3, 3, 1) 

X (2, 3, 0) (3, 2, 0) X (1, 3, 0) (3, 1, 0) (2, 2, 0) 

(3, 2, 1) X (2, 3, 1) 

(2, 2, 0) X (1, 2, 0) (3, 0, 0) (2, 1, 0) X 

(3, 1, 1) 

(1, 1, 0) 

(2, 2, 1) 

etc. (0, 0, 0)
Planning-Based Approach to Robot Control

- **Job of planner**: generate a goal to achieve, and then construct a plan to achieve it from the current state

- Must define representations of:
  - **Actions**: generate successor state descriptions by defining preconditions and effects
  - **States**: data structure describing current situation
  - **Goals**: what is to be achieved
  - **Plans**: solution is a sequence of actions
Many AI Planners in History

Well-known AI Planners:
- STRIPS (Fikes and Nilsson, 1971): theorem-proving system
- ABSTRIPS (Sacerdoti, 1974): added hierarchy of abstractions
- HACKER (Sussman, 1975): use library of procedures to plan
- NOAH (Sacerdoti, 1975): problem decomposition and plan reordering
STRIPS-Based Approach to Robot Control

- Use first-order logic and theorem proving to plan strategies from start to goal

- STRIPS language:
  - “Classical” approach that most planners use
  - Lends itself to efficient planning algorithms

- Environment: office environment with specially colored and shaped objects

- STRIPS planner: developed for this system to determine the actions of the robot should take to achieve goals

- Cost of Shakey: $100, 000

Shakey (SRI), 1960’s
STRIPS

- STRIPS (Stanford Research Institute Problem Solver)
  - a restrictive way to express states, actions and goals, but leads to more efficiency

- States: conjunctions of ground, function-free, and positive literals, such as At(Home) ^ Have(Banana)
  - Closed-world assumption is used

- Goals: conjunctions of literals, may contain variables (existential), hence goal may represent more than one state
  - E.g. At(Home) ^ ¬ Have(Bananas)
  - E.g. At(x) ^ Sells(x, Bananas)

- Actions: preconditions that must hold before execution and the effects after execution
An action schema includes:
- action name & parameter list (variables)
- precondition: a conjunction of function-free positive literals. Any variables in it must also appear in parameter list
- effect: a conjunction of function-free literals (positive or negative)
  - add-list: positive literals
  - delete-list: negative literals

Example:
Action: Buy \((x)\)
Precondition: At \((p)\), Sells \((p, x)\)
Effect: Have\((x)\)
An Example of STRIPS World Model

- A world model is a set of facts
- The robot’s knowledge can be represented by the following predicates:
  - \( \text{INROOM}(x, r) \), where \( x \) is a movable object, \( r \) is a room
  - \( \text{NEXTTO}(x, t) \)
  - \( \text{STATUS}(d, s) \), where \( d \) is a door, \( s \) means OPEN or CLOSED
  - \( \text{CONNECTS}(d, rx, ry) \)

- The world model in this figure can be represented with the above predicates, with the initial state and goal state
Shakey’s STRIPS World

- Types of actions Shakey can make
  - Move from place to place:
    - $\text{GOTODOOR}(S, dx)$:
      - PRECOND: $\text{INROOM}(S, rk) \land \text{CONNECT}(dx, rk, rm)$
      - Effect:
        - add-list: $\text{NEXTTO}(S, dx)$
        - delete-list: null
    - $\text{GOTHRUDOOR}(S, dx)$:
      - PRECOND: $\text{CONNECT}(dx, rk, rm) \land \text{NEXTTO}(S, dx) \land \text{STATUS}(dx, \text{OPEN}) \land \text{INROOM}(S, rk)$
      - Effect:
        - add-list: $\text{INROOM}(S, rm)$
        - delete-list: $\text{INROOM}(S, rk)$
Simple Example of STRIPS-Style Planning

- Goal state: ON(A, B)
- Start state: ON(A, Table); ON(B, Table); EMPTYTOP(A); EMPTYTOP(B)
- Operators:
  - Move(x, y)
    - Preconditions: ?
    - Add-list: ?
    - Delete-list: ?

Start State

- Preconditions: ON(x, Table); EMPTYTOP(y)
- Add-list: ON(x, y)
- Delete-list: EMPTYTOP(y); ON(x, Table)

Goal State
Challenges of AI and Planning

- **Closed world assumption**: assumes that world model contains everything the robot needs to know: there can be no surprise.

- **Frame problem**: how to represent real-world situations in a manner that is computationally tractable.
Planning with State-Space Search

Planning algorithms:
- The most straightforward approach is to use state-space search
- Forward state-space search (Progression)
- Backward state-space search (Regression)
Problem Formulation for Progression

- Initial state:
  - Initial state of the planning problem

- Actions:
  - Applicable to the current state (actions’ preconditions are satisfied)

- Goal test:
  - Whether the state satisfies the goal of the planning

- Step cost:
  - Each action is 1
Progression

- A plan is a sequence of STRIPS operators
- From initial state, search forward by selecting operators whose preconditions can be unified with literals in the state
- New state includes positive literals of effect; the negated literals of effect are deleted
- Search forward until goal unifies with resulting state
- This is state-space search using STRIPS operators
Regression

- A plan is a sequence of STRIPS operators
- The goal state must unify with at least one of the positive literals in the operator’s effect
- Its preconditions must hold in the previous situation, and these become subgoals which might be satisfied by the initial conditions
- Perform backward chaining from goal
- Again, this is just state-space search using STRIPS operators
STRIPS Program To Control Shakey

- Work on board
Heuristics for State-Space Search

How to find an admissible heuristic estimate?
- Distance from a state to the goal?
- Look at the effects of the actions and at the goals and guess how many actions are needed
- NP-hard

Relaxed problem
Subgoal independence assumption:
- The cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal independently
Relaxation Problem

- **Idea:** removing all preconditions from the actions
  - *Almost* implies that the number of steps required to solve a conjunction of goals is the number of unsatisfied goals
    - There may be two actions, each of which deletes the goal literal achieved by the other
    - Or, some action may achieve multiple goals
  - Combining relaxation with subgoal independence assumption → exact # of unsatisfied goals
Removing Negative Effects

- Generate a relaxed problem by removing negative effects:
  - Empty-delete-list heuristic
  - Quite accurate, but computing it involves running a simple planning algorithm
Extensions – ADL

- **State**: positive and negative literals
- **Open world assumptions**: unmentioned literals are unknown
- **Effect**: \( P \land \neg Q \) means add \( P \) and \( \neg Q \) and delete \( \neg P \) and \( Q \)
- **Goal**: quantified variables; conjunction and disjunction
- **Effects**: conditional effects
  - when \( P: E \) means \( E \) is an effect only if \( P \) is satisfied
- **Equality predicate** \((x = y)\)
- Variables can have **types** \((p: \text{Plane})\)
Example: Air Cargo Transport

- Loading/unloading cargo onto/off planes and flying it from place to place.
- Actions: Load, Unload, and Fly
- States:
  - $\text{In}(c, p)$: cargo $c$ is inside plane $p$
  - $\text{At}(x, a)$: object $x$ (plane/cargo) at airport $a$


\[
\begin{align*}
\text{Init:} & \quad \text{At}(C_1, SFO) \land \text{At}(C_2, JFK) \land \text{At}(P_1, SFO) \land \text{At}(P_2, JFK) \\
& \quad \land \text{Cargo}(C_1) \land \text{Cargo}(C_2) \land \text{Plane}(P_1) \land \text{Plane}(P_2) \\
& \quad \land \text{Airport}(JFK) \land \text{Airport}(SFO))
\end{align*}
\]

\[
\begin{align*}
\text{Goal:} & \quad \text{At}(C_1, JFK) \land \text{At}(C_2, SFO))
\end{align*}
\]

\[
\begin{align*}
\text{Action}(\text{Load}(c, p, a), \\
& \quad \text{PRECOND:} \quad \text{At}(c, a) \land \text{At}(p, a) \land \text{Cargo}(c) \land \text{Plane}(p) \land \text{Airport}(a) \\
& \quad \text{EFFECT:} \quad \neg \text{At}(c, a) \land \text{In}(c, p))
\end{align*}
\]

\[
\begin{align*}
\text{Action}(\text{Unload}(c, p, a), \\
& \quad \text{PRECOND:} \quad \text{In}(c, p) \land \text{At}(p, a) \land \text{Cargo}(c) \land \text{Plane}(p) \land \text{Airport}(a) \\
& \quad \text{EFFECT:} \quad \text{At}(c, a) \land \neg \text{In}(c, p))
\end{align*}
\]

\[
\begin{align*}
\text{Action}(\text{Fly}(p, \text{from}, \text{to}), \\
& \quad \text{PRECOND:} \quad \text{At}(p, \text{from}) \land \text{Plane}(p) \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to}) \\
& \quad \text{EFFECT:} \quad \neg \text{At}(p, \text{from}) \land \text{At}(p, \text{to}))
\end{align*}
\]
Example: Spare Tire Problem

- **Initial state**: a flat tire on the axle and a good spare tire in the trunk
- **Goal state**: have a good spare tire properly mounted onto the car’s axle

```
Init(At(Flat, Axle) \land At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(Spare, Trunk),
  PRECOND: At(Spare, Trunk)
  EFFECT: \neg At(Spare, Trunk) \land At(Spare, Ground))
Action(Remove(Flat, Axle),
  PRECOND: At(Flat, Axle)
  EFFECT: \neg At(Flat, Axle) \land At(Flat, Ground))
Action(PutOn(Spare, Axle),
  PRECOND: At(Spare, Ground) \land \neg At(Flat, Axle)
  EFFECT: \neg At(Spare, Ground) \land At(Spare, Axle))
Action(LeaveOvernight,
  PRECOND: 
  EFFECT: \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)
  \land \neg At(Flat, Ground) \land \neg At(Flat, Axle))
```
Total-Order Planning

- Forward/backward state-space searches are forms of totally ordered plan search
  - explore only strictly linear sequences of actions directly connected to the start or goal
  - cannot take advantages of problem decomposition
Partial-Order Planning

- **Idea:**
  - works on several subgoals independently
  - solves them with subplans
  - combines the subplans
  - flexibility in ordering the subplans
- **least commitment strategy:**
  - delaying a choice during search
  - Example, leave actions unordered, unless they must be sequential
POP Example

- Putting on a pair of shoes:
  - Goal\((\text{RightShoeOn} \land \text{LeftShoeOn})\)
  - Init()
  - Action: RightShoe
    - PRECOND: RightSockOn
    - EFFECT: RightShoeOn
  - Action: RightSock
    - PRECOND: None
    - EFFECT: RightSockOn
  - Action: LeftShoe
    - PRECOND: LeftSockOn
    - EFFECT: LeftShoeOn
  - Action: LeftSock
    - PRECOND: None
    - EFFECT: LeftSockOn
POP Example

Start

LeftShoeOn, RightShoeOn

Finish

Left Sock

LeftSockOn

Left Shoe

LeftShoeOn, RightShoeOn

Finish

Right Sock

RightSockOn

Right Shoe

Finish
Partial Order Plan to Total Order Plan

Partial Order Plan:

- Start
- Left Sock → LeftSockOn → Left Shoe → LeftShoeOn, RightShoeOn → Finish
- Right Sock → RightSockOn → Right Shoe

Total Order Plans:

- Start → Right Sock → Left Sock → RightSockOn → LeftSockOn → Left Shoe → LeftShoeOn, RightShoeOn → Finish
- Start → Right Sock → Left Sock → RightSockOn → LeftSockOn → Right Shoe
- Start → Right Sock → Left Sock → RightSockOn → RightSockOn → Left Shoe → LeftShoeOn, RightShoeOn → Finish
- Start → Right Sock → Left Sock → RightSockOn → RightSockOn → Right Shoe
- Start → Right Sock → Left Sock → RightSockOn → RightSockOn → Left Shoe → LeftShoeOn, RightShoeOn → Finish
- Start → Right Sock → Left Sock → RightSockOn → RightSockOn → Right Shoe
- Start → Right Sock → Left Sock → RightSockOn → RightSockOn → Left Shoe → LeftShoeOn, RightShoeOn → Finish
- Start → Right Sock → Left Sock → RightSockOn → RightSockOn → Right Shoe
How to Define Partial Order Plan?

- A set of actions that make up the steps of the plan
- A set of ordering constraints: \( A \prec B \)
  - A before B
- A set of causal links: \( A \xrightarrow{p} B \)
  - A achieves p for B
  - \( RightSock \xrightarrow{RightSockOn} RightShoe \)
  - May be conflicts if C has the effect of \( \neg p \) and if C comes after A and before B
- A set of open preconditions:
  - a precondition is open if it is not achieved by some action in the plan
The Initial Plan

- Initial plan contains:
  - **Start**:
    - PRECOND: none
    - EFFECT: Add all propositions that are initially true
  - **Finish**:
    - PRECOND: Goal state
    - EFFECT: none

- Ordering constraints: $Start \prec Finish$

- Causal links: {}

- Open preconditions:
  
  \{preconditions of Finish\}
Next...

- **Successor function**
  - arbitrarily picks one open precondition $p$ on an action $B$ and generates a successor plan for every possible consistent way of choosing an action $A$ that achieves $p$

- **Consistency:**
  - Causal link $A \xrightarrow{p} B$ and the ordering constraint are added ($A \prec B$  \text{Start} \prec A  A \prec \text{Finish}$)
  - Resolve conflict: add $B \prec C$ or $C \prec A$

- **Goal test:**
  - There are no open preconditions
Example: Final Plan

- The final plan has the following components:
  - **Actions**: \{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish\}
  - **Orderings**: \{RightSock ≺ RightShoe, LeftSock ≺ LeftShoe\}
  - **Open preconditions**: \{\}
  - **Links**:

    \[
    \begin{align*}
    \text{RightSock} &\xrightarrow{\text{RightSockOn}} \text{RightShoe} \\
    \text{LeftSock} &\xrightarrow{\text{LeftSockOn}} \text{LeftShoe} \\
    \text{RightShoe} &\xrightarrow{\text{RightShoeOn}} \text{Finish} \\
    \text{LeftShoe} &\xrightarrow{\text{LeftShoeOn}} \text{Finish}
    \end{align*}
    \]
Example Algorithm for POP

- POP: A sound, complete partial order planner using STRIPS representation

```plaintext
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
```

where:
- c is a precondition of a step $S_{need}$
- RESOLVE-THREATS: orders steps as needed to ensure intermediate steps don’t undo preconditions needed by other steps
POP Example: Flat Tire

\[
\text{Init(At(Flat, Axle) \land At(Spare, Trunk))}
\]
\[
\text{Goal(At(Spare, Axle))}
\]
\[
\text{Action(}\text{Remove(Spare, Trunk)},
\]
\quad \text{PRECOND: } \text{At(Spare, Trunk)}
\]
\quad \text{EFFECT: } \neg \text{At(Spare, Trunk)} \land \text{At(Spare, Ground)}
\]
\[
\text{Action(}\text{Remove(Flat, Axle)},
\]
\quad \text{PRECOND: } \text{At(Flat, Axle)}
\]
\quad \text{EFFECT: } \neg \text{At(Flat, Axle)} \land \text{At(Flat, Ground)}
\]
\[
\text{Action(}\text{PutOn(Spare, Axle)},
\]
\quad \text{PRECOND: } \text{At(Spare, Ground)} \land \neg \text{At(Flat, Axle)}
\]
\quad \text{EFFECT: } \neg \text{At(Spare, Ground)} \land \text{At(Spare, Axle)}
\]
\[
\text{Action(}\text{LeaveOvernight},
\]
\quad \text{PRECOND: }
\]
\quad \text{EFFECT: } \neg \text{At(Spare, Ground)} \land \neg \text{At(Spare, Axle)} \land \neg \text{At(Spare, Trunk)}
\]
\quad \land \neg \text{At(Flat, Ground)} \land \neg \text{At(Flat, Axle)}
\]
Strengths of POP

- The causal links lead to **early pruning** of portions of the search space because of irresolvable conflicts

- The solution is a partial-order plan
  - linearizations produce **flexible** plans
From Problem Solvers to Planners

Key ideas:

- "Open up" representation of states, goals, and actions
  - Use descriptions in a formal language – FOL
  - States/goals represented by sets of sentences
  - Actions represented by logical description of preconditions and effects
  - Enables planner to make direct connections between states and actions

- Planner can add actions to plan whenever needed, rather than in strictly incremental fashion
  - No necessary connection between order of planning and order of execution

- Most parts of the world are independent of most other parts of the world
  - Conjunctions can be separated and handled independently
  - Divide-and-conquer algorithms
Summary of Planning Problem

- **Planning agents** use look-ahead to find actions to contribute to goal achievement.
- Planning agents differ from problem solvers in their use of more flexible representation of states, actions, goals, and plans.
- The **STRIPS language** describes actions in terms of preconditions and effects.
- Principle of **least commitment** is preferred.
- POP is a **sound and complete** algorithm for planning using STRIPS representation.
Exercise 1

- The monkey-and-bananas problem is faced by a monkey in a lab with some bananas hanging out of reach from the ceiling. A box is available that will enable the monkey to reach the bananas if he climbs on it.

- Initially the monkey is at A, bananas at B, the box at C. The monkey and box have height Low, but if the monkey climbs onto the box he will have height High, the same as the bananas.
Exercise 1 Cont’d

- **Actions:**
  - Go from one place to another
  - Push an object from one place to another
  - ClimbUp onto or ClimbDown from an object
  - Grasp or Ungrasp an object
  - The results of a Grasp is that the monkey holds the object if the monkey and object are in the same place at the same height.

- **Questions**
  - Write down the initial state description
  - Write down the goal state (monkey has banana at location D on the ground)
  - Write the six action schemas
  - Generate the plan
Exercise 2 & 3:

- Describe the differences and similarities between problem solving and planning.

- Explain why dropping negative effects from every action schema in a planning problem results in a relaxed problem.
Planning Domains

- **Classical planning**: fully observable, static, and deterministic; actions are correct and complete

- **Incompleteness**: partially observable, nondeterministic, or both

- **Incorrectness**: the world does not necessarily match my model of the world
Bounded/Unbounded Indeterminacy

- **Bounded indeterminacy**: actions have unpredictable effects, but the possible effects can be listed in the action description axioms

- **Unbounded indeterminacy**: the set of possible preconditions or effects either is unknown or is too large to be enumerated completely
Four Planning Methods For Handling Indeterminacy

- **Sensorless planning**: constructs sequential plans to be executed without perception.

- **Conditional planning** (contingency planning): constructs a conditional plan with different branches for different contingencies that could happen.

- **Execution monitoring and replanning**: uses preceding techniques to construct a plan, but it monitors the execution process and replan when necessary.

- **Continuous planning**: a planner designed to persist over a lifetime.
Example

Scenario:
- given an initial state with a chair, a table, and some cans of paint, with everything of unknown color, achieve the state where the chair and table have the same color

- classical planning
- sensorless planning
- conditional planning
- replanning
Planning With Partial Information

- We are familiar with: Deterministic, fully observable $\rightarrow$ single-state problem
  - agent knows exactly which state it will be in
  - solution is a sequence

- Deterministic, non-observable $\rightarrow$ multi-state problem
  - Also called sensorless problems (conformant problems)
  - agent may have no idea where it is
  - solution is a sequence
The Coloring Problem

- **Initial state:**
  - Object(Table) ^ Object(Chair) ^ Can(C1) ^ Can(C2) ^ InView(Table)

- **Goal state:**
  - Color(Chair, c) ^ Color(Table, c)

- **Actions:**
  - RemoveLid
  - Paint
  - LookAt
Sensorless Planning

- RemoveLid(Can1), Paint(Chair, Can1), Paint(Table, Can1)
Contingent planning

- LookAt(Table), LookAt(Chair)
- If Color(Table, c) \^\ Color(Chair, c) then NoOp
- Else RemoveLid(Can1), LookAt(Can1), RemoveLid(Can2), LookAt(Can2)
- If Color(Table, c) \^\ (Color(can, c), then Paint (Chair, can)
- If Color(Chair, c) \^\ (Color(can, c), then Paint(Table, can)
- Else Paint(Chair, Can1), Paint(Table, Can1)
Replanning

- LookAt(Table), LookAt(Chair)
- If Color(Table, c) ^ Color(Chair, c) then NoOp
- Else RemoveLid(Can1), LookAt(Can1)
- If Color(Table, c) ^ Color(Can1, c) then Paint(Chair, Can1)
- Else replan
Example: Vacuum World

- **Single-state**, start in #5. **Solution?**
  - [Right, Suck]

- **Multi-state**, start in #1, 2, ..., 8. **Solution?**
  - [Right, Suck, Left, Suck]
Sensorless (multi-state problem)
Conditional Planning in Fully Observable Environments

- Full observability means that the agent always knows the current state.

- Non-deterministic, because the agent cannot predict the outcome of its actions.

- The agent is able to check the state of the environment to decide what to do next.

- Revisit our vacuum world example
  - Let $\text{AtL}$ ($\text{AtR}$) be true if the agent is in the left (right) square.
  - Let $\text{CleanL}$ ($\text{CleanR}$) be true if the left (right) square is clean.
Disjunctive, Conditional Effects

- To augment the STRIPS language to allow for nondeterminism, we allow actions to have **disjunctive effects** and **conditional effects**

- **Conditional effect:** \[ \text{when } <\text{condition}> : <\text{effect}> \]

  \[
  \text{Action}(\text{Left}, \text{PRECOND} : \text{AtR}, \text{EFFECT} : \text{AtL} \lor \text{AtR})
  \]

  \[
  \text{Action}(\text{Suck}, \text{PRECOND} :, \text{EFFECT} : (\text{when AtL : CleanL}) \land (\text{when AtR : CleanR}))
  \]

  A vacuum cleaner that may dump dirt on the destination square when it moves, but only if that square is clean

  \[
  \text{Action}(\text{Left}, \text{PRECOND} : \text{AtR}, \text{EFFECT} : \text{AtL} \lor (\text{AtL} \land \text{when CleanL : } \neg \text{CleanL}))
  \]
Conditional Plans

- To create conditional plans, we need conditional steps: \[ \text{if } <test> \text{ then } plan_A \text{ else } plan_B \]
  - For example: \[ \text{if } AtL \land CleanL \text{ then } Right \text{ else } Suck \]
  - By nesting conditional steps, plans become trees
  - In general, conditional plan should work \textit{regardless of which action outcomes actually occur} -- games against nature

Consider the following vacuum cleaner:

The initial state has the robot in the right square of a clean world. The environment is fully observable. The goal state has the robot in the left square of a clean world.

It sometimes deposits dirt when it moves to a clean destination square and sometimes deposits dirt if Suck is applied to a clean square.
Example: Double-Murphy Cleaner

\[ \text{[Left, if CleanL then [ ] else Suck]} \]
Solution

- A solution is a subtree that
  - has a goal node at every leaf
  - specifies one action at each of its “state” nodes
  - and includes every outcome branch at each of its “chance” nodes
AND-OR Graph

- For conditional planning, we modify the minimax algorithm
  - MAX and MIN nodes become OR and AND nodes
    - the plan needs to take some action at every state, but must handle every outcome for the action it takes
  - The algorithm needs to return a conditional plan rather than just a single move
    - at an OR node, the plan is just the action selected
    - at an AND node, the plan is a nested series of if-then-else steps specifying subplans for each outcome
And-Or-Graph Search Algorithm

The algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state. --- noncyclic solution
Example: Triple-Murphy

Vacuum cleaner sometimes fails to move when commanded

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

cyclic solution, less desirable than non-cyclic solution
Conditional Planning in Partially Observable Environments

- In real world, partial observability is more common.

- The initial state belongs to a state set, which is also called belief state.

- Assume a vacuum agent knows that it is in the right-hand square and that square is clean, but it cannot sense the dirt in other squares.

- An alternate “double-murphy” world:
  - dirt can sometimes be left behind when the agent leaves a clean square.
AND-OR Graph of Belief States
Belief States

- The agent’s **belief state is always fully observable**
- Standard fully observable problem solving is just a special case in which every belief state is a singleton set containing one physical state
- Three basic choices for belief states:
  - Sets of full state descriptions \{ (AtR \land CleanR \land CleanL), (AtR \land CleanR \land \neg CleanL) \} 
    - Problems: large belief states
  - Logical sentences that capture exactly the set of possible worlds in the belief states – **open world assumption** AtR \land CleanR 
    - Problems: requires general theorem proving among logically equivalent sentences
  - Knowledge propositions describing the agent’s knowledge – **closed world assumption** 
    - similar to the second approach 
      \[ K(AtR) \land K(CleanR) \quad \neg K(CleanL) \quad \neg K(\neg CleanL) \]
How Sensing Works?

- **Automatic sensing:**
  - at every time step the agent gets all the available percepts

- **Active sensing:**
  - the percepts are obtained only by executing specific sensory actions, such as *CheckDirt* and *CheckLocation*
Action Description Using Knowledge Propositions

- Suppose the agent moves Left in the alternate-double-Murphy world with automatic local dirt sensing: the agent might or might not leave dirt behind if the square is clean

\[ \text{Action(Left, PRECOND : AtR,} \]
\[ \quad \text{EFFECT : } K(AtL) \land \neg K(AtR) \land \]
\[ \quad \text{when CleanR : } \neg K(\text{CleanR}) \land \]
\[ \quad \text{when CleanL : } K(\text{CleanL}) \land \]
\[ \quad \text{when } \neg \text{CleanL : } K(\neg \text{CleanL}) \]

Initial state: \( K(AtR) \land K(\text{CleanR}) \)

after move Left: \( K(AtL) \land K(\text{CleanL}) \quad K(AtL) \land K(\neg \text{CleanL}) \)
Active Sensing

- With active sensing, the agents get new percepts only by asking for them

\[ \text{Action}(\text{CheckDirt}, \text{PRECOND} : ,) \]
\[ \text{EFFECT} : \text{when } AtL \land \text{CleanL} : K(\text{CleanL}) \land \]
\[ \text{when } AtL \land \neg\text{CleanL} : \neg K(\text{CleanL}) \land \]
\[ \text{when } AtR \land \text{CleanR} : K(\text{CleanR}) \land \]
\[ \text{when } AtR \land \neg\text{CleanR} : \neg K(\text{CleanR})) \]

*Left* followed by *CheckDirt* in the active setting results in the same two belief states as *Left* did in the automatic sensing setting.
Execution Monitoring and Replanning

- An execution monitoring agent:
  - checks its percepts to see whether everything is going according to plan
    - action monitoring
    - plan monitoring, more efficient

- A replanning agent knows what to do when something unexpected happens
  - call a planner again to come up with a new plan to reach the goal
  - repair the old plan
A Monitoring and Replanning Agent

function REPLANNING-AGENT(percept) returns an action

static: KB, a knowledge base (includes action descriptions)
    plan, a plan, initially []
    whole_plan, a plan, initially []
    goal, a goal

TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
current ← STATE-DESCRIPTION(KB, t)
if plan = [] then
    whole_plan ← plan ← PLANNER(current, goal, KB)
if PRECONDITIONS(FIRST(plan)) not currently true in KB then
    candidates ← SORT(whole_plan, ordered by distance to current)
    find state s in candidates such that
    failure ≠ repair ← PLANNER(current, s, KB)
    continuation ← the tail of whole_plan starting at s
    whole_plan ← plan ← APPEND(repair, continuation)
return POP(plan)
Visualize the Process

The agent executes the plan until $E$.
Before executing the remaining plan, it checks preconditions as usual and finds that it is actually in state $O$ rather than state $E$.
It then repairs the plan so that it can get from $O$ to some point $P$ on the original plan.
The new plan is the concatenation of repair and continuation.
Continuous Planning

- The agent lives through a series of goal formation, planning, and acting phases
- It has a cycle of “perceive, remove flaw, act”

```plaintext
function CONTINUOUS-POP-AGENT(percept) returns an action
    static: plan, a plan, initially with just Start, Finish

    action ← NoOp (the default)
    EFFECTS[Start] = UPDATE(EFFECTS[Start], percept)
    REMOVE-FLAW(plan) // possibly updating action
    return action
```
The start state is (a).
The goal state is $\text{On}(C, D) \land \text{On}(D, B)$.
At (b), another agent has interfered, putting D on B.
At (c), the agent has executed $\text{Move}(C, D)$ but has failed, dropping C on A instead.
It retries $\text{Move}(C, D)$, reaching the goal state (d).
The initial plan is constructed by the continuous planning agent, just like other normal partial-order planner.
Continuous Planning

After someone else moves D onto B, the unsupported links supplying Clear(B) and On(D, G) are dropped, producing this plan.
Continuous Planning

The link supplied by Move(D, B) has been replaced by one from Start, and the now-redundant step Move(D, B) has been dropped.

This process is called extending a causal link and is done whenever a condition can be supplied by an earlier instead of a later one without causing a new conflict.
Continuous Planning

After Move(C, D) is executed and removed from the plan, the effects of the Start step reflect the fact that C ended up on A instead of the intended D. The goal precondition On(C, D) is still open.
Continuous Planning

The open condition is resolved by adding Move(C, D) back in.
After Move(C, D) is executed and dropped from the plan, the remaining open condition On(C, D) is resolved by adding a causal link from the new Start step. The plan is now completed.
Summary of Planning

- Standard planning algorithms assume complete and correct information and deterministic, fully observable environments. Many domains violate this assumption.

- Conditional plans allow the agent to sense the world during execution to decide what branch of the plan to follow.

- Execution monitoring detects violations of preconditions for successful completion of the plan.

- A replanning agent uses execution monitoring and splices in repairs as needed.

- A continuous planning agent creates new goals as it goes and reacts in real time.