Planning

Chapter 11
Russell and Norvig

Monkey & Bananas

- A hungry monkey is in a room. Suspended from the roof, just out of his reach, is a bunch of bananas. In the corner of the room is a box. The monkey desperately wants the bananas but he can’t reach them. What shall he do?

Monkey & Bananas (2)

- After several unsuccessful attempts to reach the bananas, the monkey walks to the box, pushes it under the bananas, climbs on the box, picks the bananas and eats them.

- The hungry monkey is now a happy monkey.

Planning

- To solve this problem the monkey needed to devise a plan, a sequence of actions that would allow him to reach the desired goal.

- Planning is a topic of traditional interest in Artificial Intelligence as it is an important part of many different AI applications, such as robotics and intelligent agents.

- To be able to plan, a system needs to be able to reason about the individual and cumulative effects of a series of actions. This is a skill that is only observed in a few animal species and only mastered by humans.

- The planning problems we will be discussing today are mostly Toy-World problems but they can be scaled up to real-world problems such as a robot cruising a space.
Planning vs. Problem Solving

- Planning is a special case of Problem Solving (Search).
- Problem solving searches a state-space of possible actions, starting from an initial state and following any path that it believes will lead it to the goal state. Recording the actions along the path from the initial state to the goal state gives you a plan.
- Planning has two distinct features:
  1. The planner does not have to solve the problem in order (from initial to goal state). It can suggest actions to solve any sub-goals at anytime.
  2. Planners assume that most parts of the world are independent so they can be stripped apart and solved individually (turning the problem into practically sized chunks).

Planning using STRIPS

- The “classical” approach most planners use today is derived from the STRIPS language.
- STRIPS was devised by SRI in the early 1970s to control a robot called Shakey.
- Shakey’s task was to negotiate a series of rooms, move boxes, and grab objects.
- The STRIPS language was used to derive plans that would control Shakey’s movements so that he could achieve his goals.
- The STRIPS language is very simple but expressive language that lends itself to efficient planning algorithms.
- The representation we will use in Prolog is derived from the original STRIPS representation.

STRIPS Representation

- Planning can be considered as a logical inference problem:
  - a plan is inferred from facts and logical relationships.
- STRIPS represented planning problems as a series of state descriptions and operators expressed in first-order predicate logic.
  - State descriptions represent the state of the world at three points during the plan:
    - Initial state, the state of the world at the start of the problem;
    - Current state, and
    - Goal state, the state of the world we want to get to.
  - Operators are actions that can be applied to change the state of the world.
    - Each operator has outcomes i.e. how it affects the world.
    - Each operator can only be applied in certain circumstances. These are the preconditions of the operator.

Planning in Prolog

- As STRIPS uses a logic based representation of states it lends itself well to being implemented in Prolog.
- To show the development of a planning system we will implement the Monkey and Bananas problem in Prolog using STRIPS.
- When beginning to produce a planner there are certain representation considerations that need to be made:
  - How do we represent the state of the world?
  - How do we represent operators?
  - Does our representation make it easy to:
    - check preconditions;
    - alter the state of the world after performing actions; and
    - recognise the goal state?
Representing the World

- In the Monkey&Banana problem we have:
  - objects: a monkey, a box, the bananas, and a floor.
  - locations: we’ll call them a, b, and c.
  - relations of objects to locations. For example:
    - the monkey is at location a;
    - the monkey is on the floor;
    - the bananas are hanging;
    - the box is in the same location as the bananas.

- To represent these relations we need to choose appropriate predicates and arguments:
  - at(monkey,a).
  - on(monkey,floor).
  - status(bananas,hanging).
  - at(box,X), at(bananas,X).

Initial and Goal State

- Once we have decided on appropriate state predicates we need to represent the Initial and Goal states.

- Initial State:
  - on(monkey, floor),
  - on(box, floor),
  - at(monkey, a),
  - at(box, b),
  - at(bananas, c),
  - status(bananas, hanging).

- Goal State:
  - on(monkey, box),
  - on(box, floor),
  - at(monkey, c),
  - at(box, c),
  - at(bananas, c),
  - status(bananas, grabbed).

- Only this last state can be known without knowing the details of the Plan (i.e. how we’re going to get there).

Representing Operators

- STRIPS operators are defined as:
  - NAME: How we refer to the operator e.g. go(Agent, From, To).
  - PRECONDITIONS: What states need to hold for the operator to be applied. e.g. [at(Agent, From)].
  - ADD LIST: What new states are added to the world as a result of applying the operator e.g. [at(Agent, To)].
  - DELETE LIST: What old states are removed from the world as a result of applying the operator. e.g. [at(Agent, From)].

- We will specify operators within a Prolog predicate opn/4:

```
opn( go(Agent, From, To),
    [at(Agent, From)],
    [at(Agent, To)],
    [at(Agent, From) ]).
```

The Frame Problem

- When representing operators we make the assumption that the only effects our operator has on the world are those specified by the add and delete lists.

- In real-world planning this is a hard assumption to make as we can never be absolutely certain of the extent of the effects of an action.

- This is known in AI as the Frame Problem.

- Real-World systems, such as Shakey, are notoriously difficult to plan for because of this problem. Plans must constantly adapt based on incoming sensory information about the new state of the world otherwise the operator preconditions will no longer apply.

- The planning domains we will be working in our Toy-Worlds so we can assume that our framing assumptions are accurate.
### All Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Preconditions</th>
<th>Delete List</th>
<th>Add List</th>
</tr>
</thead>
<tbody>
<tr>
<td>go(X,Y)</td>
<td>at(monkey,X)</td>
<td>at(monkey,X)</td>
<td>at(monkey,Y)</td>
</tr>
<tr>
<td>climb_on(B)</td>
<td>at(monkey,X)</td>
<td>on(monkey,floor)</td>
<td>on(B,floor)</td>
</tr>
<tr>
<td>at(B)</td>
<td>at(B,X)</td>
<td>at(B,Y)</td>
<td></td>
</tr>
<tr>
<td>push(B,X,Y)</td>
<td>at(monkey,X)</td>
<td>at(monkey,X)</td>
<td>at(B,X)</td>
</tr>
<tr>
<td>grab(B)</td>
<td>at(box,X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>at(box,X)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Finding a solution

1. Look at the state of the world:
   - Is it the goal state? If so, the list of operators so far is the plan to be applied.
   - If not, go to Step 2.
2. Pick an operator:
   - Check that it has not already been applied (to stop looping).
   - Check that the preconditions are satisfied. If either of these checks fails, backtrack to get another operator.
3. Apply the operator:
   1. Make changes to the world: delete from and add to the world state.
   2. Add operator to the list of operators already applied.
   3. Go to Step 1.

### Finding a solution in Prolog

```prolog
solve(State, Goal, Plan):-
  is_subset(Goal, State),
  opn(Op, Precons, Delete, Add),
  
  delete_list(Delete, State, Remainder),
  append(Add, Remainder, NewState),
  solve(NewState, Goal, [Op|Sofar], Plan).
```

- On first glance this seems very similar to a normal depth-first search algorithm (unifies with first possible move and pursues it).

### Why use operators?

- In fact, `solve/4` is performing depth-first search through the space of possible actions but because actions are represented as operators instead of predicate definitions the result is significantly different:
  - A range of different actions can be selected using the same predicate `opn/4`.
  - The effect an action has on the world is made explicit. This allows actions to be chosen based on the preconditions of sub-goals: directing our search towards the goal rather than searching blindly.
  - Representing the state of the world as a list allows it to be dynamically modified and this implementation has two advantages:
    - No need of the frame axioms for unchanged facts in the state database.
    - No explicit of adding and extracting a fact to a database because a database needs to revert back to the original state during backtracking; if not, we couldn’t use it to generate a plan in this manner.
Representing the plan

- `solve/4` is a linear planner: it starts at the initial state and tries to find a series of operators that have the cumulative effect of adding the goal state to the world.
- We can represent its behaviour as a flow-chart.
- When an operator is applied the information in its preconditions is used to instantiate as many of its variables as possible.
- Uninstantiated variables are carried forward to be filled in later.

Representing the plan (2)

• `solve/4` chooses the push operator this time as it is the next operator after `go/2` stored in the database and `go(a,X)` is now stored in the SoFar list so `go(X,Z)` can’t be applied again.
• The preconditions of `push/3` require the monkey to be in the same location as the box so the variable location, X, from the last move inherits the value b.

Representing the plan (3)

• The operator only specifies that the monkey must climb on something in the same location; not that it must be something other than itself!
• This instantiation fails once it tries to satisfy the preconditions for the `grab/1` operator. `solve/4` backtracks and matches `climbon(box)` instead.

Representing the plan (4)

For the monkey to grab the bananas it must be in the same location, so the variable location Y inherits c.
This creates a complete plan.
Monkey & Bananas Program

Putting all the clauses together, plus the following, we may easily obtain a plan using the query test(Plan).

```prolog
test(Plan) :- solve([ on(monkey, floor), on(box, floor), at(monkey, a), at(box, b), at(bananas, c), status(bananas, hanging) ], [ status(bananas, grabbed) ], [], Plan).
```

?- test(Plan).
Plan = [grab(banana), climbOn(box), push(box, b, c), go(a, b)]

Inefficiency of forwards planning

- Linear planners like this, that progress from the initial state to the goal state can be unsuitable for problems with a large number of operators.
- Searching backwards from the Goal state usually eliminates spurious paths.
- This is also called Means Ends Analysis.

Backward state-space search

- To solve a list of Goals in state State, leading to CurrentPlan, do:
  - If all the Goals are in State then Plan = CurrentPlan.
  - Otherwise do the following:
    1. Select a still unsolved Goal from Goals.
    2. Find an Action that adds Goal to the current state.
    3. Enable Action by solving recursively the preconditions of Action, giving PrePlan.
    4. PrePlan plus Action are recorded into CurrentPlan and the program goes to step 1.
   - i.e. we search backwards from the Goal state, generating new states from the preconditions of actions, and checking to see if these are facts in our initial state.

- Backward state-space search will usually lead straight from the Goal State to the Initial State as the branching factor of the search space is usually larger going forwards compared to backwards.
- However, more complex problems can contain operators with different conditions so the backward search would be required to choose between them.
- It would require heuristics.
- Also, linear planners like these will blindly pursue sub-goals without considering whether the changes they are making undermine future goals.
- They need someway of protecting their goals.
- Both of these issues will be discussed in the next lecture.
Implementing Backward Search

plan(State, Goals, [], State) :- % Plan is empty
  satisfied(State, Goals). % Goals true in State
plan(State, Goals, Plan, FinalState) :-
  append(PrevPlan, [Action|PostPlan], Plan), % Divide plan
  member(Goal, Goals), % Select a goal
  member(Goal, State), % Relevant action
  can(Action), % Check action is possible
  plan(State, PreCons, PrePlan, MidState1), % Link Action to Initial State.
  apply(MidState1, Action, MidState2), % Apply Action
  plan(MidState2, Goals, PostPlan, FinalState). % Recurse to link Action to rest of Goals.

opn(move(Block, From, To), % Name
  [clear(Block), clear(To), on(Block, From)], % Precons
  [on(Block, To), clear(From)]). % Add List

can(move(Block, From, To)) :-
  is_block(Block), % Block to be moved
  object(To), % "To" is a block or a place
  To \= Block, % Block cannot be moved to itself
  object(From), % "From" is a block or a place
  From \= To, % Move to new position
  Block \= From. % Block not moved from itself

satisfied([], []). % All Goals are satisfied
satisfied(State, [Goal|Goals]) :-
  member(Goal, State), % Goal is in current State
  satisfied(State, Goals).

Blocks World

- Blocks World is THE classic Toy-World problem of AI. It has been used to develop AI systems for vision, learning, language understanding, and planning.
- It consists of a set of solid blocks placed on a table top (or, more often, a simulation of a table top). The task is usually to stack the blocks in some predefined order.
- It lends itself well to the planning domain as the rules, and state of the world can be represented simply and clearly.
- Solving simple problems can often prove surprisingly difficult so it provides a robust testing ground for planning systems.

Protecting Goals

- Backward Search produces a very inefficient plan:
  Plan = [move(b,3,c), move(b,c,3), move(c,a,2), move(a,1,b), move(a,b,1), move(b,3,c), move(a,1,b)]
Protecting Goals

- The plan is inefficient as the planner pursues different goals at different times.
- After achieving one goal (e.g. on(b,c) where on(c,a)) it must destroy it in order to achieve another goal (e.g. clear(a)).
- It is difficult to give our planner foresight so that it knows which preconditions are needed to satisfy later goals.
- Instead we can get it to protect goals it has already achieved.
- This forces it to backtrack and find an alternate plan if it finds itself in a situation where it must destroy a previous goal.
- Once a goal is achieved it is added to a Protected list and then every time a new Action is chosen the Action’s delete list is checked to see if it will remove a Protected goal.

Best-first Planning

- So far, our planners have generated plans using depth-first search of the space of possible actions.
- As they use no domain knowledge when choosing alternative paths, the resulting plans are very inefficient.
- There are three ways heuristic guidance could be incorporated:
  1. The order in which goals are pursued. For example, when building structures you should always start with the foundations then work up.
  2. Choosing between actions that achieve the same goal. You might want to choose an action that achieves as many goals as possible, or has preconditions that are easy to satisfy.
  3. Evaluating the cost of being in a particular state.

Best-first Planning

- The state space of a planning problem can be generated by
  - regressing a goal through all the actions that satisfy that goal, and
  - generating all possible sets of sub-goals that satisfy the preconditions for this action.
  - this is known as Goal Regression.
- The ‘cost’ of each of these sets of sub-goals can then be evaluated.
  - cost = a measure of how difficult it will be to complete a plan when a particular set of goals are left to be satisfied.
- By ordering these sets of sub-goals based on the heuristic evaluation function and always choosing the ‘cheapest’ set, our planner can derive a plan using best-first search.

Partial-Order Plans

- Principle of least commitment
**Partial-Order Plan**

- **Actions**
  The empty plan consists of the start and end actions. The start action has effects corresponding to the initial state. The end action has preconditions corresponding to the goal.

- **Ordering Constraints**
  An ordering constraint states that one action must follow another action, $A < B$.

- **Causal Links**
  For two actions, $A$ and $B$, we say that $A$ achieves $p$ for $B$, written $A \rightarrow B$.

  The plan cannot be refined to negate $p$ between $A$ and $B$.

- **Open Preconditions**
  A precondition that is not achieved by some action in the plan is called open.

**Partial-Order Plans: Definitions**

- 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 (causal link).
- A plan is **consistent** if the ordering constraints and bindings are not contradictory.
- A **solution** is a complete and consistent plan.
- Every total-order plan from a partial-order solution is a plan that achieves the goal from the initial state.

**POP vs Total-Order**

**Plan Space**

- We can search through the space of partial-order plans.

Gradually move from incomplete/vague plans to complete, correct plans.
POP Planning Example: the Monkey, Giraffe and Hamster

- You are at home and decide one day that you want to go out and see a monkey, a giraffe and a hamster, then return home.
- You know that the zoo has a monkey and a giraffe.
- You also know that your friend Bob has a hamster.

POP Planning Example, Actions

- Actions:
  - Action: go(Here, There)
    - Precondition: at(Here)
    - Add: at(There)
    - Delete: at(Here)
  - Action: see(X, Here)
    - Precondition: at(Here), has(Here, X)
    - Add: seen(X)

POP Planning Example: Start and End Actions

- Initial State:
  - Action: Start
    - Effect: at(home), has(bob, hamster), has(zoo, monkey), has(zoo, giraffe)
- Goal State:
  - Action: Finish
    - Precondition: at(home), seen(hamster), seen(monkey), seen(giraffe)

Example I

Start

at(home), has(bob, hamster), has(zoo, monkey), has(zoo, giraffe)

Finish

at(home), seen(hamster), seen(monkey), seen(giraffe)
Example II

Start

- at(L) has(L, hamster)
- at(L) has(L, monkey)
- at(L) has(L, giraffe)

(see hamster loc)
(see monkey loc)
(see giraffe loc)

seen(hamster) seen(monkey) seen(giraffe) at(home)

Finish

Example III

Start

- at(bob) has(bob, hamster)
- at(zoo) has(zoo, monkey)
- at(zoo) has(zoo, giraffe)

(see hamster bob)
(see monkey zoo)
(see giraffe zoo)

seen(hamster) seen(monkey) seen(giraffe) at(home)

Finish

Example IV

Start

- go(L, bob)
- go(L, zoo)

at(bob) has(bob, hamster)
at(zoo) has(zoo, monkey)
at(zoo) has(zoo, giraffe)

see(hamster, bob)
see(monkey, zoo)
see(giraffe, zoo)

seen(hamster) seen(monkey) seen(giraffe) at(home)

Finish

Conflicting Actions

An action conflicts with a causal link if it can possibly destroy its condition.

Demotion: put before go(L, bob)
Promotion: put after see(hamster, bob)
POP Algorithm

function POP(initial, goal, operators) returns plan
    plan ← MAKE-MINIMAL-PLAN(initial, goal)
    loop do
        if SOLUTION?(plan) then return plan
        Sneed, c ← SELECT-SUBGOAL(plan)
        CHOOSE-OPERATOR(plan, operators, Sneed, c)
        RESOLVE-THREATS(plan)
    end
    return plan

function SELECT-SUBGOAL(plan) returns Sneed, c
    pick a plan step Sneed from STEPS(plan)
    with a precondition c that has not been achieved
    return Sneed, c

function CHOOSE-OPERATOR(plan, operators, Sneed, c)
    choose a step Sadd from operators or STEPS(plan) that has c as an effect
    if there is no such step then fail
    add the causal Sneed < Sadd to LINKS(plan)
    add the ordering constraint Sadd < Sneed to ORDERINGS(plan)
    if Sadd is a newly added step from operators then
        add Sadd to STEPS(plan)
        add Start < Sadd < Finish to ORDERINGS(plan)
    end

function RESOLVE-THREATS(plan)
    for each Sthreat that threatens a link Si Sj in LINKS(plan) do
        choose either
        Demotion: Add Sthreat < Si to ORDERINGS(plan)
        Promotion: Add Sj < Sthreat to ORDERINGS(plan)
        if not CONSISTENT(plan) then fail
    end

POP is complete, and contains no repetition (systematic)
Partial-order planning

- It’s nondeterministic
- It starts with a minimal partial plan.
- It satisfies one precondition at a time.
- POP is a regression planner.
- POP is sound and complete.

Heuristics for POP and Beyond

- POP does not represent states directly, so it is harder to estimate how far a POP is from achieving a goal
- Heuristics are needed to choose which plan to refine
  - Heuristic 1: to count the number of distinct open preconditions
  - Heuristic 2: the most-constrained-variable
- Planning graph (a special data structure) to give better heuristic estimation - a sequence of levels that correspond to time steps in the plan
- GRAPHPLAN algorithm to extract a plan from a planning graph – revitalize the planning field in the late 90’s

Summary

- A Plan is a sequence of actions that changes the state of the world from an Initial state to a Goal state.
- Planning can be considered as a logical inference problem.
- STRIPS is a classic planning language.
  - It represents the state of the world as a list of facts.
  - Operators (actions) can be applied to the world if their preconditions hold.
  - The effect of applying an operator is to add and delete states from the world.
- A linear planner can be easily implemented in Prolog by:
  - Representing operators as `ops(Name, [PreCons], [Add], [Delete])`.
  - Choosing operators and applying them in a depth-first manner.
  - Using backtracking-through-failure to try multiple operators.

Summary (cont)

- Backward State-Space can be used to plan backwards from the Goal state to the initial state.
  - It often creates more direct plans,
  - But is still inefficient as it pursues goals in any order.
- Blocks World is a very common Toy-World problem in AI.
- Goal Protection: previously completed goals can be protected by making sure that later actions do not destroy them.
  - Forces generation of direct plans through backtracking.
- Best-first Planning can use knowledge about the problem domain, the order of actions, and the cost of being in a state to generate the ‘cheapest’ plan.
- Partial-Order Planning can be used for problems that contain multiple sets of goals that do not interact.