Planning

Daniel Gigliotti

1 Search vs Planning

Search is a method used to find a solution among a set of possibilities. It involves exploring a set of states or configurations, and using a set of rules or heuristics to determine which ones to explore next, until a goal state is found.

Planning, on the other hand, is a method used to determine a sequence of actions that will lead to a goal state. It involves representing the problem as a set of actions and their effects, and using a planning algorithm to determine the optimal sequence of actions to achieve the goal.

Planning is another approach to search but instead of having a representation which is like a black box (specific for every problem) we have a **factor representation** (it takes a specific form). Furthermore, in planning actions are specified by defining preconditions and effects; this is more scalable: if we want to introduce a new capability we just need to define a new action (add the action description) and the rest will be the same.

Factor representation: the representation is simply done by a set of variables with a finite domain of discrete values.

Planning becomes effective over search when the state space becomes too large and complex.

- **State space**: each node represents a state of the real world and a transition between nodes is an addition of an action in the action-sequence to reach a goal;
- **Plan space**: each node represents a partial plan; when we cross an arc we add an action from the action-set to the partial plan.

A **planning problem** is one in which we have some initial **starting state**, which we wish to transform into a desired **goal state** through the application of a set of **actions**.

- **Predicates** are used to represent relationships between objects or entities. A predicate is a statement that takes one or more arguments and returns a thruth value.

  isParentOf(x, y)
  takes two arguments and returns true if x is parent of y

- **Fluents** are expressions of the form:

  $$P(x_1, x_2, ..., x_n) \text{ or } \neg P(x_1, x_2, ..., x_n)$$

  where P is a predicate, $0 \leq n$ and x_i are either variables or constant symbols. Basically, a fluent is a predicate or the negation of a predicate.

- **States** are sets of instantiated fluents. Each fluent in the state represents if the corresponding property is satisfied (true) or not (false).

Database assumption (also known as **Close World assumption**): a state only contains positive fluents. All the missing ones are assumed to be false.

2 Classical planning

A **classical planning problem** is one in which we have some initial **starting state**, which we wish to transform into a desired **goal state** through the application of a set of **actions**. We need to define 3 things:

1. **States**, which are represented by sets of instantiated literals (with a boolean value or a constant);
2. Actions or **action schema**;
3. An environment that is:
    - fully observable: the agent knows everything about the environment;
    - deterministic: the effect of each action is guaranteed;
    - static: while the agent is thinking, nothing happens (the world does not change if the agent does not make an action and the changes happens instantly;
    - discrete and finite;

Now we can model the problem as a search problem and search for the plan that leads to the goal state. Search can happen:

- Forward (**progression**);
- Backward (**regression**);
- Using heuristics;

3 Action Schemas

Action schemas are a way to represent **actions**. Typically they includes a name, a set of preconditions, and a set of effects. The preconditions specify the conditions that must be true in the current state in order for the action to be applicable. The effects specify the changes that will be made to the state when the action is executed.

An action a with $Preconditions(a) = q$ is **applicable** in a state s iff:

- every positive fluent in q is in s;
- every negative fluent in q is not in s;

```
(:action move
    :parameters ?agent ?from ?to
    :preconditions( and
        (at ?agent ?from) (adj ?from ?to)
        (agent ?agent) (location ?from) (location ?to)
    )
    :effects( and
        (not (at ?agent ?from)) (at ?agent ?to)
    )
)
```

4 Unification

Unification of two expressions is the process of finding a substitution that, whan applied to them, makes them identical.

Example: x/b is a unifier for $p(a, x)$ and $p(a, b)$

Unification is applied to expressions of the form $P(t_1, t_2, ..., t_n)$.
$unify(P(t_1, t_2, ..., t_n), P(s_1, s_2, ..., s_n))$: find a substitution that makes $P(t_1, t_2, ..., t_n)$ and $P(s_1, s_2, ..., s_n)$ identical.

- $unify(P(X), P(a)) = X/a$
- $unify(P(a), Q(a)) =?$ (not admissible)
- $unify(P(X, X), P(a, b)) =?$ (not admissible)

4.1 Progression

Apply actions whose preconditions are satisfied until goal is found or all states have been explored.

We have to specify what is going to be the state after the execution of the action. The **effects of the action** can be positive fluents or negative fluents.

- positive fluents have to be true in the resulting state → ADD(a);
- negative fluents have to be false in the resulting state → DEL(a);
- all the fluents not in the ADD and DEL lists are kept unchanged (**persistence assumption**);

$$\text{Result(s, a)} = \text{(s - DEL(a))} \cup \text{ADD(a)}$$

Irrelevant actions cause the search space to blow.

4.2 Regression

Search backward from the goal to the init state. If we want to go from the goal g to a previous state g' with an action a:

- any positive effect of a for g is deleted;
- each precondition fluent of a is added, unless it already appears;
- any negative effect of a for g is deleted from the negative literals;
- any negative precondition fluent of a for g is added to the negative literals;

$$\text{POS(g')} = \text{(POS(g) - ADD(a))} \cup \text{POS(PREC(a))}$$
$$\text{NEG(g')} = \text{(NEG(g) - DEL(a))} \cup \text{NEG(PREC(a))}$$

4.3 Heuristic for planning

Admissible heuristics can be derived from the relaxed problem which is easier to solve. Search problems can be seen as a graph where nodes are states and edges are actions. There are three ways we can relax such a problem:

- **Ignoring preconditions**; this may lead to actions that achieve multiple goals () or to actions that undo other actions ();
- **Removing negated effects**;
- **Decomposition**: decompose the problem into smaller, individual problems; each one will have a cost; which cost should we use for the heuristic? Should we take the max of the costs or the sum? Better do the max because maybe solving one subgoal can help solving others: summing will return a higher cost in this case, hence the heuristic is not admissible (heuristic is admissible if it does not overestimate the cost).

5 Total-Order Planning (TOP)

Forward/backward (progression/regression) state-space searches are forms of totally ordered plan search: they explore linear sequences of actions that starts with the start state and ends with the goal state, maintaining a total ordering between all actions at every stage of the planning. This way we cannot take advantages of **problem decomposition**.

6 Partial-Order Planning (POP)

A **partial-order plan** specifies all actions that need to be taken, but specifies an ordering between them only when necessary. This allows us to parallelize the computation (actions are often independent one from another) and we can even benefit from a situation in which we need to work on several subgoals independently: we can solve them with subplans and then put everything together.

- Start action has the initial state description as its effects;
- Finish action has the goal description as its precondition;

Each partial plan has four components:

- A **set of actions** that makes up the steps for the plan; an empty plan only has the Start and Finish actions;
- A set of **ordering constraints** between pairs of actions in the form $A \prec B$;
- **Casual links** from outcome of one action to precondition of another $A \xrightarrow{p} B$ (A achieves p for B);
- **Open preconditions**: preconditions of an action not yet casually linked;

A plan is said to be consistent if:

- There are **no cycles** e.g. $A \prec B$ and $B \prec A$;
- There are **no conflicts**:
  If we have $A \xrightarrow{p} B$ (A achieves p for B) and it exists another action C that has effect $\neg p$ which undoes the effect of A, we have a conflict if C is between A and B:
  $A \prec C$ and $C \prec B$ We can solve it by applying:
  - **Demotion** $C \prec A$;
  - **Promotion** $B \prec C$;

If in a consistent plan there is no open precondition, the plan is a **solution**.

The algorithm works in the following way:

1. The plan only contains *Start* and *Finish* with $Start \prec Finish$.
2. The **successor function** is called:
   (a) pick an open condition p of an action B;
   (b) pick an action A that achieves p;
   (c) add the casual link $A \xrightarrow{\text{p}} B$ and $A \prec B$.
   (d) resolve conflicts if possible, otherwise backtrack.
3. The goal test succeed when there are no more open preconditions.

POP is sound and complete:

- **sound**: if the algorithm finds a solution, the solution is correct;
- **complete**: if there is a solution, the algorithm will find it;

POP depends on how it chooses the action each time to fulfill the open precondition. This can be done with a suitable heuristic function.

7 Hierarchical Task Network Planning (HTN)

The planning methods of the preceding chapters all operate with a fixed set of atomic actions. this approach is not sustainable since modern problems require thousands of actions. It is important to focus on the big picture and think about the overall strategy, rather than getting bogged down in the details of individual actions. One way to do this is to use a technique called **hierarchical decomposition**, which involves **breaking down a complex problem into smaller, more manageable parts**. The key benefit of this approach is that at each level of the hierarchy, there are fewer activities to consider, which makes it easier to find the most efficient way to arrange them for the current problem. This approach can reduce the computational cost and increase the chances of success in finding the correct way to achieve the goal.

Planning proceeds by decomposing non-primitive tasks recursively into smaller subtasks, until primitive tasks are obtained.

8 High-Level Actions (HLA)

Each HLA has one or more possible refinements, into a sequence of actions, each of which may be an HLA or a primitive action. An HLA refinement that contains only primitive actions is called an **implementation** of the HLA. We can say, then, that a **high-level plan achieves the goal from a given state if at least one of its implementations achieve the goal from that state**.

9 Search for plans

- If an HLA has **exactly one implementation**, then we can compute the preconditions-effects of the HLA exactly as if it were a primitive action: we just need to consider the sequence of primitive actions that compose it.
- An HLA has **multiple, different, implementations**

  For example, in a "Hawaii Vacation" scenario, one HLA could be "move to the airport" which could have two different implementations: "take bus" and "take taxi".

  We need to search among all the implementations.

9.1 How to search for primitive solutions?

The **hierarchical search** refines HLA from the initial abstract specification to a sequence of primitives:

- Repeatedly choose an HLA in the current plan and replace it with one of its refinements;
- The goal is checked when the plan is refined to a sequence of primitive actions;

This is computationally expensive. We could **search for abstract solutions instead**.

9.2 Search for abstract solutions

The planner should be capable to determine that the HLA (e.g. "take bus" and "take taxi") get the agent to the goal without looking for their refinements.

Approach: **define precondition-effect descriptions of each HLA** as in the case of primitives.

How can we model the effects of a HLA with multiple refinements?

9.3 Conservative approach

A solution might be to include only the positive effects that are achieved for **every** implementation of the HLA and the negative effects of **any** implementation.

9.4 Reachable sets

The **reachable set** of an HLA h from a state s is the set of states reachable by any of the HLA's implementations.

A sequence of HLAs achieves the goal if its reachable set intersect the set of goal states.

Given a sequence of HLAs, it achieves the goal **if its reachable set intersects the set of possible goal states**. If it does not intersect, there is no plan.

9.5 Reachable set descriptions

There are two descriptions which are kind of a upper/lower bound for the reachable set:

- **Optimistic**: may overstate the reachable set of an HLA;
- **Pessimistic**: may understate the reachable set of an HLA;

If an **optimistic** reachable set does **not intersect** the goal, then the plan does **not work**.

If an **optimistic** reachable set does **intersect** the goal, then **we can't say** if the plan will work.

If a **pessimistic** reachable set does **intersect** the goal, then the plan does **work**.

If a **pessimistic** reachable set does **not intersect** the goal, then **we can't say** if the plan will work.

10 Non classical planning

Classical search/planning is not adequate: in the everyday world, agents have to deal with incomplete and incorrect information. In non-classical planning, indeterminacy refers to the uncertainty or ambiguity in the outcome of a plan or action.

- **Bounded indeterminacy**: there is a limited range of possible outcomes the agent could deal with;
- **Unbounded indeterminacy**: the range of possible outcomes is not limited and the agent has no hope to deal with all the possible cases;

10.1 Belief states

In planning, a belief state refers to the current state of knowledge or information that a planner or agent has about the world. It represents the beliefs or assumptions that the agent has about the state of the environment and the outcomes of its actions. The belief state is often used in planning algorithms that incorporate uncertainty, such as probabilistic or partially observable planning.

A belief state can be represented as a probability distribution over the set of possible states of the environment. For example, in a partially observable problem, the agent may not have complete information about the state of the environment, but it can estimate the probability of each possible state given its observations. The belief state is updated as the agent receives new information through its actions and observations.

In practice, belief state is often represented as a vector of probabilities where each element corresponds to one possible state of the world.