Classical Planning

Yashovardhan Srivastava

8 July,2021

Introduction

Planning is an sub-field of AI whose aim is to devise a plan of action to achieve one's goals. This follows a more general approach than logical or problem-solving agents for problems that could not be handled by earlier approaches.

Classical Planning

Classical Planning is an factored representation approach in which state of the world is represented by a collection of variable which is used to construct a complex plan of a action. PDDL(Planning Domain Definition Language) allows us to describe all the four things of search problems - initial state,actions,result of actions and the goal state using action schemas.

Each state in problem could be expressed as a conjunction of fluents(Conjunctive Normal Form). In PDDL, Database Semantics are used: closed world assumption means that fluents that are not mentioned are false, unique names means that fluents that have different names are distinct . PDDL states can be treated either as conjunction of literals, or a set of fluents ,which can be manipulated by logical inferences.

Actions are described by a set of action schemas that define a actions and results. Classical Planning concentrates on problems where most actions leaves most things unchanged. In PDDL, result of an action is defined in terms of what changes, everything that stays the same is left unmentioned.

The action schema consists of action name, a list of all variables used in the schema, a precondition and an effect.It is a lifted representation from propositional logic to a subset of first order logic. For example, the action schema to move from place A to place B:

$$\textbf{Action}(Move(A,B)),\ \textbf{Precond:}At(A) \wedge \neg At(B)\ \textbf{Effect:}\neg At(A) \wedge At(B)$$

The precondition and effect of an action are conjunction of literals. The precondition defines the states in which action can be executes and the effect defines the result of executing the action.The result of executing action **a** in state **s** is defined as s_1 which is represented by set of fluents formed by starting with **s** , removing the fluents that appear as negative literals in action's effect(called delete list of a or **Del(a)**), and adding the fluents that are positive literals in the action effect.

A set of action schemas serves as a definition of

$$Result(s,a) = (s - Del(a)) \cup Add(a).$$

planning domain. A specific problem within the domain is defined with the addition of initial state and a goal. The initial state is just a conjunction of ground terms and goal is just like a precondition: a conjunction of literals. The problem is solved when we can find a sequence of actions that end in a state **s** that entails the goal.

Planning as State-Space Search

Planning , with the aid of action schemas can can define a search problem and search through a state space to find a goal. One of the advantages of using actions schemas is that we can also search backward from the goal , that is starting from goal and searching for initial state. Let us see forward and backward searches:

Forward(progression) state-space search

Since Planning problems can be translated into search problems, we can solve planning problems with any of the heuristic search algorithms or local search algorithms.The problem with forward state space-searching is that it is too efficient to be practical.Forward searching is prone to exploring irrelevant actions. This is because planning problems often have large state spaces: for even relatively small problem with high ground conditions, an uninformed forward search-algorithm would have to enumerate all of those conditions and find one which leads to goal. Accurate heuristics(either domain dependent or independent) is what makes forward search feasible.

Backward(regression) state-space search

In regression search, we start at the goal and apply the actions backwards until we find sequence of steps to reach the initial state. It is called **relevant state** search because we only consider actions that are relevant to the goal. In general, backward search works only when we know to regress from a state description to the predecessor state description. PDDL representation allows us to regress actions if the domain can be expressed in PDDL. Given a ground goal description **g** and a ground action **a**, the regression from **g** over **a** gives a state description **g'**

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

That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed.

The main issue is to decide which actions to regress over. In forward search - we chose actions that are **applicable** , while in backward search - we choose actions that are **relevant**. A relevant action is the one which contributes to the goal - at-least one of the actions's effects (either positive or negative) must unify with an element of the goal[1].Backward Searching keeps the branching factor lower than forward search, for most problem domains.However, the fact that backward search uses state sets(owing to the fact that it is depth-first) rather than instead of individual state makes it difficult to come up with good heuristics. This is the main reason that majority of current systems prefer forward search.

Heuristics for Planning

As the problem domain for planning problems are much bigger that classic search problem domains, neither forward or backward search is efficient without good heuristics.It is also necessary that heuristics should be **admissible** -they should not overestimate distance to goal state from current state. We saw earlier that admissible heuristics can be derived from relaxed problem that is easier to solve,then the cost of the solution of this becomes the heuristic of the original problem. Planning uses factored representation for states and action schemas that makes it easier to define good domain independent heuristics. The general idea is to find a path between goal state and initial state with some conditions of the problem relaxed.Some of the heuristics that make use of that are:

- **Ignore precondition heuristic** drops all the preconditions from actions, which allows any action to be applicable in every state, and any single goal can be achieved in one step. For many problems, an accurate heuristic is obtained by considering that actions can achieve multiple goals and ignoring the fact that actions can undo the effects of other actions. Actions schemas description makes it easier to define heuristics.

- **Ignore delete list heuristics** assumes that goals and preconditions contain only positive literals.We want to create a relaxed version of the original problem that will be easier to solve, and where length of solution makes up for a good heuristic. We can do that by removing the delete list from all actions(removing negative literals from the effects) - which makes it possible to make monotonic progress towards goal. It turns out it is NP-Hard to find optimal solution of this relaxed problem, but hill climbing works well(in polynomial time).

- **State Abstraction** is another way wherein there is relaxation on number of states - a many to one mapping from states in the ground representation of the problem to abstract representation. In our graph analogy of problem, this means reducing the number of nodes(states). This is particularly

[1]It is less obvious that action must not have any effect(positive or negative) that negates an element of goal

helpful in planning problems because planning domains have a huge number of states .The easiest form of state abstraction is to **ignore some fluents**.

A key idea in defining a heuristic is decomposition(or dicretization) : dividing the problem into sub-problems and solving them independently, and then combining the parts. Here it is assumed that **sub-goal independence** is valid, that is solving the conjunction of sub-goals is approximated by the sum of the costs of solving each sub-goal independently. There is great potential in cutting down state-space by forming abstractions. The key trick is to choose the correct abstractions for the correct domain that makes the overall problem efficient.

Planning Graphs

All of the heuristics defined above can suffer from inaccuracies. This section introduces **Planning Graph**- a data structure that can be used to give better heuristic estimates, solutions of which can found using an algorithm called $GRAPHPLAN$.

Imagine a tree of all possible actions from the initial states to the goal state. If we indexed this tree appropriately, we can answer any question about the planning domain. Since the number of states in a planning problem are very high(and thus the tree is exponential), this approach is impractical. A planning graph is a polynomial time assumption of this graph - that is more pragmatic than our previous approach. The planning graph cannot answer definitively whether the goal state G is reachable from state S or not, but it can approximate how many steps it take to reach G. The estimate is admissible and therefore it serves as an accurate heuristic.

A planning graph is a directed graph organized into levels: first a level for the initial state S_0, consisting of nodes representing each fluents that holds in S_0 , then a level A_0 consisting of nodes of each ground actions that might be applicable in S_0 then alternating levels S_i followed by A_i; until we reach a termination condition.

Planning graphs works only for propositional planning problems - ones with no variables. Despite the resulting increase in the size of the problem description, planning graphs have proved to be an effective tool for solving hard planning problems.

Planning graph for a particular problem is a structure wherein every A_i level contains all the actions that are applicable in S_i, along with constraints saying that two actions cannot both be executed at the same level(in the form of **mutex**[2] links). Every S_i level contains all the levels contains all the literals that could result from any possible choice of action in A_{i-1}, along with the constraints saying which pair of literals are not possible. It is to be noted that process of constructing the planning graph does *not* require choosing among actions, which would entail combinatorial search, instead it just records the impossibility of certain choices using mutex links.

A mutex relation holds between two **actions** at a given level if any of the following conditions hold:

- **Inconsistent effects:** One action negates an effect of the other
- **Interference:**One of the effects of one action is the negation of a precondition of the other.
- **Competing needs:** One of the preconditions of one action is mutually exclusive with a precondition of the other.

A mutex relation between two **literals** at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called **inconsistent support**.

Planning graphs for heuristic estimation

A planning graph once constructed gives a rich source of information about the problem. If a problem is unsolvable, planning graphs can predict it, secondly it gives an admissible - but not accurate heuristic for estimating the cost to reach the goal state. For this reason it common to use a **serial planning**

[2]**Mutex** links or mutually exclusive links between literals and actions indicates that they could not appear together.

graph for computing heuristics, in which only one action can actually occur at a given time step. This is done by adding mutex links between every pair of non-persistence actions.

To estimate the cost of conjunction of goals, there are mainly three approaches, namely:

- **Max-level heuristic:** It simply takes the maximum level cost of any of the goals, this is admissible,but not necessarily accurate.
- **Level sum heuristic:**It returns the sum of level cost of the goals, following the sub-goal independence assumption.This can be admissible, but it works well in practice for problems that are largely decomposable.
- **Set-level heuristic:** It finds the level at which all literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. It is admissible, it dominates max-level heuristic and works extremely well on tasks in which there is a good deal of interaction among sub-plans.

The GRAPHPLAN algorithm

GRAPHPLAN allows us to extract a plan from the planning graph, rather than just a heuristic. The algorithm repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non- mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on. The pseudo-code for it is given below:(see Ghallab *et al.* 2004)

```
function GRAPHPLAN(problem) ->return solution or failure
        graph ⇐ INITIAL-PLANNING_GRAPH(problem)
        goals ⇐ CONJUNCTS(problem.GOAL)
        nogoods ⇐ an empty hash table
        for tl=0 to ∞ do:
                if goals all non-mutex in S_t of graph then :
                        solution ⇐ EXTRACT-SOLUTION(graph,goals,
                                                    NumLevels(graph),nogoods)
                        if solution ≠ failure then return solution
                if graph and nogoods have both levelled off then return failure:
                        graph ⇐ EXPAND-GRAPH(graph,problem)
```

The GRAPHPLAN initializes the planning graph to one level representing the initial state. EXPAND-GRAPH adds the actions whose preconditions exists at the initial level along with persistence actions for all its literals. The effect of these actions are added at the next level.EXPAND-GRAPH then looks for mutex relations and adds them to the graph.

When we go back to the start of the loop, all the literals from the goal are present in the next state, and none of them is mutex with any other. That means that a solution might exist, and EXTRACT-SOLUTION will try to find it. We can think of EXTRACT-SOLUTION as a Boolean CSP - with variables as actions and constraints as mutex relations. In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the $(level, goals)$ as **no-good**, which helps in breaking from the loop and instead of searching again.

We know that planning is PSPACE-complete and that constructing the planning graph takes polynomial time, so it must be the case that the solution is intractable in the worst case. Therefore, the need of heuristic is necessary among actions. One approach that works well in practice is a greedy algorithm based on level cost of literals.

Classical Planning Approaches

Currently, the most popular approaches to fully automated planning are Translating a Boolean satisfiability(SAT) problem, Forward state-space search and Searching using a planning graph. Apart from these, numerous international competitions are held every year for solving problems in the domain. This section describe some of these approaches.

Classical Planning as Boolean Satisfiability

SATPLAN as described earlier solves planning problems that are expressed in propositional logic. In this section, we will show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of steps.

1. Propositionalize the actions : Replace each action schema with a set of ground actions formed by substituting constants for each of the variables.
2. Define the initial state : Assert the initial state for each fluent in the problem's initial state, and negation for every fluent not mentioned in the initial state.
3. Propositionalize the goal : For every variable in the goal, replace the literals that contain the variable with a disjunction over constants.
4. Add successor-state axioms : For each fluent, add an axiom that defines the successor state.
5. Add precondition axioms : For each ground action, add an axiom such that if an action is taken - then the preconditions must have been true.
6. Add action exclusion axioms : Assert that every action is distinct from every other action.

The resulting translation is in the form that we can hand to SATPLAN to find a solution.

Planning as First-Order Logical Deduction

Propositional logic representation of planning problems has their own limitations - such as the fact that the notion of time is tied directly to the fluents. First -order logic lets us get around this limitation by introducing the notion of branching *situations* using a representation known as situation calculus - that works like this :

1. The initial state is called a **situation**. A situation here corresponds to a sequence, or history of actions.
2. A function or relation that can vary from one situation to next is a fluent. By convention, the situation is always the last argument to the fluent.
3. Each action's preconditions are described with a **possibility axiom** that says when the action can be taken
4. Each fluent is described with a successor-state axiom that syas what happens to the fluent, depending on what action is taken.
5. We also need a **unique action axiom** that make sure that all actions are different.
6. A solution is a situation that satisfies the goal.

Work in situation calculus is still going on. So far, there have not been any large-scale planning programs based on logical deductions over situation calculus. This is due to the fact that inefficient inference in first-order logic and that the field has not yet developed heuristics for planning with situation calculus.

Planning as Refinement Of Partially Ordered Plans

All of the approaches so far construct *totally ordered* plans consisting of linear sequence of actions. This representation ignores the fact that many subproblems are independent. An alternative to this is to represent plans as *partially ordered* structures. Partially ordered plans are created by a search through the space of plans rather than through the state space. We start with an empty plan consisting of just the initial state and the goal. The search condition then looks for a **flaw** in the plan and makes an addition to the plan to correct the flaw. This however, introduces a new flaw that preconditions to the actions are often not satisfied. The search keeps on adding to the plan until all flaws are resolved. At every step, we make the **least commitment** possible to fix the flaw.

Despite having some computational disadvantages, partial ordering planning remains an important part of the field. For some specific tasks such as operations scheduling, partial ordering planning(with domain specific heuristics) is the technology of the choice. Operational plans for spacecrafts and rovers are generated by partial order planners and are then checked by human operators before being uploaded to the vehicles for execution.

References

[1] Russel,Stuart and Norvig,Peter.*Artificial Intelligence-A Modern Approach*(3rd ed.).Pearson Education,2015