JOURNAL OF CRITICAL REVIEWS
ISSN- 2394-5125 VOL 5, ISSUE 03, 2018

Analysis Of Planning Approaches And Algorithms For Planning With State Space SearchForward Search

[1]Kola Vasista, Director, Dark Horse Analytics Pvt Ltd, Hyderabad, India
[2]Dr. Ramesh Kumar Bhatia, Professor, KIIT, Raipur, India

AI as the study of rational action, which means that planning—devising a plan of action to achieve one's goals—is a critical part of AI. We have seen two examples of planning agents so far the search-based problem-solving agent. The problem-solving agent can find sequences of actions that result in a goal state. But it deals with atomic representations of states and thus needs good domain- specific heuristics to perform well. This paper provides the analysis of planning approaches and algorithms for planning with State Space SearchForward search.

Index Terms: Artificial Intelligence, Machine Learning, complexity

I. DEFINITION OF CLASSICAL PLANNING

In response to this, planning researchers have settled on a **factored representation**— one in which a state of the world is represented by a collection of variables. We use a language called **PDDL**, the Planning Domain Definition Language that allows us to express all $4Tn^2$ actions with one action schema. There have been several versions of PDDL.we select a simple version and alter its syntax to be consistent with the rest of the book. We now show how PDDL describes the four things we need to define a search problem: the initial state, the actions that are available in a state, the result of applying an action, and the goal test.

The schema consists of the action name, a list of all the variables used in the schema, a precondition andan effect.

A set of action schemas serves as a definition of a planning domain. A specific problem within the domain is defined with the addition of an initial state and a goal.

state is a conjunction of ground atoms. (As with all states, the closed-world assumption is used, which means that any atoms that are not mentioned are false.)

Example: Air cargo transport

An air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: Load , Unload , and Fly . The actions affect two predicates: In(c, p) means that cargo c is inside plane p, and At(x, a) means that object x (either plane or cargo) is at airport a. Note that some care must be taken to make sure the At predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be At anywhere when it is In a plane; the cargo only becomes At the new airport when it is unloaded. So At really means "available for use at a given location."

II. The complexity of classical planning

We consider the theoretical complexity of planning and distinguish two decision problems. PlanSAT is the question of whether there exists any plan that solves a planning problem. Bounded PlanSAT asks whether there is a solution of length k or less; this can be used to find an optimal plan.

The first result is that both decision problems are decidable for classical planning. The proof follows from the fact that the number of states is finite. But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semi decidable: an algorithm exists that will terminate with the correct answer for any solvable problem, but may not terminate on unsolvable problems. The Bounded PlanSAT problem remains decidable even in the presence of function symbols.

Both PlanSAT and Bounded PlanSAT are in the complexity class PSPACE, a class that is larger (and hence more difficult) than NP and refers to problems that can be solved by a deterministic Turing machine with a polynomial amount of space. Even if we make some rather severe restrictions, the problems remain quite difficult.

III. Algorithms for Planning with State Space SearchForward (progression) state-space search

Now that we have shown how a planning problem maps into a search problem, we can solve planning problems with any of the heuristic search algorithms(provided we keep track of the actions used to reach the goal). From the earliest days of planning research (around 1961) until around 1998 it was assumed that forward state-space search was too inefficient to be practical. It is not hard to come up with reasons why .

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy of AI: A Modern Approach from an online bookseller. Suppose there is an action schema Buy(isbn) with effect Own(isbn). ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.
Second, planning problems often have large state spaces. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no

planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let’s say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 2000 nodes.

Backward (regression) relevant-states search:

In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. It is called relevant-states search because we only consider actions that are relevant to the goal (or current state). As in belief-state search, there is a set of relevant states to consider at each step, not just a single state.

We start with the goal, which is a conjunction of literals forming a description of a set of states—for example, the goal Poor Famous describes those states in which Poor is false, Famous is true, and any other fluent can have any value. If there are n ground flaunts in a domain, then there are 2n ground states (each fluent can be true or false), but 3n descriptions of sets of goal states (each fluent can be positive, negative, or not mentioned).

In general, backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the n-queens problem because there is no easy way to describe the states that are one move away from the goal. Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it.

The final issue is deciding which actions are candidates to regress over. In the forward direction we chose actions that were applicable—those actions that could be the next step in the plan. In backward search we want actions that are relevant—those actions that could be the last step in a plan leading up to the current goal state.

Heuristics for planning:

Neither forward nor backward search is efficient without a good heuristic function. Recall that a heuristic function h(s) estimates the distance from a state s to the goal and that if we can derive an admissible heuristic for this distance—one that does not overestimate—then we can use A∗ search to find optimal solutions. An admissible heuristic can be derived by defining a relaxed problem that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

By definition, there is no way to analyze an atomic state, and thus it it requires some ingenuity by a human analyst to define good domain-specific heuristics for search problems with atomic states. Planning uses a factored representation for states and action schemas. That makes it possible to define good domain-independent heuristics and for programs to automatically apply a good domain-independent heuristic for a given problem.

IV. Planning Graphs

All of the heuristics we have suggested can suffer from inaccuracies. This section shows how a special data structure called a planning graph can be used to give better heuristic estimates. These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, we can search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.

A planning problem asks if we can reach a goal state from the initial state. Suppose we are given a tree of all possible actions from the initial state to successor states, and their successors, and so on. If we indexed this tree appropriately, we could answer the planning question "can we reach state G from state S0" immediately, just by looking it up. Of course, the tree is of exponential size, so this approach is impractical. A planning graph is polynomial-size approximation to this tree that can be constructed quickly. The planning graph can't answer definitively whether G is reachable from S0, but it can estimate how many steps it takes to reach G. The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.

A planning graph is a directed graph organized into levels: first a level S0 for the initial state, consisting of nodes representing each fluent that holds in S0; then a level A0 consisting of nodes for each ground action that might be applicable in S0; then alternating levels Si followed by Ai; until we reach a termination condition (to be discussed later).

Roughly speaking, Si contains all the literals that could hold at time i, depending on the actions executed at preceding time steps. If it is possible that either P or ¬P could hold, then both will be represented in Si. Also roughly speaking, Ai contains all the actions that could have their preconditions satisfied at time.

We say "roughly speaking" because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, a literal might show up at level Sj when actually it could not be true until a later level, if at all. (A literal will never show up too late.) Despite the possible error, the level j at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state.

We now define mutex links for both actions and literals. A mutex relation holds between two actions ata given level if any of the following three conditions holds:

• Inconsistent effects: one action negates an effect of the other. For example, Eat(Cake) and the persistence of Have(Cake) have inconsistent effects because they disagree on the effect 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 its precondition.

• Competing needs: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, Bake(Cake) and Eat(Cake) are mutex because they compete on the value of the Have(Cake) precondition.

A mutex relation holds 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. For example, Have(Cake) and Eaten(Cake) are mutex in S1 because the only way of achieving Have(Cake), the persistence action, is mutex with the only way of achieving Eaten(Cake), namely Eat(Cake). In S2 the two literals are not mutex, because there are new ways of achieving them, such as Bake(Cake) and the persistence of Eaten(Cake), that are not mutex.
other Classical Planning Approaches:

Currently the most popular and effective approaches to fully automated planning are:

- Translating to a Boolean satisfiability (SAT) problem
- Forward state-space search with carefully crafted heuristics
- Search using a planning graph

These three approaches are not the only ones tried in the 40-year history of automated planning. Figure

10.11 shows some of the top systems in the International Planning Competitions, which have been held every even year since 1998. In this section we first describe the translation to a satisfiability problem and then describe three other influential approaches: planning as first-order logical deduction; as constraint satisfaction; and as plan refinement.
Classical planning as Boolean satisfiability :

we saw how SATPLAN solves planning problems that are expressed in propositional logic. Here we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

- Proposition Alize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, butwill be used in subsequent steps.
- Define the initial state: assert F 0 for every fluent F in the problem's initial state, and ¬F for every fluent not mentioned in the initial state.

V. Conclusion

Planning combines the two major areas of AI we have covered so far: search and logic. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has led both to improvements in performance amounting to several orders of magnitude in the last decade and to an increased use of planners in industrial applications. Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge that dominate existing methods. This paper provided the analysis of planning approaches and algorithms for planning with State Space SearchForward search.

References

1) BONOMI, F., MILITO, R., ZHU, J., AND ADDEPALLI, S. Fog computing and its role in the internet of things. In Proceedings of the first edition of the MCC workshop on Mobile cloud computing (2012), ACM, pp. 13–16.

2) BRUSH, A., FILIPPOV, E., HUANG, D., JUNG, J., MAHAJAN, R., MARTINEZ, F., MAZHAR, K., PHANISHAYEE, A., SAMUEL, A., SCOTT, J., ET AL. Lab of things: a platform for conducting studies with connected devices in multiple homes. In Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication (2013), ACM, pp. 35–38.

3) CHALLEN, G., HASELEY, S., MAITI, A., NANDUGUDI, A., PRASAD, G., PURI, M., AND WANG, J. The mote is dead: long live the discarded smartphone! In Proceedings of the 15th Workshop on Mobile Computing Systems and Applications (2014), ACM, p. 5.

4) CHU, Y.-H., RAO, S. G., SESHAN, S., AND ZHANG, H. A case for end system multicast. Selected Areas in Communications, IEEE Journal on 20, 8 (2002), 1456–1471.

5) COOPERSTOCK, J. R., FELS, S. S., BUXTON, W., AND SMITH, K. C. Reactive environments. Communications of the ACM 40, 9 (1997), 65–73.

6) DAVIS, A., PARIKH, J., AND WEIHL, W. E. Edgecomputing: extending enterprise applications to the edge of the internet. In Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (2004), ACM, pp. 180–187.