AI Planning Historical Developments by Ryan Shrott

In this article, I will examine three major developments in the field of **AI planning research**. For each development, I will provide a short summary of the original paper along with appropriate examples to demonstrate several use cases. I will begin by describing a general framework, STRIPS, under which all planning problems can be formulated. I will then examine two state of the art algorithms which work under the STRIPS formulation. Instead of greedily searching for a solution from the start, the GraphPlan algorithm constructs a Planning Graph object which can be used to obtain a solution. A Planning Graph object can also be used for automating heuristics for any planning problem specified in the STRIPS framework. Finally, I will examine an algorithm which uses a heuristic search planner to automate the production of effective heuristics. The final algorithm was the winner of the AIPS98 Planning Contest in 1998.

Development 1: STRIPS (1971)

In 1971, Richard Fikes and Nils Nilsson at Stanford Research Institute developed a new approach to the application of theorem proving in problem solving. The model attempts to find a sequence of operators in a space of world models to transform the initial world model into a model in which the goal state exists. It attempts to model the world as a set of first-order predicate formulas and is designed to work with models consisting of a large number of formulas.

In the STRIPS formulation, we assume that there exists a set of applicable operators which transform the world model into some other world model. The task of the problem solver is to find a sequence of operators which transform the given initial problem into one that satisfies the goal conditions. Operators are the basic elements from which a solution is built. Each operator corresponds to an action routine whose execution causes the agent to take certain actions. In STRIPS, the process of theorem proving and searching are separated through a space of world models.

Formally, the problem space for STRIPS is defined by the initial world model, the set of available operators and their effects on world models, and the goal statement. The available operators are grouped into families called schemata. Each operator is defined by a description consisting of two main parts: the effects the operator has and conditions under which the operator is applicable. A problem is said to be solved when STRIPS produces a world model that satisfies the goal statement.

Let us now consider an example of applying the STRIPS language to an Air Cargo transport system using a planning search agent. Suppose we have an initial state of Cargo 1 at SFO, Cargo 2 at JFK, Plane 1 at SFO and Plane 2 at JFK. Now suppose we want to formulate an optimal plan to transport Cargo 1 to JFK and Cargo 2 to SFO. Summarizing this problem description, we have:

```
Init(At(C1, SFO) ∧ At(C2, JFK)
 ∧ At(P1, SFO) ∧ At(P2, JFK)
 ∧ Cargo(C1) ∧ Cargo(C2)
 ∧ Plane(P1) ∧ Plane(P2)
 ∧ Airport(JFK) ∧ Airport(SFO))
Goal(At(C1, JFK) ∧ At(C2, SFO))
```

We can write a function that formally defines this formulation as follows:

```
def air_cargo_p1() -> AirCargoProblem:
    cargos = ['C1', 'C2']
    planes = ['P1', 'P2']
    airports = ['JFK', 'SFO']
    pos = [expr('At(C1, SFO)'),
           expr('At(C2, JFK)'),
```

```
            expr('At(P1, SFO)'),
            expr('At(P2, JFK)'),
            ]
    neg = [expr('At(C2, SFO)'),
            expr('In(C2, P1)'),
            expr('In(C2, P2)'),
            expr('At(C1, JFK)'),
            expr('In(C1, P1)'),
            expr('In(C1, P2)'),
            expr('At(P1, JFK)'),
            expr('At(P2, SFO)'),
            ]
    init = FluentState(pos, neg)
    goal = [expr('At(C1, JFK)'),
             expr('At(C2, SFO)'),
             ]
    return AirCargoProblem(cargos, planes, airports, init, goal)
```

The AirCargoProblem class would be initialized as follows:

```
class AirCargoProblem(Problem):
    def __init__(self, cargos, planes, airports, initial: FluentState,
goal: list):
        """:param cargos: list of str
            cargos in the problem
        :param planes: list of str
            planes in the problem
        :param airports: list of str
            airports in the problem
        :param initial: FluentState object
            positive and negative literal fluents (as expr) describing
initial state
        :param goal: list of expr
            literal fluents required for goal test
        """
        self.state_map = initial.pos + initial.neg
        self.initial_state_TF = encode_state(initial, self.state_map)
        Problem.__init__(self, self.initial_state_TF, goal=goal)
        self.cargos = cargos
        self.planes = planes
        self.airports = airports
        self.actions_list = self.get_actions()
```

We use the get_actions method to instantiate a list of all action/operator objects which can act on the states. There are three types of actions we can take in this air cargo problem: load, unload and fly. The get_actions class method collect all such possible actions.

To define the operators which can act on a certain state, we can define the actions class method as follows:

```
def actions(self, state: str) -> list:
 """ Return the actions that can be executed in the given state.
:param state: str
 state represented as T/F string of mapped fluents (state variables)
            e.g. 'FTTTFF'
        :return: list of Action objects
        """
        # TODO implement
```

```
        possible_actions = []
        kb = PropKB()
        kb.tell(decode_state(state, self.state_map).pos_sentence())
        for action in self.actions_list:
              is_possible = True
              for clause in action.precond_pos:
                if clause not in kb.clauses:
                    is_possible = False
              for clause in action.precond_neg:
                if clause in kb.clauses:
                    is_possible = False
              if is_possible:
                possible_actions.append(action)

        return possible_actions
```

We also need to define a method for applying an action to a given state. The result of executing action *a* in state *s* is defined as a state *s'* which is represented by the set of fluents formed by starting with *s*, removing the fluents that appear as negative literals in the action's effects and adding the fluents that are positive literals in the action's effects.

```
def result(self, state: str, action: Action):
        """ Return the state that results from executing the given
        action in the given state. The action must be one of
        self.actions(state).
:param state: state entering node
        :param action: Action applied
        :return: resulting state after action
        """
        # TODO implement
        new_state = FluentState([], [])
        old_state = decode_state(state, self.state_map)

        for fluent in old_state.pos:
            if fluent not in action.effect_rem:
                new_state.pos.append(fluent) # add positive fluents
which are in the old state and should not be removed

        for fluent in action.effect_add:
            if fluent not in new_state.pos:
                new_state.pos.append(fluent) # add positive fluents
which should be added and have not already been added

        for fluent in old_state.neg:
            if fluent not in action.effect_add:
                new_state.neg.append(fluent) # add negative fluents
which are in the old state and should not be added

        for fluent in action.effect_rem:
            if fluent not in new_state.neg:
                new_state.neg.append(fluent) # add negative fluents
which should be removed but have not already been removed from the
negative state

        return encode_state(new_state, self.state_map)
```

Finally, we need to define the goal test method which provides a boolean value indicating whether the goal state is satisfied.

```
def goal_test(self, state: str) -> bool:
        """ Test the state to see if goal is reached
:param state: str representing state
        :return: bool
        """
        kb = PropKB()
        kb.tell(decode_state(state, self.state_map).pos_sentence())
        for clause in self.goal:
            if clause not in kb.clauses:
                return False
        return True
```

This class provides an example of a STRIPS formulation. In particular, we have specified the initial state, the goal state and a set of actions which specify preconditions and postconditions. A plan for this planning instance is a sequence of operators that can execute from the initial state and lead to a goal state. We can use progression search algorithms to form optimal plans for this example problem. Using Breadth-First-Search on this problem, the optimal plan would be Load(C2, P2, JFK), Load(C1, P1, SFO), Fly(P2, JFK, SFO), Unload(C2, P2, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK).

Development 2: Planning Graphs (1997)

In 1997, Avrium Blum and Merrick Furst at Carnegie Mellon developed a new approach to planing in STRIPS-like domains. It involved constructing and analyzing a brand new object called a Planning Graph. They developed a routine called GraphPlan which obtains the solution to the planning problem using a Planning Graph construct.

The idea is that rather than greedily searching, we first create a Planning Graph object. The Planning Graph is useful because it inherently encodes useful constraints explicitly, thereby reducing the search overhead in the future. Planning Graphs can be constructed in polynomial time and have polynomial size. On the other hand, the state space search is exponential and is much more work to build. Planning graphs are not only based on domain information, but also the goals and initial conditions of the problem and an explicit notion of time.

Planning Graphs have similar features to dynamic programming problem solvers. The GraphPlan algorithm uses a planning graph to guide its search for a plan. The algorithm guarantees that the shortest plan will be found (similar to BFS).

Edges in a planning graph represent relations between actions and propositions. If a valid plan does exist in the STRIPS formulation, then that plan must exist as a subgraph of the Planning Graph. Another essential feature of planning graphs involve specifying mutually exclusive (mutex) relationships. Two actions are mutex if no valid plan could possibly contain both, and two states are mutex if no valid plan could make both simultaneously true. Exclusion relationships propagate intuitively useful facts about the problem throughout the graph.

The GraphPlan algorithm operates on the planning graph as follows: Start with a planning graph that only encodes the initial conditions. In Stage *i,* GraphPlan take the the planning graph from state *i-1* and extends it one time step and then searches the extended planning graph for a valid plan of length *i*. If it finds a solution, then it halts, otherwise it continues to the next stage. Any plan that the algorithm finds is a legal plan and it will always find a plan if one exists. The algorithm also has a termination guarantee that is not provided by most planners.

Let's now construct a basic planning graph object and use it solve the Air Cargo Problem above. I leave out low level implementation details to reduce the length of this article. As always, the code is hosted on my GitHub. We initialize the structure as follows:

```
class PlanningGraph():
    """
    A planning graph as described in chapter 10 of the AIMA text. The
planning
    graph can be used to reason about
    """
def __init__(self, problem: Problem, state: str, serial_planning=True):
        """
        :param problem: PlanningProblem (or subclass such as
AirCargoProblem or HaveCakeProblem)
        :param state: str (will be in form TFTTFF... representing
fluent states)
        :param serial_planning: bool (whether or not to assume that
only one action can occur at a time)
        Instance variable calculated:
            fs: FluentState
                the state represented as positive and negative fluent
literal lists
            all_actions: list of the PlanningProblem valid ground
actions combined with calculated no-op actions
            s_levels: list of sets of PgNode_s, where each set in the
list represents an S-level in the planning graph
            a_levels: list of sets of PgNode_a, where each set in the
list represents an A-level in the planning graph
        """
        self.problem = problem
        self.fs = decode_state(state, problem.state_map)
        self.serial = serial_planning
        self.all_actions = self.problem.actions_list +
self.noop_actions(self.problem.state_map)
        self.s_levels = []
        self.a_levels = []
        self.create_graph()
```

The create_graph method is as follows:

```
def create_graph(self):
        """ build a Planning Graph as described in Russell-Norvig 3rd
Ed 10.3 or 2nd Ed 11.4
The S0 initial level has been implemented for you.  It has no parents
and includes all of
        the literal fluents that are part of the initial state passed
to the constructor.  At the start
        of a problem planning search, this will be the same as the
initial state of the problem.  However,
        the planning graph can be built from any state in the Planning
Problem
This function should only be called by the class constructor.
:return:
            builds the graph by filling s_levels[] and a_levels[] lists
with node sets for each level
        """
        # the graph should only be built during class construction
```

```
        if (len(self.s_levels) != 0) or (len(self.a_levels) != 0):
            raise Exception(
                'Planning Graph already created; construct a new
planning graph for each new state in the planning sequence')
# initialize S0 to literals in initial state provided.
        leveled = False
        level = 0
        self.s_levels.append(set())  # S0 set of s_nodes - empty to
start
        # for each fluent in the initial state, add the correct literal
PgNode_s
        for literal in self.fs.pos:
            self.s_levels[level].add(PgNode_s(literal, True))
        for literal in self.fs.neg:
            self.s_levels[level].add(PgNode_s(literal, False))
        # no mutexes at the first level
# continue to build the graph alternating A, S levels until last two S
levels contain the same literals,
        # i.e. until it is "leveled"
        while not leveled:
            self.add_action_level(level)
            self.update_a_mutex(self.a_levels[level])
level += 1
            self.add_literal_level(level)
            self.update_s_mutex(self.s_levels[level])
if self.s_levels[level] == self.s_levels[level - 1]:
                leveled = True
```

The mutex methods are left as an exercise to the reader. Another application of a planning graph is in heuristic estimation. We can estimate the cost of achieving any subgoal from state *s* as the level at which the goal first appears in the planning graph. If we assume that all the subgoals are independent we can simply estimate the total goal cost as the sum of subgoal costs as given in the planning graph. This heuristic would be implemented in the planning graph class as follows:

```
def h_levelsum(self) -> int:
        """The sum of the level costs of the individual goals
(admissible if goals independent)
:return: int
        """
        level_sum = 0
        goals = [PgNode_s(g, True) for g in self.problem.goal]
        # for each goal in the problem, determine the level cost, then
add them together
        for g in goals:
            if g not in self.s_levels[-1]:
                # the problem is unsolvable
                print('Unsolvable')
                level_sum = float('inf')
                break
            else:
                for level, s in enumerate(self.s_levels):
                    if g in s:
                        level_sum += level
                        break
        return level_sum
```

We can call this method within the AirCargoProblem class as follows:

```
def h_pg_levelsum(self, node: Node):
        """This heuristic uses a planning graph representation of the
problem
        state space to estimate the sum of all actions that must be
carried
        out from the current state in order to satisfy each individual
goal
        condition.
        """
        # requires implemented PlanningGraph class
        pg = PlanningGraph(self, node.state)
        pg_levelsum = pg.h_levelsum()
        return pg_levelsum
```

Using this heuristic, we can very efficiently solve complex planning problems using the A* star algorithm. I considered much more complex planning problems with several heuristics and found that the level_sum heuristic significantly outperformed (in terms of time and space complexity) all standard search algorithms including A star with relaxed problem heuristics.

Development 3: Heuristic Search Planner (HSP) (1998)

HSP is based on the idea of heuristic search. A heuristic search provides an estimate of the distance to the goal. In domain independent planning, heuristics need to be derived from the representation of actions and goals. A common way to derive a heuristic function is to solve a relaxed version of the problem. The main issue is that often the relaxed problem heuristic computation is NP-hard.

The HSP algorithm instead estimates the optimal value of the relaxed problem. The algorithm transforms the problem into a heuristic search by automatically extracting heuristics from the STRIPS encodings.

The algorithm works iteratively by generating states by the actions whose preconditions held in the previous state set. Each time an action is applied, a measure *g* is updated, which aims to estimate the number of steps involved in achieving a subgoal. For example, suppose *p* were a subgoal. We initialize *g* to zero and then when an action with preconditions *C = r_1, r_2,…,r_n* is applied, we update *g* as follows:

$$g_s(p) := \min\left[\ g(p)\ ,\ 1 + \sum_{i=1,n} g_s(r_i)\ \right]$$

It can be shown that the procedure explained above is equivalent to computing the function:

$$g_s(p) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } p \in s \\ i & \text{if } [\min_{C \to p} \sum_{r_i \in C} g_s(r_i)] = i - 1 \\ \infty & \text{otherwise} \end{cases}$$

where *C—>P* stands for the actions that assert *p* and have preconditions *C = r_1, r_2,…,r_n*. Then if we let *G* be the set of goal states, the final heuristic function would be as follows:

$$h(s) \stackrel{\text{def}}{=} \sum_{p \in G} g_s(p)$$

Note that we assume that all subgoals are independent and therefore it may be the case that the heuristic is not admissible. This HSP method is useful because it allows us to generalize a heuristic computation to any general STRIPS problem formulation.

Conclusion

The developments discussed in this article constitute 3 major advancements in the field of AI Planning. The STRIPS formulation gave researchers a general framework under which more advanced languages could be built. The Planning Graph construct was a revolutionary data structure which gave a whole new perspective on optimal planning techniques. Finally, the HSP algorithm gives an automated approach for determining heuristics to general planning problems.

References

[1] STRIPS Paper

[2] GraphPlan Paper

[3] HSP Paper