Heuristics and A* implementations

Claudio Scheer
claudio.scheer@edu.pucrs.br
Master's Degree in Computer Science
Pontifical Catholic University of Rio Grande do Sul - PUCRS
Porto Alegre - RS, Brazil

Abstract

AAAI creates proceedings, working notes, and technical reports directly from electronic source furnished by the authors. To ensure that all papers in the publication have a uniform appearance, authors must adhere to the following instructions.

Six domains were tested in the implementations: Blocksworld, Dinner, Dompteur, DWR - Dock Worker Robots, Logistics and TSP - Travel Sales Person. Not all domains were tested on the solver. However, most of the domains were tested on the heuristic functions.

In Heuristics section, I discuss the heuristics implemented. In Plan Validation section, I discuss how a plan is validated. Finally, in the section Solver using A* and h_{max}, I discuss the implementation of the solver. Some performance results are discussed in the Performance section.

Heuristics

In this section, I discuss the different heuristics implemented in the Jupyter notebook. The implementation uses the *pddl* package to parse the tested PDDL domains and problems.

h_{max} heuristic

In a nutshell, this heuristic returns the maximum cost to achieve a goal. From an initial state, the heuristic returns the longest path to reach all goals.

```
1  from pddl.heuristic import Heuristic
2
3  class MaxHeuristic(Heuristic):
4    def h(self, actions, state, goals):
5      reachable = state
6      goals_missing = goals[0]
7      max_cost = 0
8      while not goals_missing.issubset(
       reachable):
9        last_state = frozenset(
10         [a for a in actions if a.
       positive_preconditions.issubset(
       reachable)]
11       )
12       new_reachable = reachable.union([pre
       for a in last_state for pre in a.
       add_effects])
13       if new_reachable == reachable:
14         return float("inf")
15       reachable = new_reachable
16       max_cost += 1
17     return max_cost
```

Listing 1: h_{max} implementation

In the Listing 1, the function *h* returns the maximum cost to reach the *goals* from an initial *state*, considering a set of possible *actions*.

The first reachable states are the initial states, as shown in line 5. The next two lines define the goals[1] and the maximum cost to achieve the goals from the reachable state. Therefore, if all goals are in the initial state, the maximum cost is 0 and the return in line 8 is *False*.

When the goals are not in the reachable state, the algorithm takes two step:

- line 9: get all actions in which the preconditions are applicable to the current set of reachable actions.
- line 12: get the effects from the actions applicable to the current reachable state. Each time the algorithm performs this step, the reachable state becomes larger, that is, it is more likely that the goals are in the reachable state.

Finally, in line 13, it is tested whether the new reachable states are the same as the current reachable state. If *True*, there are no more states to reach and the heuristic has not achieved the goals. Therefore, inf is returned. When there are more states to test, the maximum cost is increased until all goals are reached.

h_{add} heuristic

In a nutshell, this heuristic returns the sum of all the costs to reach the goals. The Listing 2 shows the algorithm that performs this heuristic.

```
1  from pddl.heuristic import Heuristic
2
```


[1]The goals received as a parameter are divided into positive and negative. Negative goals are those with the negative sign (*not*) in the PDDL. In all heuristics, I consider only the positive goals.

```
3  class AdditiveHeuristic(Heuristic):
4    def h(self, actions, state, goals):
5      reachable = state
6      goals_missing = goals[0]
7      goals_reached = None
8      last_state = None
9      add = 0
10     costs = {p: 0 for p in state}
11     while last_state != reachable:
12       goals_reached = goals_missing.
       intersection(reachable)
13       if goals_reached:
14         add += sum(costs[g] for g in
       goals_reached)
15         goals_missing = goals_missing.
       difference(goals_reached)
16       if not goals_missing:
17         return add
18       last_state = reachable
19       for action in actions:
20         if action.positive_preconditions.
       issubset(last_state):
21           new_reachable = action.add_effects
       .difference(reachable)
22           for effect in new_reachable:
23             costs[effect] = sum(costs[pre]
       for pre in action.positive_preconditions
       ) + 1
24           reachable = reachable.union(
       new_reachable)
25     return float("inf")
```

Listing 2: h_{add} implementation

Similar to the h_{max} heuristic, the first reachable state will be the initial state and the cost of reaching goals that are in the initial state, is 0 (line 10). As we need to sum the cost of reaching all goals, it is necessary to maintain a set of all goals that have not yet been achieved.

When a goal is reached in the current reachable state (line 12), the cost of all goals reached is added to the variable *add*, as shown in line 14.

After reaching all the goals, the variable *add* is returned (line 17). If some goal cannot be reached, at some point in the execution, the previous state will be equal to the reachable state, and then return inf (line 25).

To get the next reachable state I need to filter only the actions applicable to the current state and obtain the effects of those actions (line 21). After that, the cost of each effect is calculated and added to the variable *costs* (line 23). The cost of the effect will be the sum of the costs of the preconditions plus 1, because it is the next step in the search tree.

h_{ff} heuristic

The Fast Forward heuristic creates a relaxed plan for the problem, considering only the positives preconditions and positives effects. The cost of the heuristic will be the number of actions included in the plan. The Listing 3 shows the Python script to perform this heuristic.

```
1  from pddl.heuristic import Heuristic
2
3  class FastForwardHeuristic(Heuristic):
4    ...
5    def h(self, actions, initial_state, goal):
6      add = self.build_bs_table(actions,
       initial_state, goal)
7      if add == 0:
8        return 0
9      elif add == float("inf"):
10       return float("inf")
11
12     r_plan = set()
13     actions_already_explored = set()
14     actions_to_explore = []
15
16     for g in goal[0]:
17       actions_to_explore.append(self.
       best_supporter(actions, initial_state, g
       ))
18       actions_already_explored.add(g)
19
20     while actions_to_explore:
21       action = actions_to_explore.pop()
22       if action.name != "nop" and action not
        in r_plan:
23         for precondition in action.
       positive_preconditions:
24           if precondition not in
       actions_already_explored:
25             actions_to_explore.append(
26               self.best_supporter(actions,
       initial_state, precondition)
27             )
28             actions_already_explored.add(
       precondition)
29         r_plan.add(action)
30
31     return len(r_plan)
```

Listing 3: h_{ff} implementation

The function *build_bs_table*, line 6, creates the best support table used to get the action closest to a specific action. This same function returns zero when the goals are in the initial state and inf when the goals are unreachable.

If the goals are reachable, the algorithm get the actions that best support the goals and creates a list structure (line 14). I also track all actions that have beed already explored, avoiding repeating actions.

In line 21, I pop an action that was in the list and append to the list the best supported action of its preconditions. In some cases, the action does not have the best supported action. This is represented by an action named ”nop”. Therefore, these actions are ignored (line 22). If the action has not yet been added to the plan, it will be added to the plan (line 29) and the algorithm will continue searching (line 20) for the best supported actions until all actions are processed.

In the end, line 31, the algorithm simply returns the plan length.

Plan Validation

In some scenarios, it is necessary to validade whether a given plan is valid or not. Listing 4 shows a Python code for performing plan validation.

```
1  def validate(self, actions, initial_state,
       goals, plan):
2    state = initial_state
3    for line in plan:
4      for action in actions:
5        if line.parameters == action.
       parameters:
6          if applicable(
7            state, (action.
       positive_preconditions, action.
       negative_preconditions)
8          ):
9            state = apply(state, (action.
       add_effects, action.del_effects))
10           break
11
12   goals_reached = goals[0].intersection(
       state)
13   return goals_reached == goals[0]
```

Listing 4: Plan validation implementation

The function *validate* takes as parameters the actions that can be applied to the state, the initial state, the goals and the plan to be validated. An example of a plan to be validated is shown in Listing 5.

```
(take k1 cc cb p1 l1)
(load k1 r1 cc l1)
(move r1 l1 l2)
...
```

Listing 5: Example of a plan

The main idea of a plan validation is to apply each line of the plan to the state and test whether the goals have been reached or not. In lines 3 and 4 of Listing 4, I search for the action that is applicable to the current line of the plan. When the plan line and an action have the same parameters (line 5), I apply the action on the state, interrupt the search for another action and move to the next plan line.

After all the effects of the plan are applied to the state, I search in the state for the goals. If all goals can be found in the state, the plan is valid.

Solver using A* and h_{max}

A solver is responsible for finding the best path to achieve the goal. We can consider the best path as the path that performs the least actions to achieve all goals. However, the path found can be a local minimum result, that is, there is a better path but the solver cannot see it. The solver would need to keep searching to find that path. This algorithm just searches for a path that reaches all goals, not necessarily the global minimum path.

Hence, in this section, I will show how the A* differs from the Dijkstra algorithm and also an algorithm to search for the shortest path using A* search, guided by the h_{max} heuristic.

Dijkstra and A*

A* is based on the Dijkstra algorithm. In a nutshell, Dijkstra's algorithm searches for the node in the tree that has the lowest cost. The Figure 1 helps to understand the Dijkstra's algorithm. Our goal is to be in city E, starting from city A.

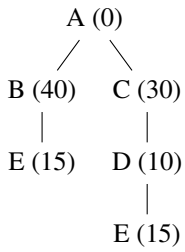

Figure 1: Dijkstra search tree

The distance between cities A and B is 40 kilometers. Therefore, Dijkstra's algorithm will choose the path from city A to C, since the distance is shorter. In the next step, the algorithm have two possible nodes to expand: A or D. The cost to travel to each of these cities will be the current cost, 30 kilometers, plus the cost to reach A or D. Therefore, the cost to reach cities A or D is 60 and 40, respectively. Now the algorithm has three options: 1) go back and, instead of city C, choose city B, which costs 40; 2) go to city A, which costs 60; 3) go to city D, which costs 40. The algorithm will choose the cheapest path. In this case, let's assume that it arbitrarily chooses B. Following the same logic, the algorithm will have to expand all the nodes until reaching the goal.

To solve this problem, the A* algorithm adds a heuristic value to the cost of moving from a city to another, for example. The heuristic may be the straight line distance between two cities. Now the algorithm can know, for example, that the path from city A to C is longer to reach the goal than from city A to B. A bad heuristic will guide the A* algorithm to the wrong direction. However, with good heuristics, A* algorithm needs to expand fewer nodes to reach the goal.

Solver Explanation

The solver uses the A* algorithm, guided by the h_{max} heuristic. To implement the search, I used the *queue* and the *pddl* packages. The first was used for the priority queue[2] and the second for parsing the PDDL domain file and problem file.

I need to store five pieces of information during the search process:

- **state**: the state of the problem at a specific point. When an action is applied to the state, it creates a new state that is added to PriorityQueue;
- **priority**: the priority of obtaining a specific state in the next exploration. The priority is the result of the cost of the state plus the heuristic value;
- **cost of the state**: the cost to reach a specific state in the tree. This value is the same as the depth of the state in the tree. The cost of a new state is the cost of the previous state plus 1;

[2]PriorityQueue, provided by Python, always returns the value with lowest priority first. If the priority is the same, it will return, in my tests, the element added first to the queue.

- **parent state**: the state that generate a new state. This information is used to search backward the actions taken to achieve the goals;
- **actions applied**: this is the actions applied to generate a specific state.

The next step in the algorithm is to explore the state with lowest priority. The actions whose preconditions are applicable to the current state are applied to the state and this information is stored in PriorityQueue for the next exploration. States whose heuristic value is $\inf$ are ignored.

After all goals are reached, the algorithm searches backward in the tree and returns the actions that need to be applied to reach the goals.

Performance

The heuristic used in the solver can influence in the execution time. I tested the solver using the h_{max} and h_{ff} heuristics. The execution time is much higher when using h_{max}. For example, when solving the problem 7 of the *blocksworld* domain, the time using h_{max} was 36.36 seconds. When using h_{ff}, the time was 0.07 seconds.

When testing the same problem in Web Planner, the time was 1.66 seconds. When comparing the actions required to solve a problem, the two planners show the same actions as results.

Conclusions

Some problems of the solver was discussed previously. However, just to emphesize, the A* implementation in my solver can be considered as a *vanilla* implementation. The main problem I had to implement the heuristics was to translate the mathematical formulas into the Python scripts.

When analysing the performance of the heuristics implemented, the h_{ff} heuristic has the best time performance. In addition, h_{max} becomes worse in time consuming, as the problem increases.