Intro to AI Project 1: Search

吴嘉骜 21307130203

2023 年 10 月 10 日

1 Search Algorithms

Each search algorithm of this project (DFS, BFS, UCS, A*) is similar in framework. The main difference is the details of how the fringe is managed. The fringe tracks all the nodes we have generated but not yet expanded. To implement a graph search algorithm, we need to record the states we have visited. The following is the pseudo code of the search algorithm.

Algorithm 1: Graph Search algorithm

```
Input: Problem
Output: A solution or Failure
1  Choose an appropriate data structure for frontier and visited
2  Initialize frontier using the initial state of Problem
3  Set visited to empty
4  while frontier is not empty do
5      Pop a node node from frontier according to some rule
6      if node contains a goal state then
7          return Solution
8      end
9      if node is in visited then
10         Continue
11     end
12     Add node to visited
13     Expand node to get the set of state successors S
14     Add all unvisited s in S to frontier
15 end
16 return Failure
```

We define a data structure **Node** ahead of the algorithms to represent a node in the search tree with four components:
state: the state to which the node corresponds;
parent: the node in the tree that generated this node;
action: the action that was applied to the parent's state to generate this node;
path_cost: the total cost of the path from the initial state to this node.
Also, we need a data structure to store the *frontier*, which depends on the search algorithm itself. We

use ***Stack***, ***Queue*** and ***PriorityQueue*** which are predefined in *util.py*.
We also use *visited* to store the states we have visited. Here it refers to the states of the expanded nodes. We check a node whether it is in *visited* double times, first when we pop a node from $frontier$, and second when we add the node to $frontier$. This can help us to do less redundant work.
Later in UCS, where we need to update the path cost of the nodes in the $frontier$ when we find a better path to the node, we change the *visited* set to *reached* dictionary to store the mappings from states to nodes with lowest path cost. We check every newly-generated node, whether or not it has been expanded.
Although algorithm frameworks are similar, there are still some details that differ in the implementation of each algorithm, which we will specify as follows.

Question1: Depth First Search

In DFS, we use ***Stack*** to store the nodes in the $frontier$. The ***Stack*** is a LIFO data structure, satisfying the principle of DFS. The path cost is not considered, and we set it to 0. DFS cannot guarantee the optimality of the solution, but can save a lot of memory. On meduimMaze, DFS expanded 130 nodes and scored 380 with total cost 130, returning a very long path.

Question2: Breadth First Search

In BFS, we use ***Queue*** to store the nodes in the $frontier$. The ***Queue*** is a FIFO data structure, satisfying the principle of BFS. The path cost is also not considered. In the implementation, we check whether a node is a solution as soon as it is generated rather than until it is popped from the $frontier$. BFS can guarantee the optimality of the solution, but it will expand a lot of nodes and consume a lot of memory. On meduimMaze, BFS expanded 267 nodes and scored 442 with tital cost 68.

Question3: Uniform Cost Search

In UCS, we use ***PriorityQueue*** to store the nodes in the $frontier$. The ***PriorityQueue*** is a data structure that pops the node with the lowest path cost, satisfying the principle of UCS. The path cost is of course considered. We use the *path_cost* of the node as the priority of the node in the ***PriorityQueue***. To make sure that the state with the lowest path cost is popped first, we update the node in the $frontier$ when we find a better path to the state, as recounted in the previous paragraph. UCS can guarantee the optimality of the solution, scoring 442 with total cost 68 and 269 expanded nodes on meduimMaze.

Question4: A* Search

In A*, we also use ***PriorityQueue*** to store the nodes in the $frontier$. The $Node$ class has a new attribute $heuristic_val$, which is the heuristic value of the state. A* considers the priority of the node in the ***PriorityQueue*** as the sum of the *path_cost* and *heuristic_val*. With an admissible and consistent heuristic, A* can guarantee the optimality of the solution. So we do not need to update the node in the $frontier$. On meduimMaze, A* expanded 221 nodes and scored 442 with total cost 68.
We can see that on openMaze, BFS, UCS and A* all find the best path, while DFS finds a very long path but with less nodes expanded.

2 Corners Problem

Question5: Finding All the Corners

The goal of corners problem is to find the shortest path through the maze that touches all four corners. Therefore, we need to define a new state representation. The state space consists of a tuple (*position* and *corner_state*), where *corner_state* records whether four corners have been visited in a dummy tuple - 0 means no and 1 yes. Thus, the **startState** is (*start_position*, (0, 0, 0, 0)) and the **goalState** is that the sum of corners tuple equals 4. To **getSuccessors**, we check whether the next position is a corner. If so, we change 0 corresponding to this corner to 1 and leave the rest unchanged; otherwise retain the *corner_state*.

Question6: Heuristic

Consider a relaxed problem: find the shortest path that touches all four corners, and we can move through walls (but still in four directions). A solution is that we move to the nearest untouched corner sequentially. An algorithm to find the solution is as follows:

Algorithm 2: Corners problem heauristic

```
  Input: $Problem, State$
  Output: $Heauristicvalue$
1 $(curr\_pos, cornerState) \leftarrow startState$
2 Initialize $heauristic$ to 0
3 while $there\ exist\ unvisited\ corners$ do
4     Find the unvisited corner $c$ that has the smallest Manhattan distance from $curr\_pos$
5     Move to $c$, update $curr\_pos$ and $corner\_state$
6     $heauristic \leftarrow heauristic + distance(curr\_pos, c)$
7 end
8 return $heauristic$
```

Analysis of Heuristic

Admissiblity: the heauristic function value is a solution of the relaxed problem, so it is a lower bound of the cost of the original problem, then it is admissible.

Consistency:

Consider a node n and its successor n'. Suppose the closest corner to n is c, and the closest corner to n' is c'. We use $d(n, c)$ to denote the Manhattan distance from n to c.

Then it is easy to verify that from n to n', $d(n', c') = d(n, c') \pm 1$ for a fixed c'.

Thus, $h(n) - h(n') = d(n, c) - d(n', c') \leq d(n, c') - d(n', c') \leq 1 = cost(n, n')$. Note that the cost of every movement is always 1, so the heauristic is consistent.

In the implementation, we use the ***Counter*** data structure defined in *util.py*, which is an extension of ***dict***. We use it to record the mapping from every unvisited corner to the Manhattan distance between it and the current position.

On mediumCorners, A* search with this heauristic expands 692 nodes and scores 434 with a total cost 106, while BFS expands 1921 nodes with the same optimal path. We can see that heuristics (used with A* search) can reduce the amount of searching required.

3 Food Search Problem

Question7: Eating All The Dots

The goal of the food search problem is to find the shortest path that collects all of the food in the Pacman world. To devise an admissible heuristic, we consider a relaxed problem: find the shortest path to the farthest food (i.e. with the max shortest path cost among all food dots). A solution is calculating every food dot's real distance (shortest path cost) from the current position by A* search, and assign the maximum to the heauristic value. A detailed algorithm is as follows:

Algorithm 3: Food search problem heauristic

```
Input: $Problem, State$
Output: $Heauristicvalue$
1  $(pos, food\_grid) \leftarrow startState$
2  Initialize $heauristic$ to 0
3  for $every\ food\ dot\ food\ in\ food\_grid$ do
4      $searchProblem \leftarrow$ a search problem with $pos$ as the start state and $food$ as the goal
5      $distances \leftarrow astarSearch(searchProblem)$
6  end
7  $heauristic \leftarrow max(distances)$
8  if $all\ food\ have\ been\ eaten$ then
9      return 0
10 end
11 return $heauristic$
```

Analysis of Heuristic

Admissiblity: the heauristic function value is a solution of the relaxed problem, so it is a lower bound of the cost of the original problem, then it is admissible.

Consistency:

Consider a node n and its successor n'. Suppose the farthest food to n is f, and n' is f'. We use $p(n, f)$ to denote the real path distance from n to f.

Then $n \to n' \to \cdots \to f$ is a path with cost $p(n', f) + 1$, so $p(n, f) \leq p(n', f) + 1$ because $p(n, f)$ is the shortest path distance. Thus, $h(n) - h(n') = p(n, f) - p(n', f') \leq p(n, f) - p(n', f) \leq 1 = cost(n, n')$. Note that the cost of every movement is always 1, so the heauristic is consistent.

On trickySearch, A* search with this heauristic expands 4137 nodes and scores 570 with a total cost 60.

4 Closest dot Search

Question8: Suboptimal Search

This problem goal is to eat the closest dot. So **isGoalState** should return true when the current position is a food dot and false otherwise. As BFS always finds the closest node which achieves the goal, the food returned by BFS for the first time should be the nearest dot. So in **findPathToClosestDot** we just return the path given by BFS.

On bigSearch, suboptimal search scores 2360 with a total cost 350.

Suboptimal search does not always find the optimal solution. Consider an example where there are 4 food dots $A(0, 0), B(3, 0), C(3, 1)$ and $D(3, 2)$. The initial position is $P(2, 0)$. Then the suboptimal search gives $P \to B \to C \to D \to A$, which takes 1+1+1+5=8, but the optimal path is $P \to A \to B \to C \to D$, which takes 2+3+1+1=7.