Report for PJ1

Yuze Ge 19307130176

October 2021

1 Search Algorithm

DFS,BFS,UCS,A*,the four search algorithms have the same framework.(The difference is only in the way the node is popped from the fringe)The basic process is as follows (all application diagram search):

Algorithm: `Graph-Search`

Input: *Problem*
Output: A *solution* or **Failure**
Choose the right data structure as the *fringe*.
Initialize the *fringe* with the initial state of the problem.
Set *explored* to empty
while *fringe* is not empty **do**
  Pop a node *s* from the *fringe* according to a certain rule
  if *s* is the goal **then**
    Return *Solution*
  end
  if *s* in *explored* **then**
    Skip this node and continue the loop.
  end
  Add *s* to *explored*
  Join the successor nodes of *s* to the *fringe* if and only if these nodes are not in *explored* or *fringe*
end
if *fringe* is empty **then**
  Return Failure
end

Data structure in code implementation:
node: I defined it before all search algorithms.It contains four attributes: state, parent, path_cost, action.
state: The state of the node in the state space
parent: The node that produced the current node (that is, the parent node)
path_cost: The action path cost from the parent node to the node
action: The action taken when the parent node generates the node
fringe: Store nodes that have been observed but not visited. Use stack, queue, priority queue .Pop up nodes in the way required by the algorithm.
explored: Store the nodes that have been visited to ensure that they will not be visited repeatedly.

Note: In the project code, the implementation of the search algorithm roughly follows the above process. Different algorithms may not be completely consistent with the above process, (such as swapping the positions of target detection and repeated node detection).This has almost no effect. For the detection of duplicate nodes, I tested both in fringe.pop and expand. This can ensure that the node will not be visited repeatedly, and the visited node will not enter the fringe.

In DFS, the ***fringe*** uses the ***Stack*** data structure, which always pops up the last entered node first. DFS cannot guarantee the optimal solution, because it always explores the deepest node first. For example, DFS scored 380 in mediumMaze, while BFS, UCS, A* scored 442.

In BFS, the ***fringe*** uses the ***Queue*** data structure, which always pops up the first entry node first. BFS can guarantee to find the optimal solution.

In UCS, the cost of different paths is no longer the same. ***fringe*** uses a ***Priority Queue*** as a data structure, and the priority is the total cost from the initial node to the current node. ***fringe*** pops the node with the least cost each time. UCS guarantees to find the optimal solution.

In the A* search, I redefine the node and add a new attribute ***heuristic_cost***, which represents the heuristic value of the node. ***fringe*** uses a ***Priority Queue***, the priority is ***path_cost***(total path cost) +***heuristic_cost***.
A* guarantees to find the optimal solution, as long as the heuristic function is acceptable and consistent (such as Manhattan distance). In openMaze, A* expanded 535 nodes and UCS expanded 682 nodes. In addition, in openMaze, BFS, UCS, and A* can all find the best path; while DFS always searches along the deepest path and finds a very long path.

2 Corner Problem

Question5: Finding All the Corners

In the corner problem, Pac-Man needs to eat the food in the four corners. Therefore, the state of the node includes two parts: the position and the state of the food in the four corners. The state of the food is made up of a tuple containing four elements. Each element is 0 or 1, 0 indicates that the food has not been eaten, and 1 indicates that the food has been eaten.
Startstate: (0,0,0,0)
GoalTest: Is the sum of the food tuple elements equal to 4
Successors: If the position of the successor node is the food position, change the element of the corresponding position in the tuple from 0 to 1. Otherwise, the tuple remains unchanged.

Question6: Corners Problem: Heuristic

We envision a **simplified problem**: Pac-Man needs to eat food in the four corners from the initial node (the food position remains unchanged), and there are no walls on the map.
Solution: We first reach a goal that is the shortest Manhattan distance from the current position among all goals, and we use the goal as our current position.Repeat the above steps until all goals are accessed.Cost is the sum of all the minimum Manhattan distances in the above process.

Algorithm: `cornersHeuristic`

Input: *Problem, State*
Output: Heuristic function value(Non-negative)
curpos ←initial position
heuvalue ← 0
while There is food that has not been eaten **do**
  Find the uneaten food that has the smallest Manhattan distance from your current location.
  curpos ←the food position
  heuvalue ←heuvalue + the smallest Manhattan distance
  Mark the food as eaten(Only this time when the function is running)
end
Return *heuvalue*

Acceptability The heuristic function is acceptable, because the above problem is obviously a relaxation of the original problem.

Consistency

We first prove that if the consistency is satisfied for any node A and its successor A', then A and any node B also satisfy the consistency.
Suppose a path from A to B is $(X_1, X_2..., X_n)$, $X_1 = A, X_n = B$
for any $i \in \{1...n-1\}$ satisfy

$$h(X_i) - h(X_{i+1}) \leqslant cost(X_i, X_{i+1})$$

Add them up

$$\sum_{i=1}^{n-1} h(X_i) - h(X_{i+1}) \leqslant \sum_{i=1}^{n-1} cost(X_i, X_{i+1})$$

$$h(A) - h(B) \leqslant cost(A, B)$$

This proves that A and B meet consistency.
Suppose B is the successor node of A. We only need to prove that A and B satisfy the consistency in this relaxed problem.
Every time Pac-Man moves, the path cost is 1. So $cost(A, B) = 1$
Suppose the closest food(Manhattan distance) to A is M and the closest food(Manhattan distance) to B is N

$$Manhattan(A, M) \leqslant Manhattan(A, N)$$

Note that for each step Pac-Man moves, the Manhattan distance to a food will either increase by 1, or decrease by 1.

$$Manhattan(A, N) \leqslant Manhattan(B, N) + 1$$

So

$$h(A) - h(B) = Manhattan(A, M) - Manhattan(B, N) \leqslant 1$$

$h(A) - h(B) \leqslant cost(A, B)$ This proves the consistency.

3 Food Search Problem

Question7: Eating All The Dots

Consider this question: Keep the food with the longest actual distance from the current node, and remove other foods.
Pac-Man should follow the maze path from the current node to the food.
We calculate the actual distance from the current location to all foods, and use the largest actual distance as the heuristic function value. I used A* in the project code to calculate the actual distance. First, function PositionSearchProblem is called with the current node and food as start and goal as parameters. And call the A* function in search.py.

Algorithm: `foodHeuristic`

Input: *Problem ,State*
Output: Heuristic function value(Non-negative)
if *The current state is the target state* **then**
  Return 0
end
Calculate the actual distance from the current node to all foods.
Return Maximum actual distance

Acceptability The above problem is obviously a relaxation of the original problem, so the heuristic function is acceptable.
Consistency
Consider the current node A and its successor node B. From the analysis of the previous question, $cost(A, B) = 1$.
M is the node with the largest Manhattan distance from A . And N is the node with the largest Manhattan distance from B. So $Manhattan(B, N) \geqslant Manhattan(B, M)$
B moves only once relative to A, so $Manhattan(B, M) \geqslant Manhattan(A, M) - 1$
$h(A) - h(B) = Manhattan(A, M) - Manhattan(B, N) \leqslant cost(A, B) = 1$.And this proves the consistency
Actually this proof is almost the same as the proof in the previous question.

Question8: Suboptimal Search

BFS always visits the node closest to the current node, and the food returned by BFS for the first time must be the food closest to the current node. So in AnyFoodSearchProblem, isGoalState should be set to return True once a food is found.
Suboptimal search does not always find the optimal solution. For example, in figure1, the food on the left is two units away from Pac-Man, and the food on the right is one unit away from Pac-Man. In suboptimal search, Pac-Man will first eat all the food to the right and then the food on the left, but the optimal path is to eat the food on the left first and then the food on the right.

Figure 1: Suboptimal search does not always find the shortest path