FIT5047

Assignment 1: Search

Ling Qin

35366478

03/04/2025

table of contents

Part 1: Single Agent Search

Question 1(a):

For this part, it gives us a maze, a start location and a destination, it's just a simple A* algorithm program with Manhattan distance. The key point is how to make a great structure for this program with the functions that are given to us and how to return the result so that it can be perfectly shown to us.

In q1a_problem.py, the class q1a_problem stores basic information about this question. In the getStartState function, we get information about the maze, including start location, walls, and destination. In the isGoalState function, we use it to know whether our current location is our destination. In the getSuccessors function, we get all the possible legal successors of our current location.
In q1a_solver.py, this is the process of solving. There are two classes, AstartData is to store openlist and closelist in A* search. Successornode is to store node information, which would be put into openlist and closelist. Information includes location of the node (x and y), cost to get to the node, the heuristic distance to the destination, and total cost. Besides, the is an actionlist in every successornode class, which stores the action to get to the node from the start.

For A* search f(n) = h(n) +c(n), h(n) is the heuristic distance from current node n to destination, in this question, we use c(n) as the cost from start to get to current node n. In every loop, we choose the node with the lowest f value in openlist and put it into clsoelist.

To return the result, when we add a node to openlist, it's actionlist equals its father node actionlist plus the move from the father node to it, and the father node must be the last one in closelist.

Pseudocode:

```
get maze information(walls, start_position, destination)
initialize A* openlist and closelist
put successornode(start) into closelist

astar_loop_body
    get last one in closedlist as current state
    get current_result
    if isGoalState:
        return current_result and the actionlist of last
        one in closelist
    else:
        get successors of last one in closelist
        if it’s not in closelist:
            calculate heuristic distance, cost and
            value of function f
            put it into openlist

    sort the open list by it’s f value
    put the first one of openlist into closelist
    pop the first one of openlist

    return current_result none
```

Part 1: Single Agent Search

Question 1(b):

For this part, we have to collect one dot among multiple dots, and minimising the total number of node expansions.

There are two key points about minimising the total number of node expansions.

First, I want to know the nearest dot. If I use Manhattan distance, it's not perfect because there are walls in the maze, so I use BFS to get the nearest dot. From the start location, I use BFS to look for dots. The first one we meet must be the nearest one, at least one of them.

Second, if we just compare the f value in openlist, it can still expand a lot of meanless nodes because many nodes have the same f value, so I think after we sort the f value, we should get smaller h value among same f value node beacuse it means it's closer to destination.

And rest part of this question is just like question1(a).

Pseudocode:

```
get maze information(walls, start_position, food_locations)
BFS for the nearest food and set it as destination
initialize A* openlist and closelist
put successornode(start) into closelist

astar_loop_body:
    get last one in closedlist as current state
    get current_result
    if isGoalState:
```

```
        return current_result and the actionlist of last
        one in closelist
else:
        get successors of last one in closelist
        if it's not in closelist:
                calculate heuristic distance, cost and
                value of function f

                put it into openlist
sort the openlist by it's f value
put the first one of openlist into closelist
pop the first one of openlist

return current_result none
```

Part 1: Single Agent Search

Question 1(c):

For this part, we need to collect as many dots as possible to maximise the score.
In the beginning, I thought I could calculate the shortest distance between each dot and the start position, and this problem turned into how to find a subsequence of all the dots to maximise the score. It's clearly a DP problem and I can get the perfect solution. Use BFS to get the shortest distance between each dot and the start position, and DP State Transition is:
$dp[S \cup \{i\}][i]=\max(dp[S \cup \{i\}][i],dp[S][j]+10-dist(j,i))$

Let dp[S][i] represent the maximum score that can be obtained from the visited node set S, which currently stops at i

When I finished it and locally run program, I found that it ran out of time on many mazes, after I checked the maze, a lot of them had more than 30 dots, and the biggest one had more than 200 dots, clearly DP would run out of time, not mention there are BFS to calculate the distance, so it's a dead end.

Greedy algorithm is the first thing came up my mind, I look this problem as many question1b, in every loop I set start location and destination, the destination is the nearest dot just like question1b, and I get the actionlist to get to the destination.

The key point is how to determine loop termination. First, I ran a BFS throughout the whole map to get the number of all the achievable dots, if the time of loop beyond that,

then it's the end. Besides, if the current shortest cost to get a dot is larger than the sum of achievable score, then this is the end.

However, I made a mistake about BFS, which made my program run out of time on two mazes. At first, I only put a node into visitedlist when I actually popped it from the queue, but it is very inefficient cause many meaningless nodes were put into queue, which caused runtime error, so I put a node into visitedlist when I put it into the queue, and it fixed my problem.

Maybe there are more useful limits, but I got full score on this question so I didn't dig into it.

Pseudocode:

```
get foodNum
BFS get achievable node number
finalreward = 500
if achievable node<foodNum:
        finalreward = 0
final_result =[]
flag = 0

while 1:
        flag+=1
        if flag > achievable node number:
                return final_result
         result = A*(current)
        if nearest food cost > rest achievable node*10 +
        finalreward:
        return final_result
        final_result = finalresult +result
return final_result
```

A*
get maze information(walls, start_position, food_locations)
BFS for the nearest food and set it as destination
initialize A* openlist and closelist
put successornode(start) into closelist

astar_loop_body:
    get last one in closedlist as current state
    get current_result
    if isGoalState:
        return current_result and the actionlist of last one in closelist
    else:
        get successors of last one in closelist
        if it's not in closelist:
            calculate heuristic distance, cost and value of function f
            put it into openlist
    sort the openlist by it's f value
    put the first one of openlist into closelist
    pop the first one of openlist

    return current_result none

Part 1: Single Agent Search

Question 2:

For this part, there are ghosts in the maze, so we have to keep away from the ghost and get as much score as we can.

At first, I just wrote a simple alpha-beta for this and used the current score to compare in the algorithm. This one got about 30 scores, but I made I mistake, I thought I could just add some constraints after I compared the score and I can get a better score, which is a wrong direction, after I add some if for my program, my score was even lower.

For the loop of alpha-beta, I just used the material in the class slide.

So the key point for this algorithm is to make an efficient evaluation function for every end status in alpha-beta algorithm.

The first thing is win or lose, the score of lose should be very low, like -1000000 or something, and we should know how many steps to get the lose, cause if I have to lose, I want to lose as late as I can. For the win, it can be represented by the current score, cause If I win, I get an extra 500, so it must be larger than other situations. But I don't want to reach the win when there still has capsule, cause after the Pacman eats the capsule, it can eat the scared ghost to get a much more score.

Then, for the other part, there are two kinds of situations, whether there are scared ghosts in the maze.

If there are scared ghosts, we should approach them.

If there has no scared ghost, there are two target should be consider, the shortest distance to food and capsule, it's more important to get closer to the capsule than the food, cause after Pacman eat a capsule, it can eat scared ghost to get much more score than just eat food.

Here is the core code of my evaluation function.

```
        if gameState.isLose():
            score = -9999999-depth
            return score
        if gameState.isWin() and len(capsules)>0:
            score = -1000
            return score
        if scared:
            min_scared_dis = BFS_scared(gameState)

            plus = plus-min_scared_dis*100

        if scared == False :

            min_food_dist,min_capsule_dist = BFS(gameState)

            plus = plus - 100*len(capsules)-
        (min_food_dist+5*min_capsule_dist)

score = score * 1000000 + scared*100000 + plus
return score
```

Why do I give score a weight like 100000? That's because when I debug my program, I would print the final score,

and I want to know every part of the calculation, in this way, different bits of the number stands for a different part of the calculation, for example, number 191xxxxxx, 19 means current score, 1 means it has scared ghost, xxxxx means the result of plus, it's much easier for me to debug.

Besides, I use another trick when I code the alpha-beta part. When the program goto alpha-beta, I give it index 0, which represent Pacman, then, when I recursive call alpha-beta, I set the new_index = (index+1) % num_of_agent, because there are 1 pacman and n ghosts, so the index of the agent should be from 0 to n-1.

Pseudocode:

```
final_move =None
max_value = -inf

for move in getlegalactions:
        new = getsuccessor(0,move)
        value = alpha-beta(new,depth,index,alpha,beta)
        if value > max_value:
                max_value = value
                final_move = move


max_eval = -10000
min_eval= 10000
beta = inf
alpha = -inf
alpha-beta(state,depth,index,alpha,beta)
        if depth==0 or win or lose:
        return evaluate(state)
        if index stand for pacman:
                get all successor node
```

```
            eval = alpha-beta(successor node)
            max_eval = max(eval,max_eval)
            alpha = max(alpha,eval)
            if beta<=alpha:
                  break
            return max_eval



      if index stand for ghost:
            get all successor node
            eval = alpha-beta(successor node)
            min_eval = min(eval,min_eval)
            beta = min(beta,eval)
            if beta<=alpha:
                  break
            return min_eval


evaluate(state)

      return Evaluation score
```

Summarize

In question1 a, I learned how to implement the basic A* algorithm。

In question1 b, I learned how to make A* more efficient by reducing the expanded node.

In question1 c, I learned when we design algorithms, we must combine the problem with the data size. An algorithm may be perfect in theory but very inefficient at large data scales.

In question2, I learned how to implement the basic alpha-beta algorithm, and the key point for it is how to design an efficient evaluation function。