FIT3080:Artificial Intelligence

Assignment 1: Search

Lim Zheng Haur

32023952

3rd September 2023

Table of Contents

Part 1: Single Agent Search

Question 1(a):

Implementation:

In Question 1(a) problem, I have to solve for a path from Pacman starting position to the food in the map. To achieve this, I have decided to implement a heuristic search. From my starting point, I expand the map into a priority queue where the priority is the heuristic value for that new position. This heuristic function is a Manhattan distance function which calculates the Manhattan distance from the position to the food. The nearest position will have priority in the queue and the Pacman position is expanded until the food is eaten by Pacman. When the food is found, then the path to the food is backtracked through the explored list and returned.

Limitations:

There is some instances where the Manhattan distance might not the most accurate heuristic function as this does not account for walls. This optimal solution might require Pacman to take a detour around the wall but instead the heuristic search will tend to make Pacman walk to the direction of the food despite there being an obstacle in that direction.

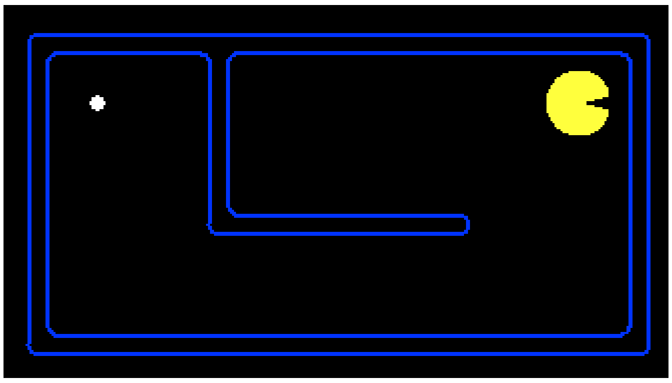

Diagram 1.1

A map with Pacman and a food

Diagram 1.1 shows a map of Pacman where heuristic searching with Manhattan distance may not be optimal. The optimal solution for this map would be “South, South, South, West (x8), North (x4)”. Using Manhattan distance on this map would tend to make Pacman move to west which is a dead end which is not optimal.

Heuristic function:

The heuristic function for this algorithm is the Manhattan distance from Pacman to the food. This heuristic function is chosen mainly due to its simplicity and low complexity.

Justification:

This method was chosen as compared to a breath-first search is due to the computational time since all positions are equally expanded which most of the time might be unnecessary. Although there might be sub-optimal solutions as described above in Diagram 1,1, these situations are rather niche, and the computational speed of heuristic search significantly outperforms the path cost gains of breath-first search. Even conducting tests on the layouts provided, heuristic search consistently only has unsignificant additional path costs with some outliers of minor additional path costs. Therefore, heuristic search was implemented for this question due to the computational time limit of 1 second.

Pseudocode:

```
function q1a_solver (problem)

    frontier <- a Priority Queue
    explored <- empty list
    push the problem start state into frontier
    current <- None
    exploredMap <- matrix of size of map height x width initialized all to false
      food <- problem food coordinate (x, y)

    while frontier is not empty and current is not goal state:
        current <- Pop(frontier)
        explored <- Append(current)
        exploredMap[Pacman X position][Pacman Y position] <- true

        for successor in current state successors:
            if successor Pacman position is false in exploredMap:
                frontier <- Push((successor, current state), ManhattanDistance((successor
Pacman position, food))
```

```
backtrack explored list to find solution
return solution
```

Question 1(b):

Implementation:

In Question 1(b) problem, I have to solve for the most optimal path for Pacman to eat all foods in the map. The solution for this problem is similar to the solution from Question 1(a). However, since there are more than 1 food in the map, the heuristic function is modified to suit this question. The solution will compute at every successor game state the nearest food to Pacman current position for the heuristic calculation.

Limitations:

This algorithm is greedy in a way that it will only target the closest food from its current position. There exist certain scenarios where the algorithm would not be optimal.

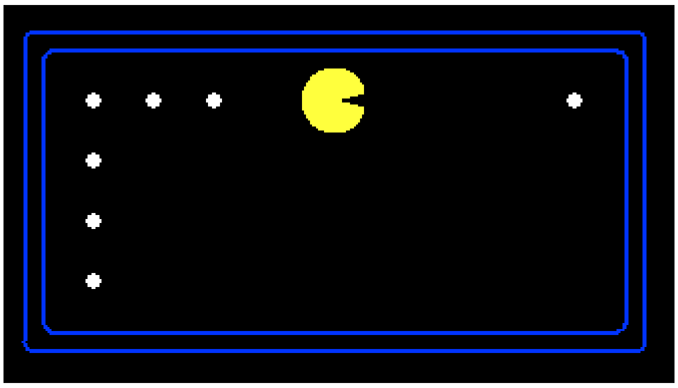

Diagram 1.2
A map with a greedy bait for Pacman.

Diagram 1.2 shows a map where this greedy algorithm would not produce the optimal solution. The most optimal solution would be for Pacman to eat the food at the top-right corner then the rest of the food. However, since this algorithm is greedy and targets the closest food from Pacman current position, it will be inclined to eat the group of food on the left and lastly eat the food on the top-right corner. This solution would be suboptimal.

Heuristic function:

The heuristic function for this is again the Manhattan distance from Pacman current position to the nearest food to Pacman current position. This heuristic function is again chosen due to its simplicity and effectiveness.

Justification:

The justification for this question would be the same as Question 1(a). Since the time constraint in this question is only 5 seconds, breath-first search would take up too much computational time and a heuristic search would be much more efficient to finish computation within the time constraint. The greedy algorithm is also adopted as the layouts in for this question only has 4 food each. Therefore, a greedy algorithm would only affect the first part of the search from the Pacman start position to its first food and the rest of the path would generally be a constant.

Pseudocode:

```
function q1b_solver (problem)

    frontier <- a Priority Queue
    explored <- empty list
    push the problem start state into frontier
    current <- None
    exploredMap <- matrix of size of map height * width with all values initialized to
false
    foodLeft <- problem start state number of food left

    while frontier is not empty and current is not goal state:
        current <- Pop(frontier)
        explored <- Append(current)
        exploredMap[Pacman X position][Pacman Y position] <- true

        if current state number of food == 0:
            break loop
        if current state number of food < foodLeft:
            reset exploredMap to false
            reset frontier priority queue
        if current state number of food <= foodLeft:
            food <- nearest food to current Pacman position by ManhattanDistance

        for successor in current state successors:
            if successor Pacman position is false in exploredMap:
                frontier <- Push((successor, current state), ManhattanDistance(successor
Pacman position, food)

     backtrack explored list to find solution
    return solution
```

Question 1(c):

Implementation:

In Question 1(c) problem, I have to solve for the most efficient path to eat all food in the map. The implementation for this question is the A* search algorithm. The general algorithm of the search is similar to heuristic search, but the heuristic value is calculated differently. The f-score for A* search algorithm in this question is the sum of g-score which is the current path cost and the h-score which is the current distance from Pacman to the nearest food.

Limitations:

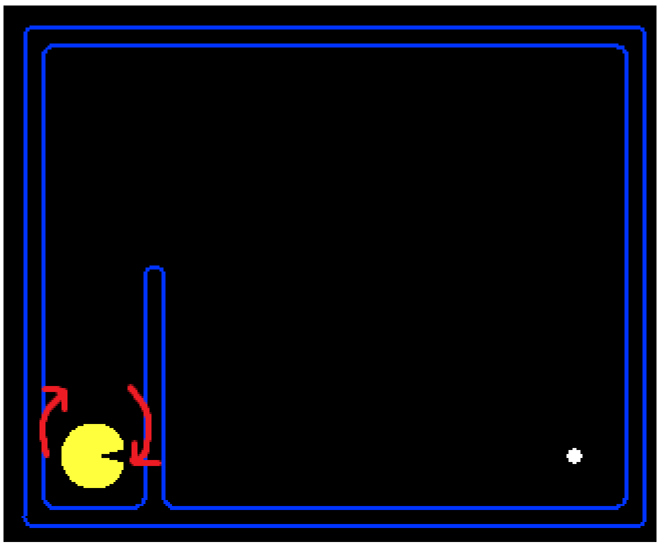

Diagram 1.3

Pacman in a cycle moving from 1 best position to another.

Since the f-score is mostly affected by the Manhattan distance from Pacman to the nearest food, there might be certain areas where Pacman will be stuck such as Diagram 1.3. To prevent Pacman from moving in the same two position back and forth infinitely, I have implemented a matrix of the game positions and Pacman is not allowed to move to a location that has been visited before. This matrix will be reset every time a food is consumed by Pacman. However, this leads to the same greedy problem as Question 1(b) since Pacman focuses on only 1 food at a time rather than a true A* search algorithm.

Heuristic function:

The f-score for A* search algorithm is the sum of the g-score and the h-score. The g-score is the known cost so far which is the actual path cost from the start position to the current position of Pacman. On the other hand, the h-score is the heuristic estimate where I have again used Manhattan distance between the current Pacman position and its nearest food. I ran the same search algorithm a total of 5 times where 1 is unbiased and the other 4 is biased to one of the 4 directions each. If it is biased to one direction, the action of that direction will have its h-score deducted by 2.

Justification:

To solve the greedy problem and since there are many foods in Question 1(c) layouts, I have implemented a solution which is adding a bias to the f-score. The same search algorithm will run 5 iterations with the first iteration of no bias, and the following iterations with bias to the 4 directions respectively. This allows the Pacman to turn in different directions when reaching a junction. This bias adds to the f-score of the algorithm resulting in different paths taken in each iteration. The iteration with the most optimal solution (the shortest path) is then returned. This solution allows me to remove a little randomness to the algorithm which from testing, produce better results than without.

Pseudocode:

```
function q1c_solver (problem)

    biases <- ["", "North", "South", "East", "West"]
    solutions <- empty list
    for each bias in biases:

        frontier <- a Priority Queue
        explored <- empty list
        push the problem start state into frontier
        current <- None
        exploredMap <- matrix of size of map height x width initialized all to false
        foodLeft <- problem start state number of food left
        while frontier is not empty and current is not goal state:

            current <- Pop(frontier)

            explored <- Append(current)
            exploredMap[Pacman X position][Pacman Y position] <- true

            if current state number of food == 0:
                break loop

            if current state number of food < foodLeft:
                reset exploredMap to false
                reset frontier priority queue

            if current state number of food <= foodLeft:
                food <- nearest food to current Pacman position by ManhattanDistance


            for successor in current state successors:
                if successor Pacman position is false in exploredMap:
                    g-score <- path cost from start to current state
                    h-score <- ManhattanDistance(current Pacman position, food)
                    if successor action == bias:
                        h-score <- h-score - 2
                    frontier <- Push((successor, current state), g-score + h-score)

        solution <- backtrack explored list to find solution
        solutions <- Append(solution)

    return shortest solution from solutions
```

Part 2: Adversarial Search

Question 2(a):

Implementation:

Question 2(a) problem requires us to return an action that is best for each turn of the Pacman by using the implementation of alpha beta search. The alpha beta search algorithm is a more optimized version of minimax search algorithm by pruning the trees that do not need to be explored further by comparing the alpha, beta, the local maximum, and minimum values, making it more efficient than minimax search.

Limitations:

The main limitation in this implementation is the accuracy of the evaluation function. Alpha beta search algorithm relies heavily on the evaluation function to estimate the best move through the game state. The current evaluation function that I have created is not the most optimal evaluation function and there exists some games where Pacman will lose. However, with more testing and different methods such as machine learning, a better evaluation function which is able to produce win results would be possible. However, due to the time limitations, I have settled with the current evaluation function.

Heuristic function:

The main heuristic function of this alpha beta search is as follows:

current score + 9 / nearest food distance – 35 / nearest ghost distance + 190 / nearest scared ghost distance – number of food left * 10 – number of capsule left * 50

The property of a 1/n is that the result is greatest when n approaches 0 and that the results approach 0 when n increases. This property is suitable for us to calculate the heuristic as we would want the heuristic to be maximum when the distance is nearest and minimum when the distance is big as a far variable such as a ghost or food should not affect Pacman's action greatly. By adjusting the numerator, we can directly control how much a variable (e.g. food, ghost) would affect the heuristic, where a greater numerator infers a greater influence on the heuristic.

The initial values are chosen as the " score benefit" for each target. For example, eating a food will increase the score by 10, hence the numerator for the nearest food distance is a value just below 10 and the numerator the nearest scared ghost is 190 as eating a scared ghost will increase the score by 200. This heuristic equation is the results of repeated testing conducted throughout the assignment.

Justification:

Minimax search algorithm would be also plausible in this scenario. However, Alpha Beta search algorithm is much preferred as it is more efficient than Minimax search algorithm. This is because Alpha Beta search algorithm is essentially Minimax search algorithm, but with an additional pruning step. This additional pruning step compares the child values against the alpha and beta value. For example, if the child has a score of less than the alpha value, it means that the algorithm has a better option elsewhere, hence pruning the child node.

Pseudocode:

```
function getAction(self, gameState):

    function minSearch(gameState, depth, agentIndex, alpha, beta):
        if gameState is win or gameState is lose or depth == self.depth:
            return heuristics(gameState)
        retValue <- inf
        currBeta <- beta
        nextAgent <- (agentIndex + 1) % gameState number of agents
        if nextAgent == 0:
            for action in agentIndex legal actions:
                successor <- game state successor of agentIndex and action
                retValue <- min(retvalue, maxSearch(successor, depth + 1, nextAgent, alpha,
currBeta))
                if retValue < alpha:
                    return retValue # pruning
                if retValue < currBeta:
                    currBeta <- retValue
        else:
            for action in agentIndex legal actions:
                successor <- game state successor of agentIndex and action
                retValue <- min(retvalue, minSearch(successor, depth + 1, nextAgent, alpha,
currBeta))
                if retValue < alpha:
                    return retValue # pruning
                if retValue < currBeta:
                    currBeta <- retValue
        return retValue
    function maxSearch(gameState, depth, agentIndex, alpha, beta):
        if gameState is win or gameState is lose or depth == self.depth:
            return heuristics(gameState)
        retValue <- -inf
        currAlpha <- alpha
        for action in agentIndex legal actions:
            successor <- game state successor of agentIndex and action
            retValue <- max(retvalue, minSearch(successor, depth, agentIndex + 1,
currAlpha, beta))
        if retValue > beta:
            return retValue # pruning
        if retValue < currAlpha:
            currAlpha = retValue
        return retValue

    maxValue <- -inf
    nextAction <- "Stop"

    for action in Pacman legal actions:
        successor <- successor of Pacman and action
        actionVal <- minSearch(successor, 0, 1, -inf, inf)
        if action != "Stop" and actionVal > maxVal:
            nextAction <- action
            maxVal <- actionVal
    return nextAction

function heuristics (gameState):
    pacman <- current pacman position
    if number of food left > 20:
        nearestFood <- distance of nearest food to pacman from a sample of 15 food
    else:
        nearestFood <- distance of nearest food to pacman

    nearestGhost <- distance of nearest ghost to pacman
    nearestScaredGhost <- distance of nearest scared ghost to pacman
     nearestCapsule <- distance of nearest capsule to pacman

    heuristic <- 9/nearestFood - 35/nearestGhost + 190/nearestScaredGhost +
30/nearestCapsule- number of food left * 10 - number of capsule left * 50

   return current score + heuristic
```

Question 2(b):

Implementation:

The problem of this question is essentially the same as the problem of Question 2(a), but with more ghost. There are 4 ghosts in this question. The number of node levels for the search tree will be the number of agents multiplied by the depth of the search. Since the number of agents has increased to 5, I have decreased the depth of the search from 3 in Question 2(a) to 2. This prevents the algorithm calculation from branching out into too many terminal nodes which increases the complexity exponentially as the time complexity of Alpha Beta search is b^d where b is the branching factor while d is the depth. Hence, the implementation of the solution is also an Alpha Beta search which is the same as Question 2(a) but instead with a depth of 2.

Limitations:

The limitation of this algorithm is that by reducing the depth will decrease the optimality of the action by Pacman. However, this is a sacrifice that must be made as a depth of 3 has too much computation complexity which results in the algorithm exceeding the time constraint.

Justification:

Alternative algorithms such as a Deep Search algorithm that does not consider the presence of ghosts would be preferred as compared to alpha beta search algorithm due to the high number of ghosts in this question. This allows the algorithm to expand Pacman search tree further without consideration of the ghosts. However, due to my own lack of understanding of the said algorithm, I have decided to still implement alpha beta search algorithm for this question.

Heuristic function and Pseudocode:

The heuristic function and pseudocode of this question is the same as Question 2(a). *Refer to page 12 to 14.*