Project Report: Search

Members:

1. Nhữ Anh Thư - 17021342 – 10
2. Nguyễn Sĩ Tùng - 17021354 - 9
3. Nguyễn Vương Tiến – 17021343 - 9

**Question 1:** **Depth First Search**

* Idea:
* In this problem we have to implement Depth First Search algorithm in order to find pacman a route leading him to the goal point.
* The Depth First Search (DFS) algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.



* So the idea in here is that we consider each pacman move’s coordinate is a node, bring it up this strategy:
* Pick a starting node (starting coordinate of pacman) along with his current move taken (currently none because he haven taken any move yet) and push into a stack.
* So in each pacman move’s coordinate have 0 to 4 possible move for the next step. Pop a node and move from stack to select the next node to visit and push all its along with the move been updated into a stack. Remember to mark all the visited node.
* Repeat this process until the stack is empty or pacman has founded his path to the goal point.
* Pseudo code:

Def depthFirstSearch():

stack = Stack()

     visited = []

     stack.push([ starting positon , [] ])

     while not stack.isEmpty():

         current,instruct = stack.pop()

         if current is goalpoint:

             return instruct

         if current not in visited:

             visited.append(current)

         for coordinate,move in all possible step:

             if coordinate not in visited:

                 way = instruct + move

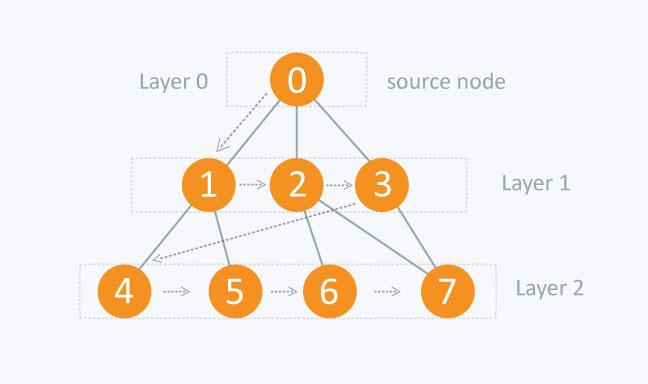
                 stack.push([coordinate,way])

* Result: **3/3 tests**

(See the full code at search.py in depthFirstSearch function)

**Question 2:** **Breadth First Search**

* Idea:
* Along with DFS, Breadth First Search agorithm is also a popular traverse graphs to help pacman find his path.
* Breadth First Search (BFS) is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.



* As the name speak for itself, we came up with this strategy (nearly the same as Depth First Search but we use queue instead of stack):
* Pick a starting node (starting coordinate of pacman) along with his current move taken (currently none because he haven taken any move yet) and push into a queue.
* So in each pacman move’s coordinate have 0 to 4 possible move for the next step. Pop out a node and move from queue to select the next node to visit and push all its along with the move been updated into queue. Remember to mark all the visited node.
* Repeat this process until the queue is empty or pacman has founded his path to the goal point.
* Pseudo code:

def breadthFirstSearch():

    queue = Queue()

    visited = []

    queue.push([ starting positon , [] ])

    while not queue.isEmpty():

        current,instruct = queue.pop()

        if current is goalpoint:

            return instruct

        if current not in visited:

            visited.append(current)

        for coordinate,move in all possible step:

            if coordinate not in visited:

way = instruct + move

                queue.push([coordinate,way])

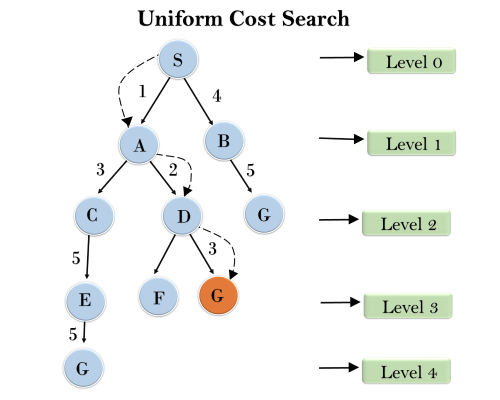
* Result: **3/3 tests**

(See the full code at search.py in breadthFirstSearch function)

**Question 3: Uniform-cost graph search**

* Idea:
* A bit similar to BFS and DFS algorithm but cost between nodes are considered, we got Uniform-cost graph search.
* Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand.

A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost.



* Our strategy still keep it simple to search the node of least total cost first:
* Push in the priority queue 2 parameter:

1. An object including starting node (starting coordinate of pacman) and move (currently none because he haven taken any move yet)
2. Total cost as priority number.

* Pop out a node and move from priority queue to select the next node to visit. Recalculate the cost to that node and push all of it into the priority queue. Because the priority queue will put the lowest cost to the top of the queue so this will pop out the possible shortest path first.
* Repeat this process until the queue is empty or pacman has founded his path to the goal point.
* Pseudo code:
* def uniformCostSearch():
* priorityQueue = PriorityQueue()
* visited = []
* cost = 0
* priorityQueue.push([starting positon,[]],0)
* while not priorityQueue.isEmpty():
* current,instruct = priorityQueue.pop()
* if problem.isGoalState(current):
* return instruct
* if current not in visited:
* visited.append(current)
* for coordinate , move in all possible step:
* if coordinate not in visited:
* way = instruct + move
* cost = calculateCost(way)
* priorityQueue.update([coordinate,way],cost)
* Result: **3/3 tests**

(See the full code at search.py in uniformCostSearch function)

**Question 4:** **A\* search**

* Idea:
* More advanced than Uniform-cost search,the A\* search the node that has the lowest combined cost and heuristic first.
* A\* algorithm works based on heuristic methods and this helps achieve optimality. A\* is a different form of the best-first algorithm. Optimality empowers an algorithm to find the best possible solution to a problem. Such algorithms also offer completeness, if there is any solution possible to an existing problem, the algorithm will definitely find it.
* To solve this problem with A\* search, we keep using priority queue like the previous question. But the priority number of the queue is consist of cost and heuristic:
* Push in the priority queue 2 parameter:

1. An object including starting node (starting coordinate of pacman) and move (currently none because he haven taken any move yet)
2. Sum of cost and heuristic as priority number.

* Pop out a node and move from priority queue to select the next node to visit. Recalculate the cost, heuristic to that node and push all of it into the priority queue. Because the priority queue will put the lowest cost to the top of the queue so this will pop out the possible shortest path first.
* Repeat this process until the queue is empty or pacman has founded his path to the goal point.
* Pseudo code:

def aStarSearch():

    priorityQueue = util.PriorityQueue()

    visited = []

    cost = 0

    priorityQueue.push([starting positon,[]],cost + heuristic(starting positon))

    while not priorityQueue.isEmpty():

        current,instruct = priorityQueue.pop()

        if problem.isGoalState(current):

            return instruct

        if current not in visited:

            visited.append(current)

            for coordinate , move in all possible step:

                if coordinate not in visited:

                    way = instruct + move

                    cost = calculateCost(way)

                    heru = heuristic(coordinate)

                    priority = cost + heru

                    priorityQueue.push([coordinate,way],priority)

* Result: **3/3 tests**

(See the full code at search.py in aStarSearch function)

**Question 5:** **Finding All the Corners**

* Idea:
* This question is a lot tricky than 4 questions above. Instead of 1 goal point, there are 4 goal point at 4 corners of the maze.This make us have to modify the logic of the game.
* So first the idea was change the goal point logic of the game, we create an array called visitedCorners had 4 False boolean variable.

Pacman only stop when all varible in that array are True. But function isGoalState and getSuccessors function always reset the array when we call it so we can’t put the array visitedCorners their. Instead we init the visitedCorners array at \_\_init\_\_ function and pass it through state. So state now will return pacman coordinate and visitedCorners, by doing that the visitedCorners will be constantly updated and our final job was just modifying isGoalState and getSuccessors function to make the program run.

* Pseudo code:

function isGoalState(state) {

    visitedCorner := state[1]

    for each corner in list visitedCorner {

         if corner if not visited

              return false

    }

    return true

}

function isGoalState(state) {

    visitedCorner := state[1]

    for each corner in list visitedCorner {

         if corner is not visited

              return false

    }

    return true

}

function getSuccessors(state) {

  visitedCorner := state[1]

  if nextPosition not hit walls {

    if nextPosition is corner then

      update list visitedCorner of that corner to true

  }

  append the list visitedCorner and the nextPosition to the successors

  with the action and cost and return it

  return successors

}

* Result: **3/3 tests**

(See the full code at searchAgents.py in CornersProblem class)

**Question 6:** **Corners Problem: Heuristic**

* Idea:
* In this question, the idea here is the heuristic function return the sum of manhattan distance from the position to the cloesest corner according to the pacman position.
* There is mutiple dots needed pacman to eat in order to get his goal point. The calculation of Heuristic function up to one point must consist of the manhatta distance from current coordinate to one dot and from that dot to other existing dots on the map.
* The Solution for this problem:
* Create an NearestCorner function return the nearest corner index. This function will calculate the nearest corner by calculate the manhattan distance and pick the smallest distance.
* In cornersHeuristic function if from the current position of pacman to the visited dot, it will return to 0. If have not visited,

Heuristic will equal to manhatta distance from current coordinate to one dot plus manhatta distance from that dot to other existing dots on the map.

* The NearestCorner function would help us calculate manhatta distance from one point to another.
* Pseudo code:

heuristic\_value = 0

    for all corner {

      for each corner in corner list{

        Calculate the manhattan distances

        from currentPosition to each corner

      }

      Get the closestCorner by compare the manhattan distances

      if (the closestCorner not visited)

          add the distances to the closestCorner to heuristic\_value

          update the position to closestCorner

          visit closestCorner

      else

          break

    }

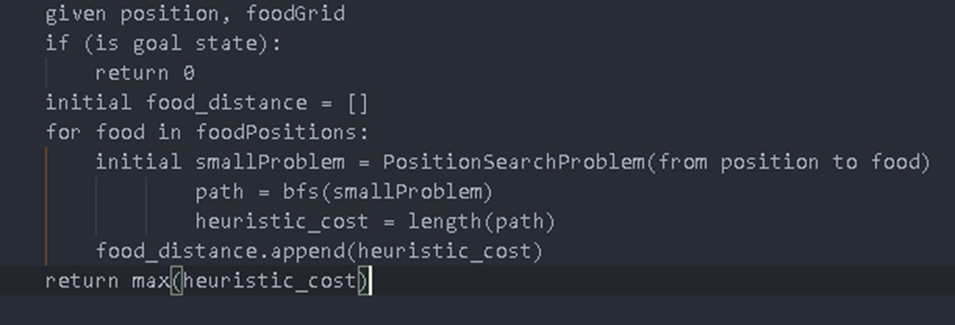
    return heuristic\_value

* Result: **3/3 tests**

(See the full code at searchAgents.py incornersHeuristic function and NearestCorner function)

**Question 7:** **Eating All The Dots**

* Idea:
* In this problem, the idea here is to find and calculate all the paths from the start position to all the available foods on the maze then return the heuristic which is equal to the longest paths to all the foods:
  + Firstly we find and calculate the paths to all the foods by using the PositionSearchProblem to find and call the bfs to calculate the paths which is the costs from start position to the foods and push them into a list called heuristic\_list
  + After that, we return the maximum of the heuristic\_list since it is the last visited node also the goal state and to avoid overestimating the cost to the goal state
* Pseudo code:



\*Note: If you run this code on window, this would take about 20-30s to Pacman to run. Ubuntu and MacOs run just fine.

* Result: **5/4 tests**

(“Search nodes expanded: 4137” this was incredible job made by Nhữ Anh Thư and that why I think he should earn extra credit)

(See the full code at searchAgents.py in foodHeuristic function)

**Question 8:** **Suboptimal Search**

* Idea:
* In this problem, there’re two parts, one is in the ClosestDotSearchAgent and the other is in the AnyFoodSearchProblem class. So we used the following solution for each of the parts
  + Firstly, with the ClosestDotSearchAgent, the main aim is to return a path to the closest dot which can be done easily with most of the search algorithms so we chose bfs for this problem
  + Secondly, the missing function of the AnyFoodSearchProblem is the isGoalState function. Because we just need to find the closest food so we just return a bool value of the question if the next position is a food.
* Pseudo code:
* For the ClosestDotSearchAgent in findPathToClosestDot function



* For the AnyFoodSearchProblem in the isGoalState function



* Result: **3/3 tests**

Provisional grades

==================

Question q1: 3/3

Question q2: 3/3

Question q3: 3/3

Question q4: 3/3

Question q5: 3/3

Question q6: 3/3

Question q7: 5/4

Question q8: 3/3

------------------

* **Total: 26/25**