**Project Report**

**Search**

Members: SID: Studen Code: Grade:

1. Vũ Quốc Khánh - 17021275 - Question 6 , Question 7 10

2. Lê Đình Hoàng - 17021254 - Question 3 , Question 5 10

3. Nguyễn Việt Hoàng - 17021257 - Question 1 , Question 2 8

4. Vũ Trọng Dương - 17021229 - Question 4 , Question 8 9

(Grade 0-10)

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

* Idea: We have a tree in which each node is a position on the map, with the terminating node (goal state) is the dot Mr. Pacman wants to eat. We need to find a route from the starting position to the dot by using Depth First Search algorithm. To do this, we decided to use the predefined Stack data structure since it is based on LIFO (last in first out) mechanism, allowing us to traverse each node's children before continuing with the neighbor nodes and their children.
* pseudo code:

Initiate Stack.

Initiate Visited set.

Stack.push(Starting state).

While Stack != Empty {

State = Stack.pop.

If State is Goal {

Return path to State.

}

If State not in Visited {

Add State to Visited.

For every Successors of State {

Successor.Action = State.Action + Successor.Action.

Stack.push(Successor).

}

}

}

* Result: 3/3 tests.

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

* Idea: We have a tree in which each node is a position on the map, with the terminating node (goal state) is the dot Mr. Pacman wants to eat. We need to find a route from the starting position to the dot by using Breadth First Search algorithm. To do this, we decided to use the predefined Queue data structure since it is based on FIFO (first in first out) mechanism, allowing us to traverse each node's neighbors before continuing with it's own child nodes and it neighbor's child nodes.
* pseudo code:

Initiate Queue.

Initiate Visited set.

Queue.push(Starting state).

While Queue != Empty {

State = Queue.pop.

If State is Goal {

Return path to State.

}

If State not in Visited {

Add State to Visited.

For every Successors of State {

Successor.Action = State.Action + Successor.Action.

Queue.push(Successor).

}

}

}

* Result: 3/3 tests.

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

* Idea: We have a tree in which each node is a position on the map, with the terminating node (goal state) is the dot Mr. Pacman wants to eat. We need to find a route with the lowest cost from the starting position to the dot by using Uniform-cost Search algorithm, which always choose the path with lowest cost first for each traverse from any node to another. To do this, we decided to use the predefined PriorityQueue data structure since it is based on FIFO (first in first out) with cost for each node and a mechanism to always pop out the lowest cost node (as implemented by default), allowing us to traverse from node to node with lowest cost.
* pseudo code:

Initiate PriorityQueue.

Initiate Visited set.

PriorityQueue.push(Starting state).

While PriorityQueue != Empty {

State = PriorityQueue.pop.

If State is Goal {

Return path to State.

}

If State not in Visited {

Add State to Visited.

For every Successors of State {

Successor.Action = State.Action + Successor.Action.

Successor.Cost = State.Cost + Successor.Cost.

PriorityQueue.push(Successor).

}

}

}

* Result: 3/3 tests.

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

* Idea: We have a tree in which each node is a position on the map, with the terminating node (goal state) is the dot Mr. Pacman wants to eat. We need to find a route with the lowest cost from the starting position to the dot by using A\* (A-star) Search algorithm, which uses heuristic function h(n) (estimated cost to get from a node to the goal), and cost to reach the node n from the start state g(n) to estimate cost of the cheapest solution f(n) (Or in short: f(n) = g(n) + h(n)). We then traverse as we would with Uniform-cost Search (UCS) but this time we choose the child node with lowest f(n) instead. To do this, we decided to use the predefined PriorityQueue data structure as with UCS, but with f(n) function instead of cost for each node. To get g(n) we already have "getCostOfActions" function function and for h(n) we also have "heuristic" function, all predefined for us in this exercise.
* pseudo code:

Initiate PriorityQueue.

Initiate Visited set.

PriorityQueue.push(Starting state).

While PriorityQueue != Empty {

State = PriorityQueue.pop.

If State is Goal {

Return path to State.

}

If State not in Visited {

Add State to Visited.

For every Successors of State {

Successor.Action = State.Action + Successor.Action.

Successor.Function = Successor.Action.Cost + Successor.Heuristic.

PriorityQueue.push(Successor).

}

}

}

* Result: 3/3 tests.

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

* Idea: We have a tree in which each node is a position on the map, with the terminating node (goal state) is the dot Mr. Pacman wants to eat. We need to find the shortest path through the CornerMaze (a predefined maze with 4 dots on each corner) that goes to all 4 corners. In this Quiz we first have to define the problem's start state, goal state and successors.
* pseudo code:

- Start state:

GetStartState {

Return Starting position, 4 corner's locations.

}

- Goal state:

IsGoalState {

Return true if the state is in any corner

}

- Successors:

GetSuccessors {

Initiate Successor list.

For any action in either one of 4 directions {

Check if next coordinate is wall.

If next state's coordinate is not wall {

If next coordinate is corner {

Remove corner from corner list.

Move to next corner.

}

Append next state to Successor list.

}

}

}

* Result: 3/3 tests.

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

* Idea: Using Manhattan distance from the given function to calculate the smallest distance to the closest unvisited corners in the map.
* pseudo code:
* cornersHeuristic {

<Predefined code>

Initialize current\_position and corner\_left = state

If there is no corner left (which is visited) in map return 0

Initialize dist\_to\_closest = smallest values from current position to nearest corner using Manhattan distance

Initialize other-corner = corner left in map

Return dist\_to\_closest + other\_corner

* Result: 3/3 tests.

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

* Idea: Same as 6 - Using Manhattan distance from the given function to calculate the smallest distance to the closest food in the map.
* pseudo code:
* foodHeuristic {

<Predefined code>

Initialize dist\_to closest = 0

Initialize foodList = foodGrid.asList()

If there is no food left in map return 0

While len(foodList !=0)

Sort the foodList to get the closest food to current position using manhattanDistance

Initialize nearest\_food = foodList[0]

Update dist\_to\_closest to get the distance to closest food using manhattanDistance

Update the current position to the nearest food

Remove the nearest food

Return dist\_to\_closest

* Result: 5/4 tests.

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

* Idea: We have a tree in which each node is a position on the map, with the terminating node (goal state) is the dot Mr. Pacman wants to eat. We need to find find a reasonably good path to the goal from the start position, while being reasonably quick to come up with the result. We have to write an agent that always greedily eats the closest dot. For this Quiz, we chose Uniform-cost Search (UCS) algorithm as the basis for our agent, since this time we use greedy strategy and UCS guarantees a good result while being reasonably simple calculation wise (unlike A\* which is too complicated and will use a lot more resources for bigger maps). ClosestDotSearchAgent is already preimplemented for us and we need to direct the function to our search functions (done in previous Quizzes), and define the goal state for each food dot (of which the dot will terminate).
* pseudo code:

- ClosestDotSearchAgent {

<Predefined code>

Return Path from current state to a dot by using Uniform-cost Search.

}

- IsGoalState {

If Coordinate has food {

Return True.

}

Else {

Return False.

}

}

* Result: 3/3 tests.