**Project Report 1**

**Project Name: The Searchin’ Pacman**

**Team Members: 1. Damayanti Sengupta 112078047**

**2. Sania Parveen 112027609**

**Purpose:** The purpose of this report is to explain and summarize the various approaches/strategies implemented for all the tasks part of the Project 01

**Search Strategies:**

1. **Depth-First Search:**

**Implementation Details:**

For DFS we started exploration from the starting node of the packman and explored all its successor nodes as long as possible before backtracking. While visiting every new node we verified if the node we are visiting is our goal state. If yes, we returned the path from the starting position to our goal state. Otherwise we kept exploring the rest of the nodes not yet visited until we reached the goal state. To implement DFS we used the standard approach of storing the nodes on the current search path in a stack. We also maintained a list of nodes already visited in a fringe list to avoid the execution getting stuck in an infinite loop.

Metrics: (Tiny Maze)

|  |  |
| --- | --- |
| Number of nodes expanded | 15 |
| Memory usage | 10.3MB |
| Running Time | 0.0 sec |

Metrics: (Medium Maze)

|  |  |
| --- | --- |
| Number of nodes expanded | 269 |
| Memory usage | 12.4MB |
| Running Time | 0.0 sec |

Metrics: (Big Maze)

|  |  |
| --- | --- |
| Number of nodes expanded | 466 |
| Memory usage | 13.0MB |
| Running Time | 0.0 sec |

**Analysis:**

DFS is a search algorithm which is simple to implement. It is sub-optimal as it may not return the best optimal solution.

Its space complexity is O(|V|) in the worst case where V is the length of the longest path and can take time upto O(|V| + |E|) where V is the number of vertices and E is the number of edges.

1. **Breath-First Search:**

**Implementation Details:**

For BFS, we started exploration from the starting node of the Pacman and explored all the neighbor nodes at the present depth prior to moving on to the nodes on the next level. While visiting every new node we verified if the node we are visiting is our goal state. If yes, we returned the path from the starting position to our goal state. Otherwise we kept exploring the rest of the nodes not yet visited until we reached the goal state.   
To implement BFS we used the standard approach of storing the nodes on the current search path in a queue. To track path between a visited node and the starting point, while pushing the node’s state in the queue we pushed its complete path from the starting point as part of the state

Metrics: (Tiny Maze)

|  |  |
| --- | --- |
| Number of nodes expanded | 15 |
| Memory usage | 10.2 MB |
| Running Time | 0.0 sec |

Metrics: (Medium Maze)

|  |  |
| --- | --- |
| Number of nodes expanded | 269 |
| Memory usage | 11.6 MB |
| Running Time | 0.0 sec |

Metrics: (Big Maze)

|  |  |
| --- | --- |
| Number of nodes expanded | 620 |
| Memory usage | 13.1 MB |
| Running Time | 0.0 sec |

**Analysis:**

BFS is a complete search algorithm i.e. its optimal as it returns the shortest route (if it exists) between two nodes.

BFS may use more memory than DFS depending on the branching factor. It takes up-to ***O(bd)*** space where ***b*** is the branching factor and ***d*** is the depth

Time complexity of BFS is same as DFS

1. **Uniform Cost Search:**

**Implementation Details:**

Since Uniform-Cost Search is an extension of BFS, we used a similar algorithm as BFS to search the state space. Instead of using a queue to store the fringe list(like we do in BST), we now used a min priority queue. The states stored in the priority queue are ordered by the cost of reaching that node from the starting point, which is given by the function ***g(n)***. So instead of expanding the shallowest node, UCS expands the node ***n*** with the lowest path cost ***g(n)***.

While exploring the path the goal test is applied to a node when it is selected for expansion rather than when it is first generated. The reason why this is done is because the first goal that is generated might be on a suboptimal path.

Metrics: (For Medium Maze)

|  |  |
| --- | --- |
| Number of nodes expanded | 267 |
| Memory usage | 11.7MB |
| Running Time | 0.0 sec |

Metrics: (Medium Dotted Maze with StayEastSearchAgent)

|  |  |
| --- | --- |
| Number of nodes expanded | 186 |
| Memory usage | 11.5MB |
| Running Time | 0.0 sec |

Metrics: (Medium Scary Maze with StayWestSearchAgent)

|  |  |
| --- | --- |
| Number of nodes expanded | 108 |
| Memory usage | 15.1MB |
| Running Time | 0.0 sec |

**Analysis:**

Uniform-Cost Search is an uninformed search. It is an extension of BST but it can work with paths with non-uniform weights. Unlike UCS, BFS can generate optimal path only if all the steps have same cost.

Uniform-Cost Search always expand nodes in order of their optimal cost. Hence the first goal node selected for expansion is the optimal solution. The worst-case space and time complexity for UCS is ***O(b(1+floor(C/ε))***, where ***b*** is the branching factor, ***C*** is the total optimal cost to destination and ***ε*** least cost of every action.

1. **A\* Search:**

**Implementation Details:**

For A\* search at each step we picked the next node for exploration according to a value: ‘**f**’ which is a parameter equal to the sum of two other parameters – ‘**g**’ and ‘**h**’. At each step we picked the node/cell having the lowest ‘**f**’ and processed that node/cell. Here ‘**h’** is the heuristic function which estimates the distance between two nodes based on the Manhattan Distance.

To store the fringe list we used a min priority queue same as UCS. The nodes in the fringe list were maintained based on the priority **f**, which is **g + h**. While picking the heuristic function, we ensured it was admissible and consistent.

Metrics: (Big Maze)

|  |  |
| --- | --- |
| Number of nodes expanded | 585 |
| Memory usage | 13.5MB |
| Running Time | 0.0 sec |

**Analysis:**

A\* is a widely used algorithm for searching a state space because of its accuracy and performance. Its an informed search algorithm as it uses heuristics to guide its search. In every iteration, A\* aims to find a node which has minimum cost which is calculated by a function ***f(n)***

***f(n) = g(n) + h(n)***, where ***g(n)*** is the cost from the starting node to the current node and heuristic function ***h(n)*** is the estimated cost from the current node to the goal node.

A\* is complete and optimal. Compared to UCS, A\* finds the optimal solution faster. For a big maze where UCS expanded 620 nodes, A\* expanded 585 nodes to find the path.

The complexity of the A\* search depends largely on the heuristic function we use. In the worst case its time complexity is ***O(bd)*** as the number of nodes expanded is dependent on the branching factor ***‘b’*** and the depth of the solution ***‘d’***

A\* search assumes that the goal state exists and is reachable from the start space. If not, the search algorithm will never terminate.

1. **Corners Problem:**

**Implementation Details:**

The details of the abstract methods which we implemented is as follows:

* **getStartState:** This method returns the starting position of the Pacman.
* **isGoalState:** This method checks if we have visited all the 4 corners or not. If yes, returns True otherwise False. The count of the number of corners visited is stored in the state which is passed as an argument to this method.
* **getSuccessors:** The method returns all possible moves the Pacman can make from the current state. While generating the successors we also verify if the current state is a goal state which we have not already visited. If yes, we append the current node in the list of visited goal nodes and store this list of goals in the state of the successor node so that the goal list can be retrieved in the next iteration for performing validations in **isGoalState** method.

Metrics: (Tiny Corners)

|  |  |
| --- | --- |
| Number of nodes expanded | 435 |
| Memory usage | 10.5MB |
| Running Time | 0.0 sec |

Metrics: (Medium Corners)

|  |  |
| --- | --- |
| Number of nodes expanded | 2448 |
| Memory usage | 11.6MB |
| Running Time | 0.0 sec |

**Analysis:**

Since for this task we used BFS as a search algorithm, it expands too many search nodes as BFS is an uninformed search algorithm and does not use heuristics. One key point to note is that in this search approach a goal node may be visited multiple times while searching the state space to find an optimal path visiting all the goal nodes. But no goal nodes are visited twice within a state space tree.

1. **Corners Heuristic:**

**Implementation Details:**

The heuristic function we decided for this task is intuitive. We first look for the closest corner from the starting position of the pac- man. This we do by iterating over the list of the corners and calculating the manhattan distance between the starting node and the goal node. Once we have our closest goal node from the starting position (let this be **G1**) we now start iterating over the remaining goal states which are yet not visited and find the closest corner from **G1**. Let this node be **G2.** For the third iteration **G2** becomes our starting point and we then look for the corner closest to **G2.** This goes on until we include all the goal nodes in our path starting from Pacman’s initial state. The heuristic function returns the total cost of the path from starting node to G1, then G2 till the last goal, by aggregating the manhattan distance for each path.

Metrics: (Tiny Corners)

|  |  |
| --- | --- |
| Number of nodes expanded | 217 |
| Memory usage | 10.6MB |
| Running Time | 0.0 sec |

Metrics: (Medium Corners)

|  |  |
| --- | --- |
| Number of nodes expanded | 901 |
| Memory usage | 11.4MB |
| Running Time | 0.1 sec |

**Analysis:**

For the search problem if the heuristic used is admissible, A\* algorithm will never return suboptimal path. Since the heuristic function we have used is admissible (as our estimated path cost is always less than equal to the actual cost), the path returned by the algorithm is optimal. For searching the path visiting all the corners, the A\* with heuristic did much better than BFS search as it expanded lesser nodes than BFS.

1. **Food Heuristic:**

**Implementation Details:**

First, we find a dot that is surrounded by food on all sides in our Food List. There may be more than one such dot. In that case, we can break the tie randomly.   
Second, we find how far the *furthest food* is from our **target** food selected above. This is to find out how much distance we add to our cost if we travel to **target** first and eventually need to travel to eat the *furthest food from* ***target***. Since we are relaxing our problem, we use Manhattan distance to calculate the most distant food.  
Third, we see which point we are currently close to: our **target**, or the *furthest food* *from* ***target***. If the *furthest food* *from* ***target*** is closer to us, it makes sense to eat it first, and then go to the **target**, as we won’t have to travel back part of the same path and add to our cost in that case.   
Hence, we take the minimum of (distance to **target** from current position, distance to *furthest food from target* from current position). We use BFS to get actual maze distance and add the maze distance between our **target** and *furthest food from* ***target***, as once we choose our next position, we will still have to travel to the other as part of our optimal path. This is the heuristic that we return.

Metrics: (Tiny Search)

|  |  |
| --- | --- |
| Number of nodes expanded | 842 |
| Memory usage | 12MB |
| Running Time | 1.1 sec |

Metrics: (Tricky Search)

|  |  |
| --- | --- |
| Number of nodes expanded | 1828 |
| Memory usage | 17.7MB |
| Running Time | 4.8 sec |

**Analysis:**

For this problem, taking a simple Manhattan distance as in the Corners heuristic would be inadmissible, and would return a sub-optimal cost/path.   
We relaxed the problem by assuming that we can go through the walls. An intuitive approach to solve food search would be to head towards a spot in the maze where a lot of food dots are clustered together. Clustering implies that the inter-food distances in that area are small, so if we move to that spot, we can eat the food in one go without traversing large distances (as opposed to moving in a zig-zag manner).   
This heuristic is both **consistent** and **admissible** as we are estimating the max inter-food distance using Manhattan Distance (a relaxation of the problem) and then using the real distances as part of our heuristic when deciding which point to travel to next. Since we are only counting the distance to travel to the nearest of the potential “next position” and the distance between them, and not considering the other food dots we have to travel to, we always underestimate the cost of the optimal path.

**Conclusions:**

Below are the key take-aways from this assignment:

1. Informed search does much better than un-informed search while searching a path in a state space. As informed search algorithms expand much less nodes than uninformed.
2. A\* search algorithm can get stuck in an infinite loop if the state space we are searching has no goal state.
3. The performance and the complexity of A\* search largely depends on the heuristic function used by the algorithm.
4. While implementing heuristic function it is essential to ensure the heuristic is admissible and consistent. Otherwise informed search algorithm may return sub-optimal result, even if it may appear to be optimal because of low number of search nodes expanded.