FIT 3080 Assignment 1 report

Name: Tuan Muhammad Zafri
ID:31989632

Q1

Proposed approach: A* algorithm, uses actual cost and a heuristic function to estimate the cost to goal. Actual cost, g(n) is calculated by number of actions (each action has cost=1) from pacman's starteState position to current position, whereas heuristic h(n) is calculated using the manhattan distance from pacman's current position to the goal/food.

Heuristic

Q1a) manhattan distance to the food from pacman's position
Q1b) & Q1c) minimum manhattan distance to each food

Pseudo-code of A* algorithm:

```
Function q1a_solver(problem):
    Initialize start_state
    Initialize goal_position
    Initialize start_node with (start_state, None, None, 0)
    Initialize frontier as an empty PriorityQueue
    Push start_node into frontier with priority 0
    Initialize explored as an empty set

    While frontier is not empty:
        Pop node from frontier

        If number of food is less by one: (For q1b and q1c only)
            Empty explored
            Reset frontier
            Push popped node
            continue

        If node's position is in explored:
            Continue

        Add node's position to explored
```

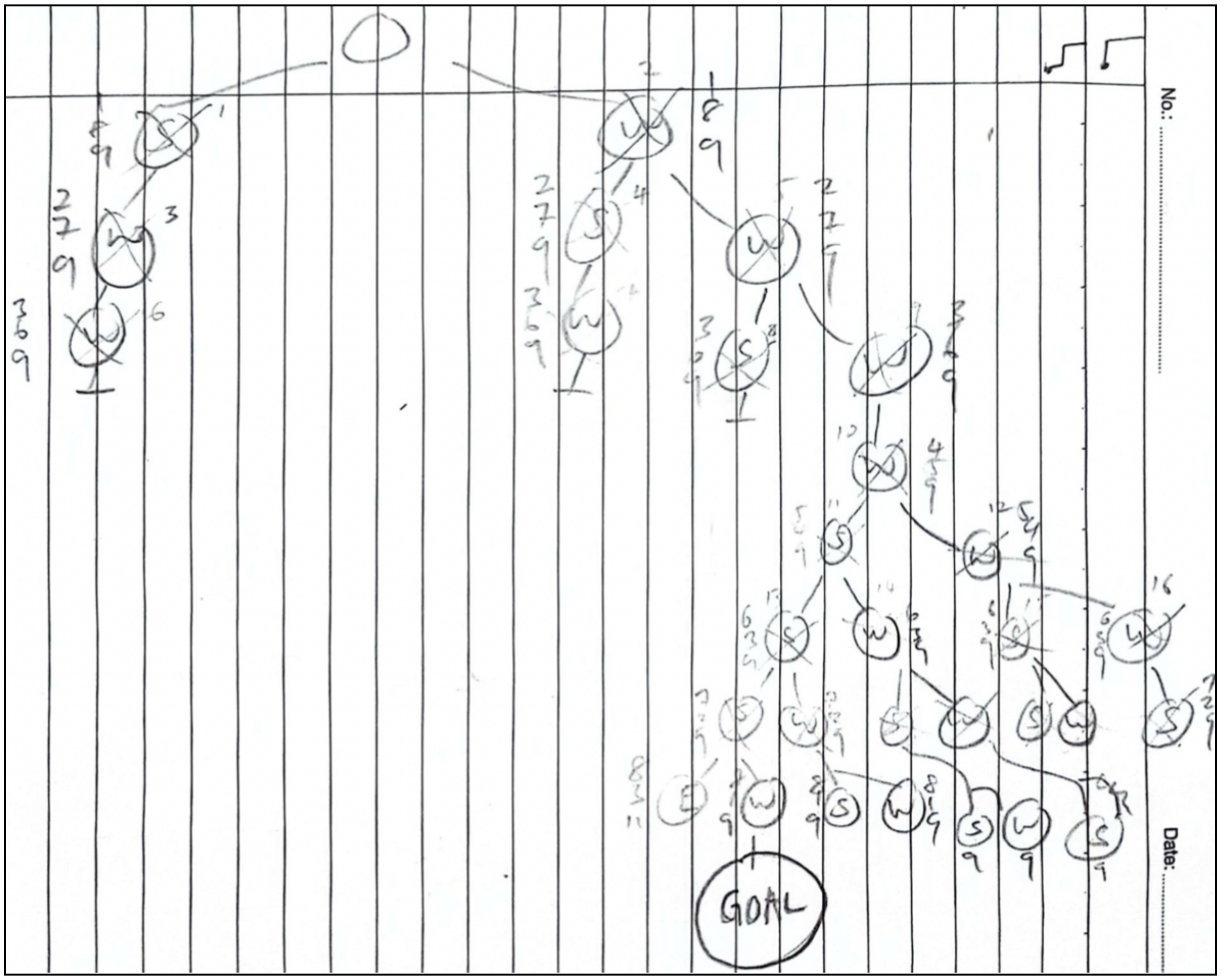
```
If node's state is a goal state:
    Return node's path

For each successor in problem.getSuccessors(node's state):
    If successor's Pacman position is not in explored:
        Initialize child_node with successor's state, node, successor's action, and len(node's
path) + 1
       For goal in goal_position(only for q1b and q1c)
        Calculate manhattan distance from child_node.position to goal_position

        If child_node's state is a goal state:
            Return child_node's path

        Else:
            Calculate fn = child_node's cost + manhattan

        Update frontier with child_node and priority fn
```


(My working when debugging the A* algorithm)

Motivation to choose the proposed ideas:

For this problem, we know the gamestate which provides the informations of pacman and food position, therefore informed search algorithms are suitable compared to uninformed search algorithms.
There are 3 ideas of informed search algorithms, A* algorithm, IDA* algorithm and greedy best first search.

Comparative analysis

Ideas	A* algo	IDA* algo	Greedy Best First Search
Complete	Yes	Yes	No
Optimal	Yes	Yes	No
Revisit nodes	No	Yes	No
Space	O(b^d)	O(d)	O(b^d)
Time	O(b^d)	O(d)	O(b^d)

Advantages and Disadvantages in practice

A* Algorithm

Advantages:
Optimality: A* guarantees an optimal solution when an admissible heuristic is used.
Efficiency: It is generally faster in finding an optimal solution compared to uninformed search algorithms like BFS or DFS.

Disadvantages:
Memory Usage: A* has a high memory footprint due to the storage of all visited nodes.
Heuristic Dependency: The algorithm's performance is highly dependent on the quality of the heuristic used.
Example: In a Pacman game where the goal is to find the shortest path to a food item, A* would excel if a suitable heuristic like Manhattan distance is used.

IDA* Algorithm

Advantages:
Memory Efficiency: IDA* uses significantly less memory than A*, making it suitable for memory-constrained environments.
Optimality: Like A*, IDA* also guarantees an optimal solution when an admissible heuristic is used.

Disadvantages:
Time Complexity: IDA* can be slower than A* because it revisits nodes across different iterations.
Heuristic Dependency: The algorithm still relies on a good heuristic for efficient functioning.

Example: In a large maze where memory is a constraint, IDA* would be more suitable than A*, even though it may take longer to find the solution.

Greedy Best-First Search

Advantages:
Speed: Greedy Best-First Search often finds a solution quickly, making it suitable for time-sensitive applications.
Simplicity: The algorithm is easier to implement compared to A* or IDA*.

Disadvantages:
Suboptimality: The algorithm is not guaranteed to find the shortest path to the goal.
Incompleteness: In some scenarios, the algorithm may get stuck in loops and fail to find a solution.

Example: In a Pacman game where speed is more important than finding the shortest path, Greedy Best-First could be an appropriate choice.

Experiment (For A* and greedy best first search algo)

Heuristic (A* algo)

a) Total manhattan distance to all food on the layout currently
    - I realized that this heuristic will take a longer path and the reason is it might overestimate the estimated cost and hence it is not admissible anymore
b) Minimum manhattan distance to all the food on the layout currently
    - This heuristic is admissible and more accurate, hence leading to the goal state faster

I implemented b) for q1a,b,c, however for q1b and q1c I need to clear the frontier and explored list after eating one food so that pacman's path will not be blocked by the states that were visited to get to the previous food.

Actual cost
A* calculates the actual cost needed from the start state to the current state and add it with the heuristic for its final cost f(n), whereas Greedy BFS only uses the heuristic.

Benchmark
After experimenting, greedy BFS does not guarantee a solution as it might be stuck in local optima (not considering walls), whereas A* will guarantee an optimal solution. On average, Greedy BFS takes faster time to compute but will produce a longer path, A* takes longer time to compute but will produce a shortal optimal path. I used the time taken and the path length as benchmarks.

Limitations

While A* is optimal, its memory usage could be a limitation for larger mazes, as it needs to store all the visited nodes/states. Future works could involve parrallelizing the search or use the more memory efficient IDA* algorithm, but IDA* might be slower.

Conclusion

For the layouts provided, A* is the most suitable informed search algorithm as it is optimal and considered efficient. Greedy BFS takes a shorter time but does not guarantees an optimal solution.

Q2

Approach: Minimax function for q2a and q2b

Pseudocode for Minimax function

```
Algorithm Minimax(gameState, depth, agentIndex, alpha, beta)
    Input: gameState, depth, agentIndex, alpha, beta
    Output: Best utility value and corresponding action

    if gameState is a win or a lose or depth is 0
        return evaluationFunction(gameState), None

    numAgents = get number of agents in gameState
    legalActions = get legal actions for agent at index agentIndex

    if agentIndex is 0 (Pacman)
        maxBackup = -Infinity
        bestAction = None
        for each action in legalActions
            successor = generate successor gameState using action
            backup, _ = Minimax(successor, depth, (agentIndex + 1) % numAgents, alpha, beta)

            if backup > maxBackup
                maxBackup = backup
                bestAction = action

            alpha = max(alpha, backup)

            if beta <= alpha
                break  // Beta cut-off
        return maxBackup, bestAction

    else (Ghosts)
        minBackup = +Infinity
        bestAction = None
        for each action in legalActions
            successor = generate successor gameState using action

            if agentIndex is the last agent
                backup, _ = Minimax(successor, depth - 1, 0, alpha, beta)
            else
                backup, _ = Minimax(successor, depth, (agentIndex + 1) % numAgents, alpha, beta)

            if backup < minBackup
                minBackup = backup
                bestAction = action

            beta = min(beta, backup)

            if beta <= alpha
                break  // Alpha cut-off
        return minBackup, bestAction
```

Motivation to choose the proposed ideas:

Minimax

Minimax is a decision rule used for minimizing the possible loss for a worst-case scenario in two-player games (such as Pac-Man). When dealing with games like Pac-Man that involve adversaries (Pacman vs. Ghosts), it's crucial to consider not only how to maximize your own benefit but also how your opponent might act to minimize it. Here are some reasons why Minimax is often used:

Advantages:
Optimality: Guarantees the best possible outcome against a rational opponent when the game tree is fully explored.
Complete Information: Assumes that all information is available to all players. This is true for many classic games like chess, tic-tac-toe, and in your case, Pac-Man.
Generality: Can be applied to any two-player game with win/lose outcomes.
Alpha-Beta Pruning: Can be optimized with techniques like alpha-beta pruning to reduce the number of nodes evaluated in the search tree.

Disadvantages:
Computational Complexity: Minimax has a time complexity of O(b ^d), where
b is the branching factor and d is the depth of the game tree. This makes it computationally expensive for games with a high branching factor and depth.

Memory Usage: The algorithm keeps the entire game tree in memory. For large trees, this could be a memory-intensive operation.

Assumption of Rationality: Minimax assumes that the opponent will play optimally, which might not always be true in real-world scenarios.

Comparative Analysis:

Versus Greedy Algorithms: While Minimax considers the entire game tree up to a specific depth, greedy algorithms only look at the immediate next state. This makes Minimax more accurate, albeit at the expense of computational time.

Versus Monte Carlo Tree Search (MCTS): MCTS employs randomness and does not make assumptions about optimal opponent behavior. This makes MCTS faster in some cases but potentially less accurate.

Relevance to the Pac-Man Problem:
In the context of Pac-Man, the ghosts serve as adversaries aiming to capture Pac-Man. A simplistic algorithm may fail to effectively consider how the ghosts could obstruct or catch Pac-Man. Minimax, on the other hand, evaluates both the best and worst-case outcomes at each decision point, making it more suited for such adversarial contexts.

Experiment (For Q2a & Q2b)

ScoreEvaluationFunction

1. Score minus minimum food distance and minimum ghost distance
Pros:
Simple and computationally inexpensive.
Takes into account both food and ghost distances, which are critical game components.
Cons:
May not consider other critical aspects like scared ghosts or capsules.

2. Detailed Evaluation Function
Pros:
Comprehensive, takes into account a variety of factors (ghost distance, food distance, scared ghosts, number of food and capsules remaining).
Adapts the score based on Pac-Man's proximity to various game elements.
Cons:
More computationally intensive due to the number of conditions and calculations.
Fixed weights on the number of remaining food and capsules may not always be ideal.

3. Global Last Action Tracker
Pros:
Tries to avoid Pac-Man getting stuck by penalizing the same position.
Includes detailed aspects like local and global penalties, and even a random tie-breaker.
Cons:
Requires a global variable to track the last action, which might make the function less pure and harder to debug.
Even more computationally intensive due to additional calculations for local penalties and random tie-breaker.

Benchmarks
1. Score minus minimum food distance and minimum ghost distance
Time Taken: Likely the fastest due to its simplicity.
Score: May not maximize the score as it doesn't consider many game aspects.
Path Length: Could be longer as it doesn't account for all game elements.

2. Detailed Evaluation Function
Time Taken: Moderate, due to the multiple conditions checked.
Score: Likely to be higher as it considers multiple aspects to make a decision.
Path Length: Expected to be optimal or near-optimal because of the comprehensive evaluation.

3. Global Last Action Tracker

Time Taken: Potentially the slowest due to the complexity and additional calculations.
Score: Could be higher but depends on how well the penalties and tie-breakers work.
Path Length: May be shorter if the function effectively avoids re-visiting locations, but could be longer if it gets stuck in less optimal paths due to the penalties.

Summary
The first function is likely the fastest but may achieve lower scores and longer paths.
The second function is a balanced option, potentially achieving higher scores and shorter paths at the cost of moderate computation time.
The third function might take the most time and its effectiveness in achieving high scores or short paths is dependent on the specific penalties and tie-breakers used.

Remaining Gaps

Optimality: None of the evaluation functions guarantee an optimal solution in terms of maximizing the score. Further fine-tuning or a different approach may be needed for that.

Scalability: The third function's computational complexity might become an issue in more complex game states or larger mazes.

Parameter Tuning: All three functions have hard-coded weights for different aspects like food, capsules, and ghosts. These could be optimized further.

Conclusion

After careful consideration, I decided to use the second function for Q2a and the third function for Q2b for the following reasons:

Balance: The second function provides a good balance between computational efficiency and a comprehensive evaluation of the game state, making it ideal for the simpler scenario in Q2a.

State Avoidance: The third function's capability to avoid revisiting states is particularly useful for Q2b, where the game state can get more complex and avoiding repeated states becomes crucial.

Local and Global Penalties: The third function offers a more nuanced evaluation by considering both local and global game state aspects, which becomes increasingly important as the game progresses in complexity in Q2b.