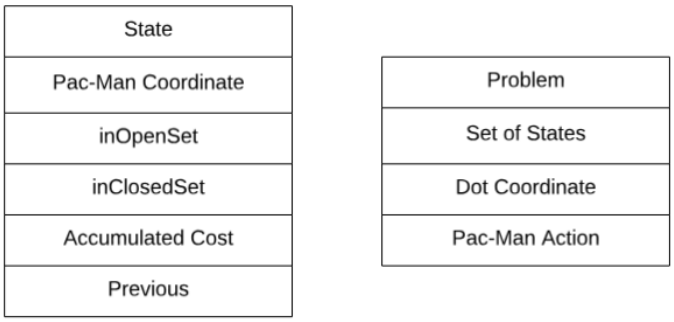
Question 1(a)

Problem Description

The problem wants us to identify the most optimal path from a starting position of Pac-Man to collect a single dot across the maze using A* algorithm and the Manhattan distance heuristic.

Problem Representation



The State data structure will keep track of the current position of Pac-Man in a grid represented as a tuple (x, y) to identify successor states in the grid, determine whether Pac-Man collected the single dot and to compute heuristic value from each state to the goal state.

The advantage of this representation is that we can determine if the state is in the open and closed set during each expansion as a constant time operation as opposed to a linear time operation. This slightly improved the algorithm's overall search time.

The advantage of keeping track of the reference to the previous state as opposed to storing the list of accumulated actions at every state is that we can avoid storing repetitive data which takes up linear space for each created state. Instead, we can perform backtracking once the goal state is reached to reconstruct the action which takes up constant space. This minor optimization prevented timeout errors that I was experiencing when evaluating my code.

The coordinate of the dot, set of states and Pac-Man's action are represented outside of the State class because these attributes are static and global for all states that are within this search problem. Hence, we do not have to store unnecessary information that does not directly relate to a specific state. For instance, the single dot is static throughout the search problem until it is collected in the goal state, in which the search problem is already complete.

Solution Approach

For this problem I utilized a single agent grid-based A* search where the single-agent is the Pac-Man. Each state represents a cell in the grid that the Pac-Man can access from its actions. To ensure no duplicated states are created, a dictionary data structure is used to keep track of the created states and reference to successor states can be accessed via the grid coordinates. The compute the heuristic, the algorithm computes the Manhattan distance from the coordinate of the current agent's state to the coordinate of the goal state. This heuristic function is suitable because the Manhattan distance between the Pac-Man's coordinate and the coordinate of the dot will always be smaller than or equal to the true cost of the actual path. Thereby, making it admissible. To ensure the most optimal nodes are expanded first, a priority queue is used to store the discovered states with the sum of accumulated cost path and the heuristic as the f

value. This ensures that the path discovered by Pac-Man is the most optimal path with the lowest cost.

Question 1(b)

Problem Description

The problem wants us to identify the most optimal path from a starting position of Pac-Man to collect the closest available dot from multiple dots across the maze while ensuring the number of nodes expanded during search is optimal.

Problem Representation

The problem representation is the same as Question 1(a). However, our problem class will now keep track of the list of dot coordinates instead of one. Again, it is not necessary to keep track of the list of food in the state because the goal state is reached once a dot is collected by the Pac-Man.

Solution Approach

Since there are multiple goal states that can be reached by Pac-Man in this problem, I modified the heuristic function of the A* algorithm to accommodate the Manhattan distance that Pac-Man needs to travel to reach all possible dot in the grid.

Below, I identified and analyzed 2 different methodologies to tackle this problem:

Methodology 1)

In this methodology, I computed the Manhattan distance for Pac-Man to reach all dot in the grid and pushed each computed heuristics of the same state to the priority-queue/closed-set. This methodology then relies on the base A*algorithm to determine the most optimal node to expand from each state to reach all possible goals.

Pseudocode:

```
Function astar_heuristic(pacman_coordinate, list_of_goal_coordinate)
{
        Initialize empty list to store manhattan distance, M.
        For each goal_coordinate in list_of_goal_coordinate
        {
                Compute manhattan distance from pacman_coordinate to goal_coordinate, D.
                Store D in M.
        }
        Return M
}
```

Methodology 2)

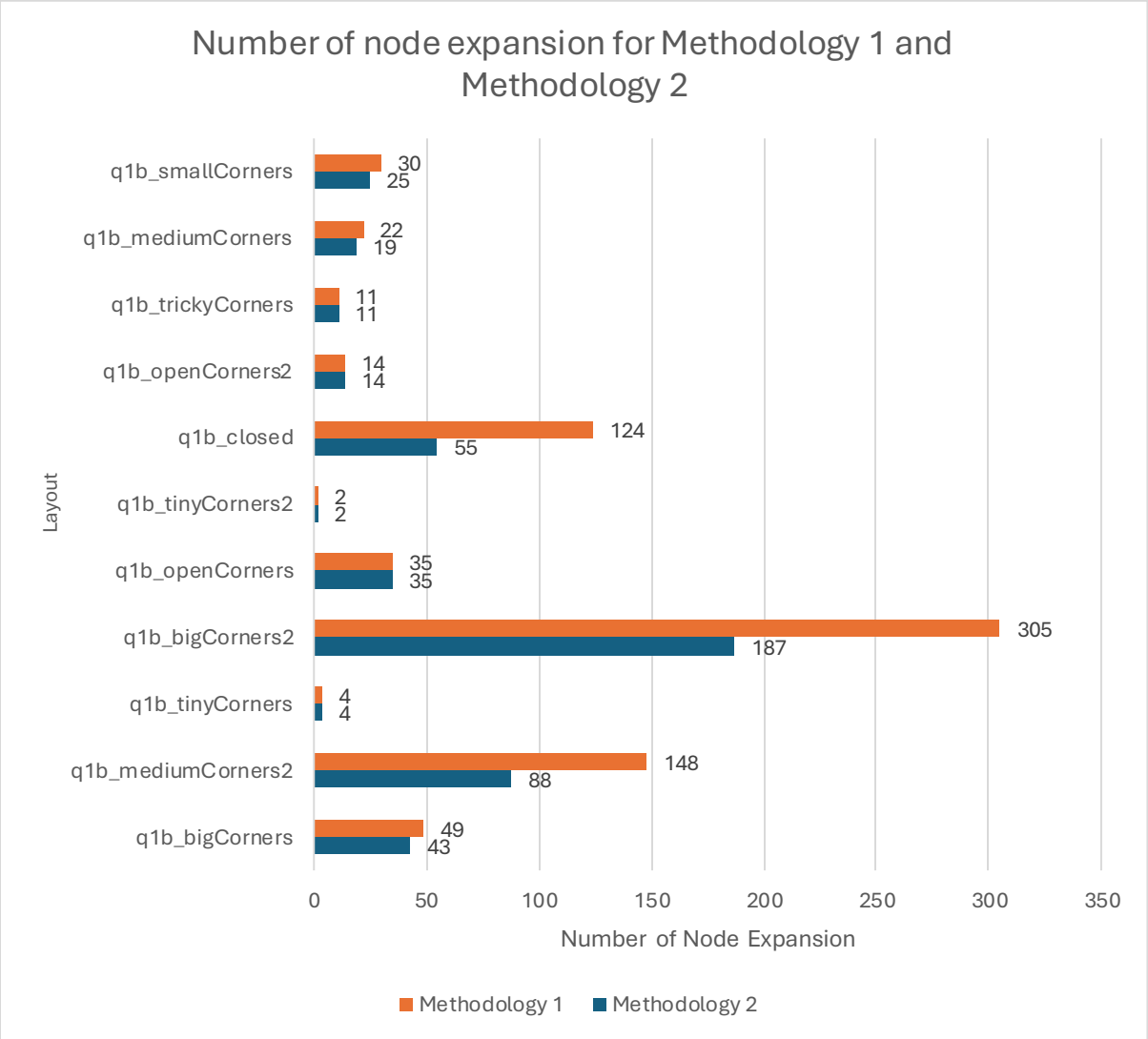
In this methodology, instead of returning the heuristic to all possible goals. We only take the shortest heuristic to reach any one of the goals from each discovered state. This is a more optimum approach than Methodology 1 because do not have to perform unnecessary node expansion to other goal states as the A* algorithm is only limited to the information of the closest goal state from the current state. This will also handle the problem where the closest

goal state to Pac-Man is not reachable because discovering adjacent states will get Pac-Man closer to other goal state that may be shorter for Pac-Man to reach in terms of true cost.

Pseudocode:

```
Function astar_heuristic(pacman_coordinate, list_of_goal_coordinate)
{
        Initialize empty list to store manhattan distance, M.
        For each goal_coordinate in list_of_goal_coordinate
        {
                Compute manhattan distance from pacman_coordinate to goal_coordinate, D.
                Store D in M.
        }
        Return the minimum manhattan distance of M.
}
```

Analysis of each Methodology:



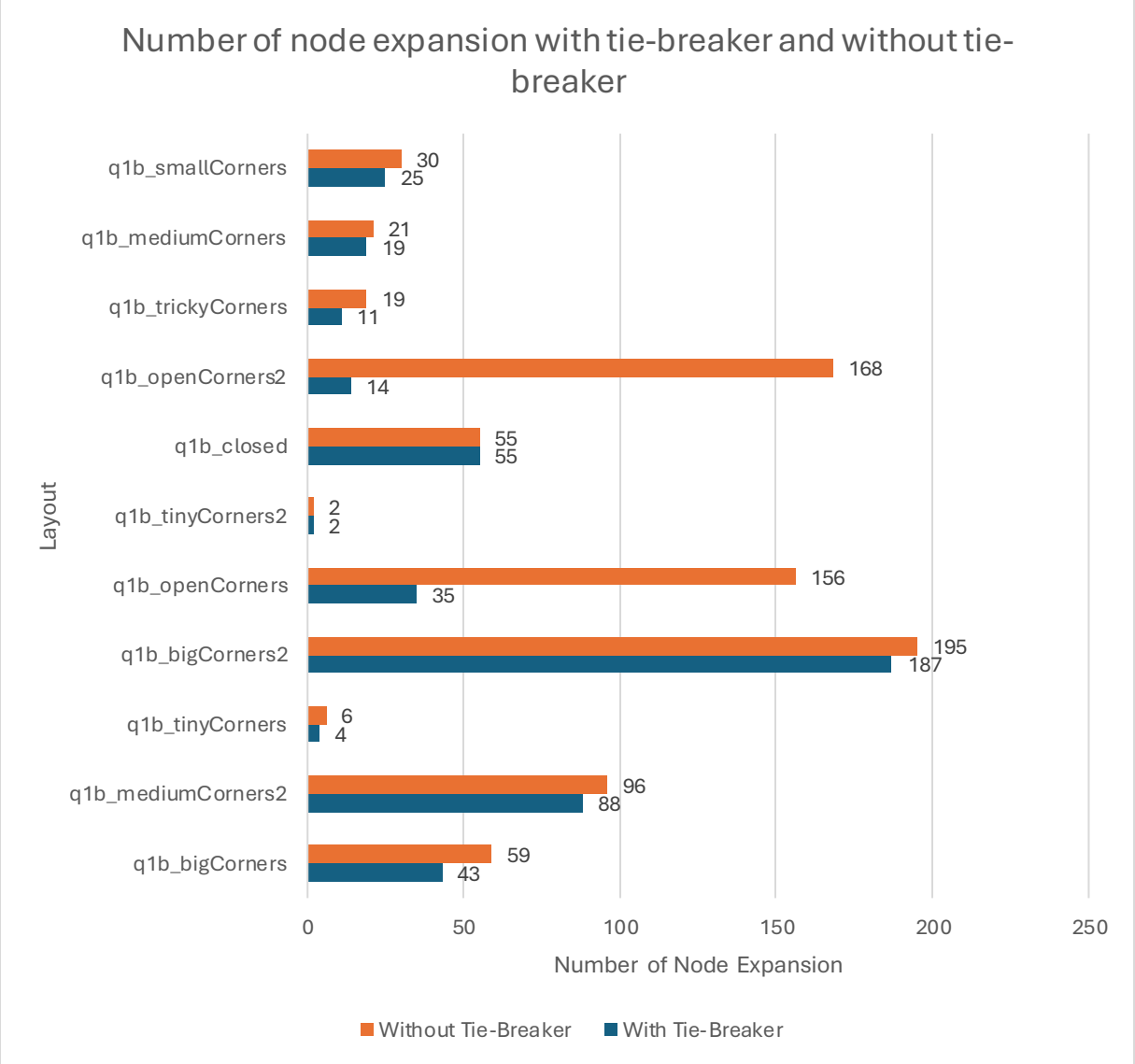
The diagram above illustrates the number of nodes expanded for each layout during the A* search for each methodology. As we can see, Methodology 1 expanded more nodes according to our hypothesis above. Therefore, Methodology 2 was utilized to eliminate unnecessary node expansions.

However, choosing an optimal methodology for our heuristic function still did not meet the optimal number of nodes that our A* search is expected to expand. I decided to observe how the A* algorithm is expanding nodes during runtime and I realized that there were many states in the closed set with the same f-value, which resulted in the A* algorithm to be expanding unnecessary number of nodes before handling the node closer to the goal state. To solve this problem, I decided to implement a tie-breaking strategy by including the h-value along with the f-value as a tuple (f-value, h-value). This approach will allow the A* algorithm to expand nodes that are closer to the goal state when there are multiple nodes with the same f-value.

Pseudocode:

```
Function push_to_open_set(pacman_coordinate, list_of_goal_coordinate, current_state)
{
        Compute heuristic value with astar_heuristic(pacman_coordinate, list_of_goal_coordinate), h_value.
        Compute f_value as -> accumulated cost to current_state + h_value.
        Store tuple T as -> (f_value, h_value).
        Push T into priority_queue/open_set.
}
```

Analysis of Tie-Breaking Strategy:



The diagram above illustrates the number of nodes expanded with and without the Tie-Breaker strategy. As we can see, the tie-breaking strategy decreases the number of node expansions according to our hypothesis above. Therefore, a tie-breaking strategy was utilized to solve this problem while ensuring node expansions are optimal.

Question 1(c)

Problem Description

The problem wants us to maximize our score by having Pac-Man collect as many dot as possible within a grid of multiple dots while ensuring the cost to move Pac-Man towards the dot does not outweigh the score benefit.

Problem Representation

The problem representation is the same as Question 1(a). However, each individual state will now keep track of the remaining dot that Pac-Man has not captured yet. This is because the goal state is reached once all dots have been captured by Pac-Man or the remaining dots does not provide a score that outweighs the cost required for Pac-Man to collect them. Hence, to determine the goal state, the current state needs to be constantly updated with the remaining food.

Solution Approach

Now that the Pac-Man can collect multiple dots. My solution builds on top of the approach used in Question 1(b) where the Pac-Man will utilize the A* search algorithm to identify the shortest sequence of actions to collect the closest dot. However, instead of terminating the search once the closest dot is collected, the search will re-initialize the state space with the Pac-Man's coordinate after collecting the previous closest dot and update the remaining dots left to be collected. The same search will then repeat the process of identifying the next shortest sequence of actions for Pac-Man to collect the closest remaining dots. This approach ensures that Pac-Man will prioritize the collection of the closest dot first, so that redundant actions can be avoided such as retracing the same path to collect an unclaimed dot which will increase the cost and lower the overall score.

Pseudocode:

```
Function id_astar_loop_body(problem: q1c_problem, data: IDAStarData)
{
        ...Standard A* search operation
        If Pac-Man's coordinate is in one of the dot's coordinate in the current state
        {
                Update remaining dot of current state.
                Track accumulated action from current state.
                Reinitialize state space.
                If current state is a goal state
                {
                        Terminate the search.
                }
                Otherwise, update the open set with new initialized state.
        }
        Otherwise, continue with standard A* search operation...
}
```

There are instances where the closest remaining dot from the Pac-Man will have a larger cost than the score gained from collecting the dot. To solve this problem, I extended the current A* algorithm into an Iterative Deepening A* algorithm that determines whether the heuristic of the current node is beneficial to be expanded. If the computed heuristic value to reach the closest remaining dot is greater than or equal to the score to collect all remaining dots including the last dot, this signifies that the cost to reach the closest dot will exceed the maximum remaining score the Pac-Man can achieve. This is not an optimal decision/action for Pac-Man to take. Hence, the search algorithm will terminate to ensure an optimal score is preserved over a game win.

Pseudocode:

```
Function id_astar_loop_body(problem: q1c_problem, data: IDAStarData)
{
        Expand current node from open set, C.
        Compute minimum heuristic value from C to all remaining_dots, H.
        If H >= (number_of_remaining_dots * 10) + 500
        {
                Terminate the search.
        }
        Otherwise, continue with standard A* search operation...
}
```

Note: The provided layouts do not contain such instances, so analysis cannot be made but theoretical reasoning should suffice.

Question 2

Problem Description

The problem wants us to develop an agent that can maximize the score of Pac-Man through collecting multiple dots across the grid, collecting pellets to consume scared ghost, winning the game and avoiding the ghost from consuming Pac-Man.

Solution Approach

To solve this problem, I will be using the alpha-beta search algorithm where the Pac-Man will represent the maximizer agent and the ghost will represent the minimizer agent. The goal of this search is to return the most optimal action the current state of Pac-Man can make, assuming the ghost is making the best moves to minimize the score of Pac-Man. The optimality of this action is determined through the depths of states computed during the search to identify the successor states of Pac-Man with the highest possible evaluation score while assuming each ghost plays optimally to limit the evaluation score after Pac-Man makes a move.

There are no fixed number of minimizer agents. Therefore, I extended the alpha-beta search algorithm to track the current index of the agent that is making an action to determine whether it is a minimizer or maximizer agent.

Unlike games like a 3x3 Tic-Tac-Toe with limited/minimal number of possible game states, our Pac-Man game can be executed in different layouts and can consist of multiple numbers of

minimizer agents which will increase the number of possible states that needs to be considered during our alpha-beta search. Therefore, it is not time-optimal to perform alpha-beta search until the goal state for each updated action made by Pac-Man. To solve this problem, the alpha-beta search will be limited to a fixed number of depths with a trade-off for increased search time for suboptimal decisions due to limited knowledgebase.

To incorporate both depth and index to the alpha-beta algorithm, the pseudocode below shows how the depth value is only updated after the maximizer and all minimizer agents make a move. Alongside updating the current agent index to repeat the search for the next depth.

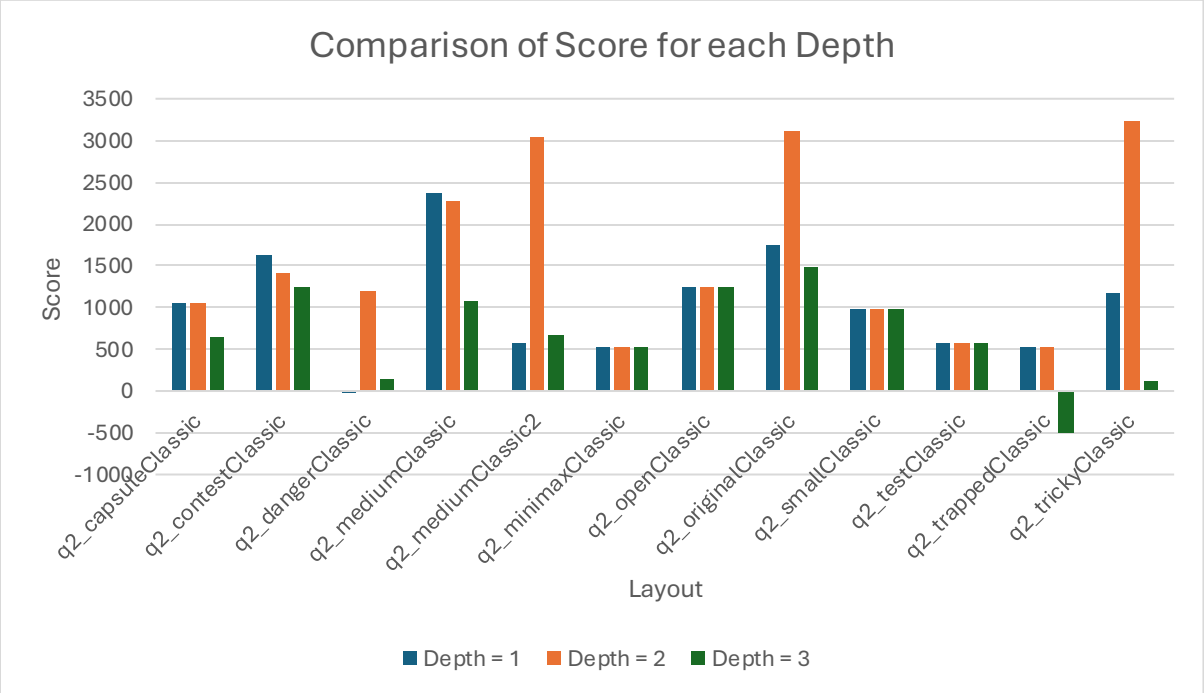
Pseudocode:

```
Function alpha_beta_search(game_state, current_index, current_depth, alpha_value, beta_value)
{

        If current_index exceeds number of agents
        {
                Increment current_depth.
                Reset current_index.
        }
        Continue with standard alpha-beta search operation...
}
```

When choosing the depth of the alpha-beta search, we need to ensure that the search time falls within the timeout limit. I have trialed different depths and realized that any depth above 3 will result in timeouts across multiple layouts especially in layouts that are larger due to the increase in state space. Therefore, I only included the depth that are smaller than or equal to 3 in my analysis.

Analysis of Depth choice:



From the result above we can deduce that the depth of 1 had consistent scores overall. However, the lack of depth resulted in Pac-Man not being capable of identifying nearby pellets to consume ghosts and maximize score. Likewise, in some layouts like q2_dangerClassic, the

alpha-beta search did not perform a deep enough search in corridor areas resulting in Pac-Man being trapped and losing. For the depth of 3, I realized that Pac-Man was wasting a lot of cost-actions by avoiding the ghost instead of collecting food pellets, this resulted in a lower score compared to other depths due to Pac-Man being capable of identifying ghost in further states which decreased the evaluation score for those states and prioritized Pac-Man to avoid ghost instead of prioritizing the score. For the depth of 2, I found the balance between avoiding ghost and prioritizing the score for Pac-Man. In this depth, I realized that Pac-Man prioritizes the collection of dots while avoiding ghost from an optimal distance that does not result in unnecessary cost-actions from avoiding. With the increase of depth from depth=1, Pac-Man can identify nearby pellets to consume ghost and maximize score which is the reason why there is a big spike in score for certain layouts against other depths.

One of the main issues of using the default evaluation function, is that it only considers the game score of the specified state and we know that our Pac-Man is limited to only a 2-depth search. This implies that Pac-Man cannot identify and evaluate states that are above 2-depth from the current Pac-Man state. Therefore, when all possible states that are 2-depth from the current state of Pac-Man have the same evaluation score, the alpha-beta search will not be able to provide an optimal action that can maximize the Pac-Man's score. This instance typically occurs when there is no food for all states 2-depth around the Pac-Man's current state and this leads to the same evaluation score since there are no food that can be collected to increase the Pac-Man's score and Pac-Man will repeatedly perform the same sequence of action which reduces the overall score of Pac-Man.

To counteract this problem, I decided to implement the Manhattan heuristics to my score evaluation function to ensure that dots outside of the 2-depth can still affect the evaluated score for each state. This was a great idea, but I realized that there are instances where different states can still have the same Manhattan distance for the closest dot. To solve this problem, I decided to compute the true cost/distance required for each state to reach the closest dot. This approach ensured that the evaluation score computed for each state will be unique which allows the alpha-beta search to return an optimal action for Pac-Man to collect the closest dot despite having the same game score for all 2-depth states.

With the foundation set in place, I wanted to further optimize my evaluation function so that my Pac-Man can consider different parameters that can increase the overall game score instead of only focusing on collecting all dots in the grid.

I devised two methodologies; one limits the evaluation function to only considering the parameters that directly increase the game score while the other also considers parameters that can indirectly affect the game score.

Methodology 1)

In this methodology, I limit the parameters to the score of the current game state, the true cost to reach the closest dot from the current game state and the true cost to reach the closest scared ghost from the current game state. Collecting dots and scared ghosts are the only actions that can directly increase the game score.

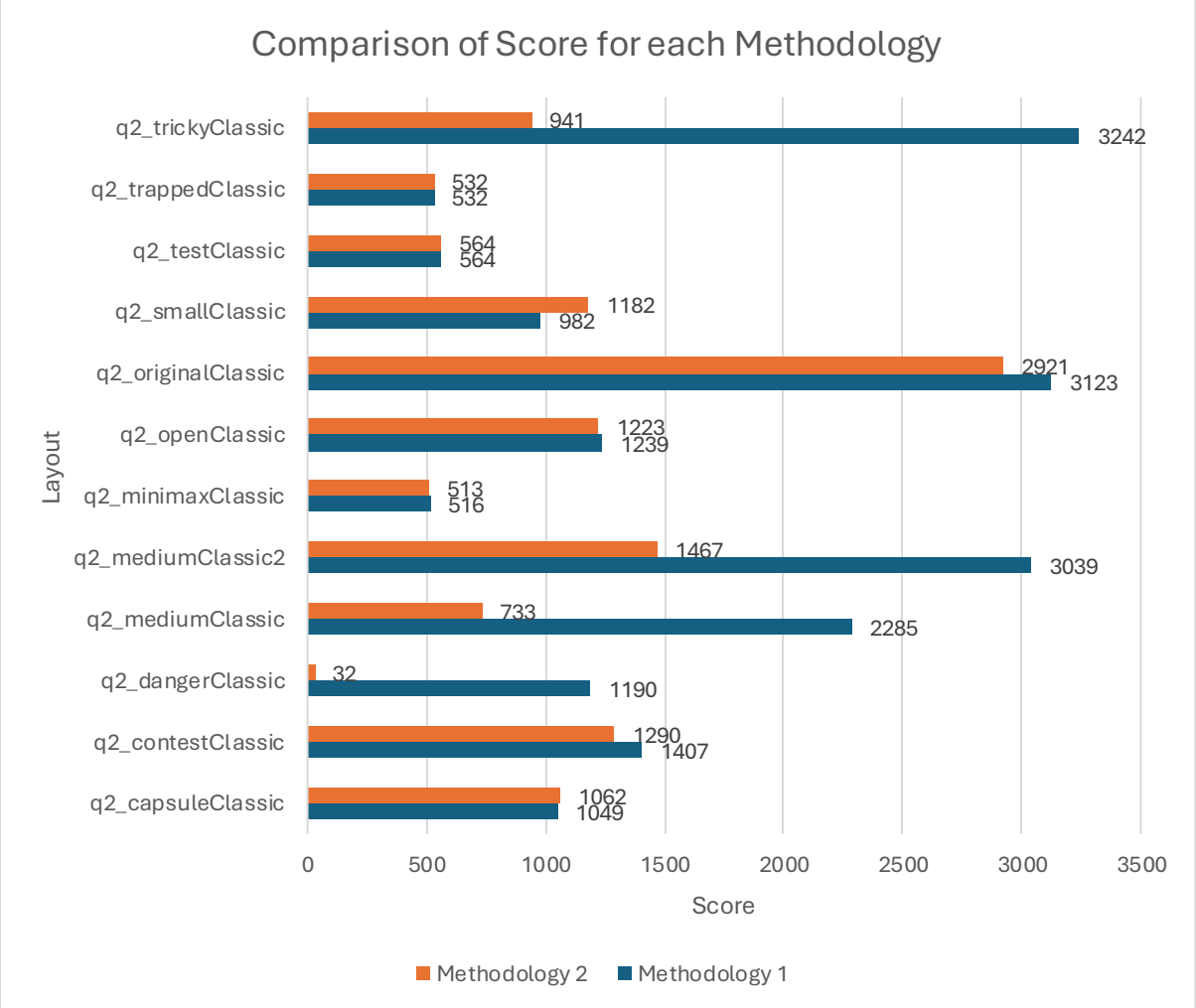
Pseudocode:

```
Function score_evaluation_function(current_game_state)
{
        Iterate over each ghost state and append the coordinate of the ghost if it is currently scared.
        If there are scared ghost in the grid
        {
                Compute the true cost to the closest scared ghost, T.
        }
        Otherwise
        {
                Compute the true cost to the closest dot, T.
        }
        Return the score of the current game state and negative of T as a tuple.
}
```

Methodology 2)

In this methodology, I considered additional parameters on top of Methodology 1 that indirectly increase the game score such as the position of each ghost relative to Pac-Man. This allows Pac-Man to evaluate the position of ghost without the ghost being within range of the search depth. This indirectly increases the game score by making Pac-Man play safely. Another parameter is the position of each capsule relative to Pac-Man. This allows Pac-Man to collect capsules that can turn ghosts into scared state, which indirectly allows Pac-Man to increase the score by consuming scared ghosts but consuming capsules itself does not increase the score.

Analysis of Evaluation Function:



From the results above, we can observe that the methodology with limited parameters has overall higher average scores compared to the methodology with more parameters. The reason behind these results is because the evaluation scores computed in methodology 1 are more consistent than the evaluation scores computed in methodology 2. This is because, the limited number of parameters allowed the evaluation function to prioritize states that very closely aligns with our desired parameters whereas the higher number of parameters had multiple states with similar/equivalent evaluation scores due to the satisfaction of different parameters from each state. This resulted in many redundant actions and early-loss from Pac-Man trying to make high-risk and high-reward actions such as collecting capsules to consume ghost. For instance, in layouts such as q2_trickyClassic and q2_dangerClassic, methodology 2 performed very badly due to early-loss from Pac-Man prioritizing pellets, but these layouts have many corridors which resulted in Pac-Man being trapped.

However, there were also layouts where the score is higher for methodology 2 such as q2_smallClassic because Pac-Man can make riskier plays to maximize score when the layouts provide Pac-Man with more states to escape freely. Overall, I decided to solve this problem with methodology 1 because the scores achieved over multiple runs of the game were very consistent even when the behavior of the ghost is random.