FIT 3080: Artificial Intelligence
Report for Assignment-1: Search

Paarth Sharma
Student ID: 32601905

Introduction

This report analyses and details the development of search agents for different puzzles and scenarios for the given Pacman game. The assignment is split into two parts, both relating to different search techniques and algorithms.

Question 1 entails single agent search algorithms to find optimal solutions to three types of problems, one which involves finding the optimal path through the maze to a single food pellet, one involving pellets at different corners and the last one that had food at places other than the corners. This is single agent search as there are no ghosts and Pacman cannot die in any way other than the code timing out.

Question 2 on the other hand involves multi-agent adversarial search in where we must consider ghosts that are out to kill Pacman if possible and reduce its score as much as possible. There were two parts to this question as well, where the number of ghosts is two and four respectively. Unlike Question 1, Pacman can die in these modes and must prevent itself from doing so while also maximizing the potential score.

Question 1
1a. Finding Optimal Path to one food pellet.

Overview
The A* algorithm was used to develop a Pacman agent that finds the best possible path to a solitary goal while going around a maze.

Problem Formulation
The file q1a_problem.py contains code for the problem formulation for the approach and the method to get successors for a selected state. The class q1a_problem involves the following class variables:

self.startingGameState –> This is the starting state of the problem.
self.goal –> This the coordinates of the food pellet represented as a tuple in the form of (x, y).
self.startState –> This the starting position of Pacman also represented as a tuple in the form of (x, y)
The method **isGoalState** is used to check if the game is at goal state.

The method **getSuccessors** returns a list of successors which are represented as tuples in the form (successorState, action, cost) where all possible successors are generated by performing possible actions (North, South, East, West) onto the current Pacman position if there is no wall at the prospective successor. The cost of each such move is 1.

Approach to Solving the given Problem.

Pseudocode
```
Function q1a_solver(problem):
        Function a_star(problem):
                Get starting state and the goal of the problem.
                Set F(start) = 0
                Initialise a priority queue for open nodes.
                Initialise a set for closed nodes.
                Push a tuple containing starting state, with priority F(start)
                While openNodeList is not empty:
                        Pop an open node
                        If goal is reached:
                                return path
                        if current node is not closed:
                                add node to closed nodes
                                for each generated successor of current node:
                                        if successor is not a closed node:
                                                compute f(node) using manhattan distance and
                                                the cost till now.
                                                Also append action to the path to be taken
                                                add successor to openNode
```

```
            return None if no path is found
        return a_star(problem)
```

Alternative Approaches
Before moving forward with the A* algorithm, I considered implementing both BFS and Dijkstra algorithm for this question but then decided on implementing the A* due to the following reasons:

- A* is generally faster in computing the optimal path than both Dijkstra algorithm as it is driven by the heuristics unlike the other two uninformed algorithms.
- Also, I considered the following problems where the problem was similar but would have to incorporate minor changes and the flexibility of using heuristic with A* made my work easier.

On the other hand, the A* algorithm depends on the heuristic very much and a bad heuristic would mean that the algorithm would not be as efficient as it should be. But the pros outweighed the cons and I decided to use A* algorithm.

Experimental Evaluation
The main metrics I used to evaluate between Dijkstra and A* was the average time taken to find path for the bigMaze map as both algorithms guarantee optimality under specific conditions. [1].
The findings were that the A* algorithm took **0.009s** to find the optimal path while Dijkstra took a marginally larger **0.010s** to find one.

Conclusion
To conclude A* search offers Pac-Man an efficient and ideal way to move through a maze and find the food. It is better than the other algorithms due to its heuristic-guided nature and extensibility to future questions.

1b. Finding Optimal Path to food pellets in corners.

Overview
The A* algorithm was used to develop a Pacman agent that finds the best possible path to goals in each corner while traversing a maze of walls and turns.

Problem Formulation
The file q1b_problem.py contains code for the problem formulation for the approach and the method to get successors for a selected state. The class q1b_problem involves the following class variables:

self.startingGameState –> This is the starting state of the problem.
self.goal –> This the coordinates of the food pellet represented as a tuple in the form of (x, y).
self.startState –> This the starting position of Pacman also represented as a tuple in the form of ((x, y), (postions of all the remaining food).
The method **isGoalState** is used to check if the game is at goal state, i.e., no remaining goals.

The method **getSuccessors** returns a list of successors which are represented as tuples in the form (successorState, action, cost) where all possible successors are generated by performing possible actions (North, South, East, West) onto the current Pacman position if there is no wall at the prospective successor, and also updates remaining goals if the successor state has a food particle, by removing the goal from the successor's goals list. The cost of each such move is 1.

Approach to Solving the given Problem.

Pseudocode

```
Function q1b_solver(problem):
    Function a_star(problem):
        Get starting state and the goal of the problem.
        Set F(start) = the minimum of all goals on the board at the start
        Initialise a priority queue for open nodes.
        Initialise a set for closed nodes.
        Push a tuple containing starting state, with priority F(start)
        While openNodeList is not empty:
            Pop an open node
            If goal is reached:
                return path
            if current node is not closed:
                add node to closed nodes
                for each generated successor of current node:
                    if successor is not a closed node:
                        compute f(node) using minimimum of manhattan
                        distances and the cost till now.
                        Also append action to the path to be taken
                        add successor to openNode

        return None if no path is found
    return a_star(problem)
```

Alternative Approaches

Just like 1(a), before moving forward with the A* algorithm, I considered implementing both BFS and Dijkstra algorithm for the corners problem but then decided on implementing the A* due to the following reasons:

- A* is generally faster in computing the optimal path than both Dijkstra algorithm as it is driven by the heuristics unlike the other two uninformed algorithms, especially with more than one goal on the board.

The implementation for this problem was very similar to the implementation in 1(a) where the implementation was very successful in finding the optimal path and was easily adapted to these problems.

On the other hand, the A* algorithm depends on the heuristic very much and a bad heuristic would mean that the algorithm would not be as efficient as it should be. But the pros outweighed the cons and I decided to use A* algorithm.

Also, I tried using Euclidean distance as my heuristic for the code but found that the minimum of Manhattan distances to foods worked the best and fastest while also being an admissible and consistent heuristic for a single agent search in a maze. And as we were under a time constraint regarding the sheer number of pellets and the timeout, I was looking to get a path as fast as possible.

Other Heuristics I considered were [2]:

- Sum of Euclidean Distances
- Sum of Manhattan Distances to all goals

But I used the sum of Manhattan distances as it worked the fastest.

Conclusion
To conclude A* search offers Pac-Man an efficient and ideal way to move through a maze and find the food. It is better than the other algorithms due to its heuristic-guided nature and extensibility to similar future problems.

1c. Finding Optimal Path to multiple food pellets.

Overview
The A* and Beam Search algorithms were used to develop a Pacman agent that finds the best possible path to several goals while traversing a maze of walls and turns.

Problem Formulation
The file q1c_problem.py contains code for the problem formulation for the approach and the method to get successors for a selected state. The class q1c_problem is basically identical to the one in 1(b) and involves the following class variables:

self.startingGameState –> This is the starting state of the problem.
self.goal –> This the coordinates of the food pellet represented as a tuple in the form of (x, y).
self.startState –> This the starting position of Pacman also represented as a tuple in the form of ((x, y), (postions of all the remaining food).
The method **isGoalState** is used to check if the game is at goal state, i.e., no remaining goals.

The method **getSuccessors** returns a list of successors which are represented as tuples in the form (successorState, action, cost) where all possible successors are generated by performing possible actions (North, South, East, West) onto the current Pacman position if there is no wall at the prospective successor, and also updates remaining goals if the successor state has a food particle, by removing the goal from the successor's goals list. The cost of each such move is 1.

Approach to Solving the given Problem.
Pseudocode
```
Function q1c_solver(problem):
    Function a_star(problem):
        Get starting state and the goal of the problem.
        Set F(start) = the minimum of all goals on the board at the start
        Initialise a priority queue for open nodes.
        Initialise a set for closed nodes.
        Push a tuple containing starting state, with priority F(start)
        While openNodeList is not empty:
            Pop an open node
            If goal is reached:
                return path
            if current node is not closed:
                add node to closed nodes
                for each generated successor of current node:
```

```
                        if successor is not a closed node:
                            compute f(node) using minimimum of manhattan
                            distances and the cost till now.
                            Also append action to the path to be taken
                            add successor to openNode

        return None if no path is found



    def beam_search(problem):
        initialise width of the beam
        initialise starting state, open list as a priority queue, visited list
        while open list is not empty:
            take the best n(width) elements of the open list to make a new beam
            re initialise open list to empty Priority Queue
            for each node in this new beam:
                if game is at goal state:
                    return path

                get successors to current state
                for each generated successor of current node:
                    if successor is not a closed node:
                        compute f(node) using minimimum of manhattan
                        distances.
                        Also append action to the path to be taken
                        add successor to openNode and visited
            return None

    If number of food <= 25:
            Return a_star(problem)

        Else:
            Return beam_search(problem)
```

Alternative Approaches

Just like 1(a) and 1(b), before moving forward with the A* algorithm, I considered implementing both BFS and Dijkstra algorithm for the problem but then decided on implementing the A* due to the following reasons:

- A* is generally faster in computing the optimal path than both Dijkstra algorithm as it is driven by the heuristics unlike the other two uninformed algorithms, especially with more than one goal on the board.

The implementation for this problem was theoretically very similar to the implementation in 1(b) where the implementation was successful in finding the optimal path and was easily adapted to this problem.

But as the sample size of the bigger problems tended to be huge and A* was taking a lot of time to come up with the best possible path. I chose to compromise a bit on optimality for speed as I implemented the Beam Search Algorithm [3] [4] to have a path ready to get every goal for the cases with a high number of goals. I also tried implementing BFS and DFS for these cases but as with A* they had too many nodes to expand and did not even return a path in time for a lot of cases.

The smaller games (with less than or equal to 25 food pellets) still undergo the A* algorithm to get the most optimal results but in cases where such an approach requires high time and resource, I thought that it was best to stick with Beam Search and find somewhat suboptimal but still efficient paths.

Conclusion
To conclude, A* search offers Pac-Man an efficient and ideal way to move through a maze and find the food. It is better than the other algorithms due to its heuristic-guided nature and extensibility to similar future problems. But due to higher sample size of some problems, we can use the beam search algorithm to find suboptimal paths within the given timeout limit.

Question 2
2a&b. Implementing Adversarial search.

Overview
These questions involved the actual classic game of Pacman where the Pacman must maximise the score and avoid ghosts which are out to kill it. Question 2a uses the minimax algorithm to find the best possible values while Q2b uses Alpha Beta pruning which cuts down on execution time.

Problem Formulation
The classes Q2A_Agent and Q2B_Agent are designed to use the adversarial search algorithms to find the best possible value considering both types of agents. And a customised evaluation function was used as the heuristic. The classes mainly use the following class variables:

self.index –> Pacman's index (always 0)
self.evaluationFunction –> function to evaluate game state
self.depth–> depth of minimax
The method getAction has a recursive method minimax that is implemented to find the best action possible.

Approach to Solving the given Problem.
Pseudocode (2a. Minimax)
```
Function getAction(self, gameState):
    Function minimax(agentIndex, depth, gameState):
        If game has been won or lost or max depth is reached by Pacman:
            Return evaluationFunction(gameState), ""
        Get Legal Actions of agent
        If no legal action:
            Return evaluationFunction(gameState), ""

        Initialise nextAgent
        If pacman's turn:
            Best val is set to negative infinity
```

```
            bestActions = singleton list with only element as [“”]
            for each action in legal Actions:
                generate successor state
                value, _ = minimax(nextAgent, (depth -1 if pacman, else depth),
                successor)
                get maximum possible value and append action to list
                if there is a tie in value, use random/choice to select random from
                the action list.
                Return value, and the random choice generated from action
        Else:
            Best val is set to infinity
            bestActions = singleton list with only element as [“”]
            for each action in legal Actions:
                generate successor state
                value, _ = minimax(nextAgent, (depth -1 if pacman, else depth),
                successor)
                get minimum possible value and append action to list
                if there is a tie in value, use random/choice to select random from
                the action list.
                Return value, and the random choice generated from action

    Return the best Action generated by minimax algorithm
```

Pseudocode (2b. Alpha-Beta Pruning)

```
Function getAction(self, gameState):
    Function alpha_beta(agentIndex, depth, gameState, alpha, beta):
        If game has been won or lost or max depth is reached by Pacman:
            Return evaluationFunction(gameState), ""
        Get Legal Actions of agent
        If no legal actions:
            Return evaluationFunction(gameState), ""

        Initialise nextAgent
        If pacman’s turn:
            Best val is set to negative infinity
            bestActions = singleton list with only element as [“”]
            for each action in legal Actions:
                generate successor state
                value, _= minimax(nextAgent, (depth -1), successor, alpha, beta)
                get maximum possible value and append action to list.
                set alpha as maximum of alpha and the value.
                BREAK if alpha > Beta
                if there is a tie in actions having same value, use random/choice
                to select random from the action list.
                Return value, and the random choice generated from action

        Else:
            Best val is set to infinity
            bestActions = singleton list with only element as [“”]
```

for each action in legal Actions:
    generate successor state
    value, _= minimax(nextAgent, (depth -1 if pacman, else depth), successor)
    get minimum possible value and append action to list
    set beta as minimum of beta and value
    BREAK if alpha > Beta
    if there is a tie in actions having same value, use random/choice to select random from the action list.
    Return value, and the random choice generated from action

Initialise time limit and best action.
For depth from 1 to given depth: #Iterative deepening
    If time elapsed > time limit:
        Break
    Call alpha_beta and update best action.
Return the best Action generated by alpha_beta algorithm.

Optimisations for AlphaBeta

To combat problems introduced by the time constraint, I have taken an iterative deepening approach where the algorithm is run for all depths from 1 to the given depth, and the loop breaks if the time limit is exceeded to ensure that the code does not get stuck and lead to time out. Depth is also put as 6 because with a depth of 3 the Pacman has limited vision, and this depth is also possible because of the time limits implemented.

Evaluation Function for Alpha-Beta

The evaluation function in Q2b_agent calculates the closest distance to a pellet, capsule, and checks if a ghost is in proximity. It returns the score as a heuristic to be searched, I have put additional checks to ensure that the Pacman has more incentive to go after the ghost when a capsule is eaten.

Advantages and Disadvantages [5]

Minimax

Advantages

- Guarantees Optimality
- If tree is explored fully, select best possible move for the max agent.
- Straightforward to implement.

Disadvantages

- Examines every node possible, which can be inefficient.
- Time complexity id O(b^d) where b is branch factor and d is the depth, can be slow if b becomes too high.
- Optimality also depends on heuristic.
- High Space Complexity as well, because entire trr till max depth is generated.
- Limited depth may lead to suboptimal paths as tree may not be explored fully.

Alpha-Beta Pruning

Advantages

- Cuts down on execution time and the number of actual nodes evaluated.
- If tree is explored fully, finds the same optimal move as minimax, despite pruning.

- Straightforward to extend minimax to include pruning.
- Generally requires less memory as well, as sub-trees can be pruned.

Disadvantages

- Bit more complex to understand than standard minimax
- May not be optimal and depends on heuristic a lot.
- Worst Case time complexity is still O(b^d), although the algorithm is generally faster than minimax.

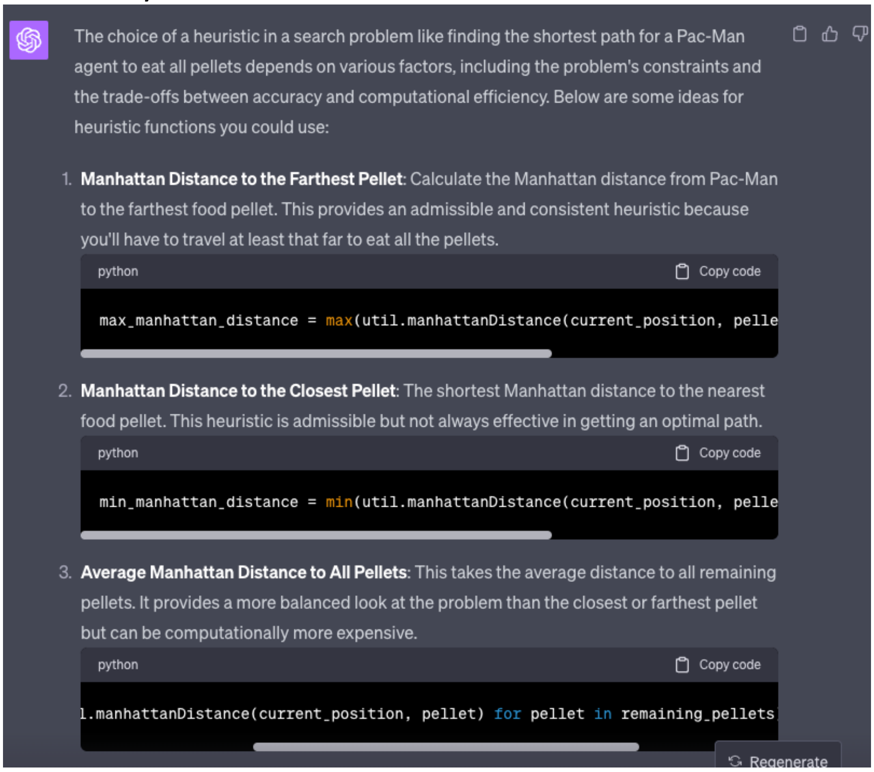
Conclusion

Alpha Beta Pruning offers a more computationally efficient means for the Pacman agent to operate in an multiagent adversarial environment. This ensures that Pacman and Ghosts make optimal moves, and that excessive time is not spent on inconsequential nodes which would be the case in standard minimax. The custom evaluation function also enhances the performance of the code, and iterative deepening is used enforce time constraints.

REFERENCES

[1] D. Harabor, "FIT 3080 Lecture Slides ," *Moodle*, Jul. 2023. (accessed Sep. 2023).

[2] OpenAI. (2023). *ChatGPT* [Large language model]. https://chat.openai.com
Prompt: What can be good heuristic for pacman search agent
Answer by GPT:



The choice of a heuristic in a search problem like finding the shortest path for a Pac-Man agent to eat all pellets depends on various factors, including the problem's constraints and the trade-offs between accuracy and computational efficiency. Below are some ideas for heuristic functions you could use:

1. **Manhattan Distance to the Farthest Pellet**: Calculate the Manhattan distance from Pac-Man to the farthest food pellet. This provides an admissible and consistent heuristic because you'll have to travel at least that far to eat all the pellets.

python Copy code

```python
max_manhattan_distance = max(util.manhattanDistance(current_position, pelle
```

2. **Manhattan Distance to the Closest Pellet**: The shortest Manhattan distance to the nearest food pellet. This heuristic is admissible but not always effective in getting an optimal path.

python Copy code

```python
min_manhattan_distance = min(util.manhattanDistance(current_position, pelle
```

3. **Average Manhattan Distance to All Pellets**: This takes the average distance to all remaining pellets. It provides a more balanced look at the problem than the closest or farthest pellet but can be computationally more expensive.

python Copy code

```python
l.manhattanDistance(current_position, pellet) for pellet in remaining_pellets
```

Regenerate

4. **Euclidean Distance**: Sometimes the Manhattan distance is too restrictive. The Euclidean distance can provide a more accurate (but computationally intensive) estimate.

python   Copy code

```python
max_euclidean_distance = max(util.euclideanDistance(current_position, pelle
```

5. **Count of Remaining Pellets**: Simply counting the remaining pellets provides an admissible and consistent heuristic, albeit a very naive one. It will significantly underestimate the cost but is very fast to compute.

python   Copy code

```python
remaining_pellet_count = len(remaining_pellets)
```

6. **Maze Distance to Farthest Pellet**: Compute the actual shortest path distance to the farthest pellet. This would be the most accurate but is also computationally expensive.

Depending on your specific requirements, you can also consider combining multiple heuristics to get a more balanced and accurate estimation. But always remember to ensure that your heuristic is admissible (never overestimates the cost) and consistent (satisfies the triangle inequality) to guarantee finding an optimal solution when using algorithms like A*.

Regenerate

[3] OpenAI. (2023). *ChatGPT*) [Large language model]. https://chat.openai.com
Prompt: as A star, bfs and dfs arent really working, according to you what should I implement for a board with around 200ish food pellets?
Answer by GPT

For a board with around 200ish food pellets, traditional algorithms like A*, BFS, and DFS might not be the most efficient because of the large state space. You'd need a heuristic that's both admissible and consistent to ensure optimality while also being fast. But even with a great heuristic, A* could still take a long time given the large number of states.

Here are some suggestions:

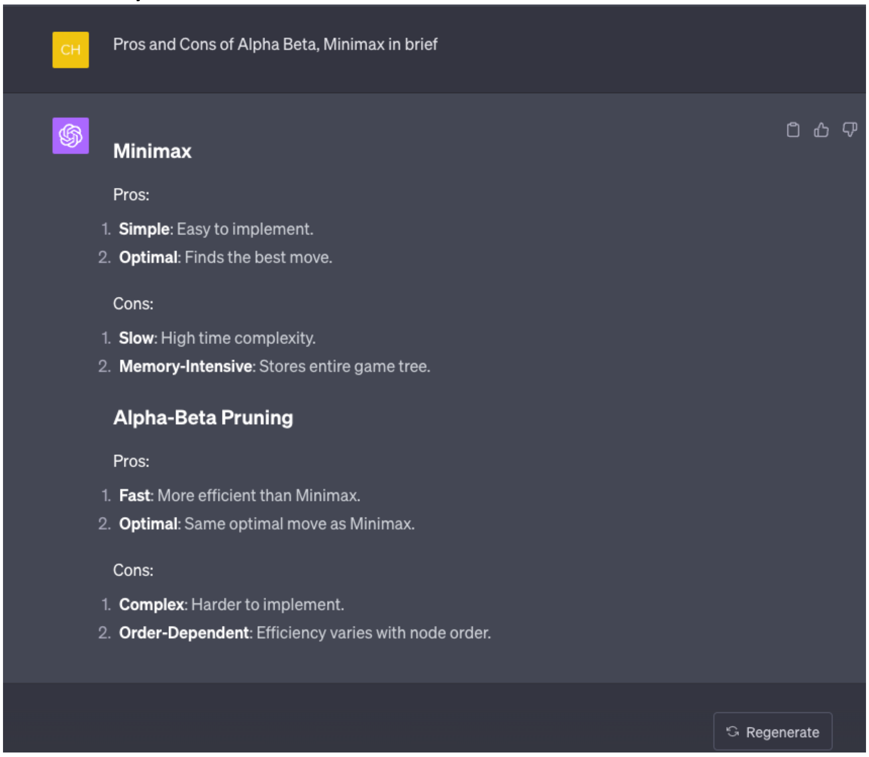
1. **Approximate Algorithms**: If you are willing to compromise a bit on optimality for speed, you might look into approximate algorithms like Greedy Best First Search or even Beam Search, which you have already tried.
2. **Monte Carlo Tree Search (MCTS)**: This is a more advanced algorithm that has been used in various game-playing applications and could be applied here. It doesn't guarantee optimality but can offer a good balance between exploration and exploitation, finding reasonably good paths in large state spaces.
3. **Divide and Conquer**: Divide the board into smaller sections and solve each one individually, then concatenate the paths. Be cautious with this approach as it doesn't guarantee a minimal path.
4. **Heuristic Tweaks**: If you still want to use A*, consider a more complex heuristic that takes into account more features of the board. Just remember, the more complex the heuristic, the longer it will take to compute, so it's a trade-off.
5. **Real-time Search Algorithms**: Algorithms like RTA* or LRTA* are designed to operate in large state spaces by only considering information available in a local neighborhood around the agent. These might be applicable depending on the specifics of your problem.
6. **Dynamic Programming**: Algorithms like Value Iteration could be used if you can clearly define the utility function for the states. This can sometimes be faster for problems with large state spaces but where each state has only a small number of successors.

If optimality isn't a hard requirement, you might find Beam Search or Greedy Best-First Search to be the most straightforward to implement. These algorithms can be significantly faster than A* at the cost of not always finding the most optimal path.

Regenerate

[4] “Introduction to Beam Search Algorithm,” *GeeksforGeeks*, Jul. 15, 2021. https://www.geeksforgeeks.org/introduction-to-beam-search-algorithm/

[5] OpenAI. (2023). *ChatGPT* [Large language model]. https://chat.openai.com
Prompt: Pros and Cons of Alpha Beta, Minimax in brief
Answer by GPT:



(I acknowledge the use of ChatGPT (https://chat.openai.com/) to generate materials for background research, debugging and self-study in the drafting of this assessment.
The output from the generative artificial intelligence was adapted and modified for the final response.)