SEARCH IN THE PACMAN ENVIRONMENT

Vasiliki Varvara Plevridi 100557506

May 2025

Contents

1 Introduction

This project focuses on the comparison of different search algorithms—specifically DFS, BFS, UCS, and A* Search with various heuristics—implemented within the context of a Pacman maze problem. The following report presents their performance across several mazes that I designed, along with a discussion of their behavior and efficiency.

The second part of the project shifts focus to the comparison of different agents whose goal is to collect all the food dots in the maze. A similar analysis is conducted, evaluating each agent's strategy and performance, followed by overall conclusions drawn from the observations.

2 Understanding search algorithms

2.1 Class `SearchProblem`

The `SearchProblem` class outlines the structure of a generic search problem but does not implement any of its methods. It serves as an abstract base class. The following functions are defined as placeholders and must be implemented in subclasses:

- `getStartState()`: Returns the start state of the search problem.
- `isGoalState(state)`: Evaluates whether a given state is a valid goal state.
- `getSuccessors(state)`: Returns a list of successor states from the current state, along with the action required to reach them and the associated cost.
- `getCostOfActions(actions)`: Returns the total cost of a specific sequence of actions.

The reason these methods are analyzed is because they are the ones that connect the `PositionSearchProblem` class with all the search algorithm implementations in `search.py`.

Also, it is really important to analyze the structure of the successors. This is the fundamental structure used in the general graph search algorithm, and therefore in all search algorithms that follow. The successor function returns a list of triples in the form:

```
(successor_state, action, step_cost)
```

- `successor_state`: the resulting state after applying an action
- `action`: the action taken to reach that state (e.g., North, South, etc.)
- `step_cost`: the cost of taking that action, usually 1

2.2 Function `generalGraphSearch`

The **generalGraphSearch** function is a generic and flexible graph search framework that abstracts the common logic shared by multiple search algorithms, including Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS), and A*. It operates on a simple principle: the method of exploration depends entirely on the structure used to store the frontier. The frontier is essentially a list of unexplored paths that are waiting in a queue to be explored.

The algorithm proceeds as follows:

1. **Initialize the frontier:** Insert the start state into the frontier as a path represented by a list of tuples. Each tuple contains a state, the action that led to it, and the cost of that action:

$$\text{path} = [(state_0, action_0, cost_0), (state_1, action_1, cost_1), \ldots]$$

2. **Create an empty list of visited states:** This prevents the algorithm from re-exploring already visited nodes and helps avoid cycles.

3. **Loop until a solution is found or the frontier is empty:**

   a) Pop the next path from the frontier using the structure's priority logic.

   b) Extract the current state from the last tuple in the path.

   c) If the current state is the goal, return the list of actions that led to it.

   d) If the current state has not been visited:

   - Mark it as visited.
   - For each successor of the current state:
     - Create a new path by appending the successor to the current path.
     - Push this new path into the frontier.

4. **Failure case:** If the frontier becomes empty without reaching the goal, return failure.

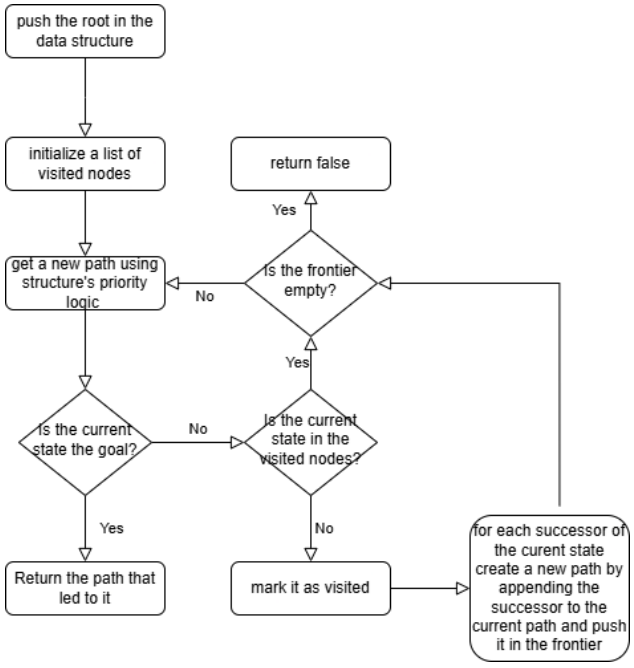


Figure 1: Flowchart of General Graph Search Algorithm

2.3 Key Concept: Frontier Structure

The choice of the structure determines the type of search:

- **DFS:** Stack (LIFO)
- **BFS:** Queue (FIFO)
- **UCS:** Priority queue sorted by path cost $g(n)$
- **A*:** Priority queue sorted by $f(n) = g(n) + h(n)$

The next functions implemented in the search.py file—depthFirstSearch, breadthFirstSearch, uniformCostSearch, and aStarSearch—essentially use the generalGraphSearch function with the required structure for each case. All of these algorithms share a common core logic, which is why the general function is suitable for all of them.

It is also important to mention that although each search algorithm is implemented in the search.py file, the classes and the helper functions that are essential for the implementation are included in the utils.py file.

2.4 Depth-First Search (DFS)

Depth-First Search uses a **stack** (LIFO) to control the frontier, meaning the most recently discovered paths are explored first. This leads the algorithm to go deep into one branch before exploring others.In implementation, the function sets the structure to be a stack by initializing an instance of the Stack class (included in the utils.py file) and using it as input structure to the general graph search algorithm.

2.5 Breadth-First Search (BFS)

Breadth-First Search uses a **queue** (FIFO) to manage the frontier. This ensures that the algorithm expands the shallowest nodes first, exploring the state space level by level. In implementation, the function sets the structure to be a queue by initializing an instance of the Queue class (included in the utils.py file) and using it as input structure to the general graph search algorithm.

2.6 Uniform-Cost Search (UCS)

Uniform-Cost Search uses a **priority queue** where paths are sorted by the cumulative cost $g(n)$ from the start node to the current state. This makes UCS equivalent to Dijkstra's algorithm. UCS depends on the problem's `getCostOfActions` function to evaluate the cost of each path correctly.

2.7 A* Search

A* Search combines the ideas of UCS with a heuristic function. It uses a **priority queue** where the priority is defined as:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost from the start to node n, and $h(n)$ is an estimate of the cost from n to the goal. In the implementation, the algorithm uses a pair of priority functions: one for total cost $f(n)$ and one for tie-breaking based on the heuristic $h(n)$.

3 Problem 1 Part 1: empirical study of algorithms

The aim of this section is to understand the problem where Pacman has to collect a fixed food dot in the maze. This functionality is implemented in the class `PositionSearchProblem`, which is defined in the `searchAgents.py` file.

3.1 Connecting `PositionSearchProblem` and Search Algorithms

The `PositionSearchProblem` class describes the actual problem to solve — such as navigating a maze, determining where Pacman starts, where the goal (food) is, and what moves are possible.

In order for Pacman to move and reach the goal, this class must be connected to the search algorithms defined in `search.py`. These algorithms are general-purpose: they do not need to know the specifics of the problem. They simply work with any problem that provides a start state, a goal test, and a way to generate successor states.

The key to connecting these two components lies in the fact that `PositionSearchProblem` is a subclass of the abstract `SearchProblem` class, which was analyzed in subsection 2.1. While `SearchProblem` defines the required interface (i.e., the methods that must be implemented), `PositionSearchProblem` provides a concrete implementation of these methods, making it suitable for solving real mazes.

This design allows the search algorithms to operate on any problem that follows the `SearchProblem` structure, including `PositionSearchProblem`, without needing to know the internal details of the problem itself.

All the functions implemented in `PositionSearchProblem` are required by the `SearchProblem` interface. These functions are essential, as they are directly used by the general search algorithms to explore the state space and find a solution. Additionally, `PositionSearchProblem` includes specific logic and internal variables to support visualization and dynamic goal detection:

- `__init__()`: Initializes the start and goal state, wall layout, and cost function.
- `self.goal`: May be dynamically detected if only one food dot exists.
- `self.visualize`, `self._visited`, `self._expanded`: Used for display and tracking during search.

These extra elements are not required by the search algorithms but are useful for visual feedback and flexibility within the Pacman environment.

3.2 Problem Analysis and State Space Representation

3.2.1 State space

A state in this problem is represented by the coordinates (x, y), which corresponds to Pacman's position at any given point in time. These positions must lie within the grid that is not blocked by walls. The `gameState` provides the `walls` array to indicate where Pacman cannot move. The part of the code below shows where this was specified:

```
x, y = state
dx, dy = Actions.directionToVector(action)
nextx, nexty = int(x + dx), int(y + dy)
if not self.walls[nextx][nexty]:
    nextState = (nextx, nexty)
```

The size of the state space depends on the dimensions of the grid (width and height of the board game) and the presence of the walls. If there were no walls, the state space size is simply the product of the grid's width and height, i.e., $width \times height$. However, in the actual game, there are walls (`self.walls` in the code). So, the state space size S can be calculated as:

$$S = \text{Width} \times \text{Height} - \text{Number of wall positions}$$

Thus, the state space size is reduced by the number of blocked positions caused by the walls. This is not the final state space size. The game we can have multiple food items so if there are N valid positions where food can be placed, there are 2^N possible configurations for the food (since each food position can either contain food or not). Thus, the correct state space size is:

$$\texttt{State space size} = S \times 2^N$$

3.2.2 Initial and goal states

The initial state is given by the `startState` in the code:

```
self.startState = gameState.getPacmanPosition()
if start != None: self.startState = start
```

So, the initial state is simply the coordinates (x, y) of where Pacman starts on the grid.

About the goal state, is the position Pacman must reach in order to finish the task. In the code, its is defined as:

```
if (gameState.getNumFood() == 1):
    foods = gameState.getFood()
    npfood = np.array(list(foods))
    thefood = np.where(npfood == True)
    self.goal = tuple(tuple(x)[0] for x in thefood)
else:
    self.goal = goal
```

The code retrieves the position of the food item and assigns it to `self.goal`. Therefore, the goal state is the position (x, y) of the food.

3.2.3 Operators

Operators are the possible actions Pacman can take. In this is problem they are:

```
NORTH, SOUTH, EAST, WEST
```

They can be found directly in the `getSuccessors` method:

```
for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
```

An action is applicable if moving in that direction does not lead to a wall. This is checked in the code by:

```
if not self.walls[nextx][nexty]:
    nextState = (nextx, nexty)
    cost = self.costFn(nextState)
    successors.append( ( nextState, action, cost) )
```

And finally, as we can see from the code above, applying an action results in a new state — the position Pacman reaches after the move. The cost is recalculated, and the new state is added to the list of successors.

3.3 Comparative Analysis of Search Algorithms

In this section, four different search algorithms are presented and tested on a series of mazes that I have designed with varying sizes and layouts. Each maze is analyzed individually to examine the behavior of the search agents. At the end, a comparative summary of the results is provided, highlighting the differences in performance and efficiency among the algorithms.

3.3.1 Maze 1

This is a simple maze, designed to observe the behavior of the different search agents in a non-complex environment. As we can see, all algorithms successfully find the optimal path (i.e., the one with the lowest cost), except for Depth-First Search (DFS). DFS, as a strategy that dives deeply into one possible path until it reaches the end, tends to choose the first valid path it encounters rather than the most efficient one.

Among the rest, it is worth noting that A* Search with the Manhattan distance heuristic expands the fewest nodes, while Breadth-First Search (BFS) expands the most. This result is expected, as BFS systematically explores all possible paths at each level before proceeding, making it exhaustive but inefficient in terms of node expansion.

Regarding execution time, although one might expect a proportional relationship between the number of nodes expanded and the computation time, this is not observed. This is because each algorithm has different computational complexities, and their internal operations introduce overheads that make the relationship between time and node expansion non-linear.

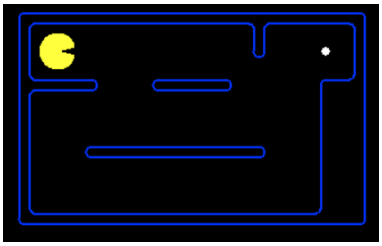


Figure 2: Maze 1

Algorithm	**Time (s)**	**Nodes Expanded**	**Path Cost**
Breadth-First Search (BFS)	0.00324	25	12
Depth-First Search (DFS)	0.00409	29	22
Uniform Cost Search (UCS)	0.00598	25	12
A* Search (Manhattan Heuristic)	0.00170	15	12
A* Search (Euclidean Heuristic)	0.01311	16	12

Table 1: Performance Comparison of Search Algorithms on Maze 1

3.3.2 Maze 2

In this maze, the goal was to observe a more specific and interesting behavior of the search algorithms. First of all, all algorithms successfully found the optimal path. It is important to note that since BFS always guarantees the shortest path in terms of step cost, when another algorithm produces the same path cost, we can conclude that it also found the optimal path.

What stands out in this case is that DFS not only found the optimal path, but did so while expanding significantly fewer nodes than all other agents — especially compared to A* Search with the Manhattan heuristic. A*, being an informed search, explores additional alternatives to ensure optimality, even if the correct path is directly available. This demonstrates the trade-off of informed search: while it is systematic and guarantees optimality, it may explore more than necessary in some cases.

On the other hand, DFS aggressively follows a single path and, in this particular instance, happened to choose the optimal one. This example highlights how DFS, under favorable conditions, can outperform more sophisticated algorithms like A* in terms of node expansion, while still achieving the same final result — a rare but insightful outcome showing the impact of search order and maze structure.

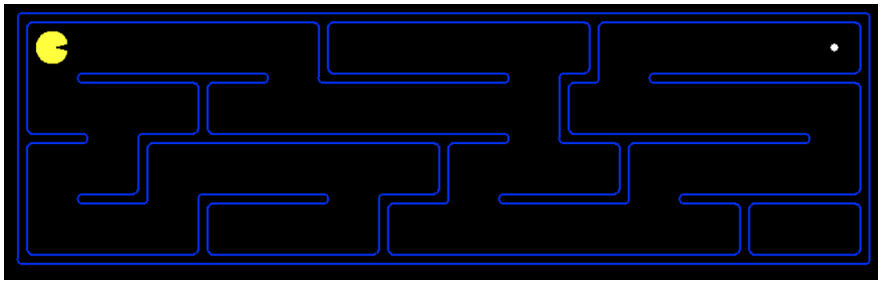


Figure 3: Maze 2

Algorithm	Time (s)	Nodes Expanded	Path Cost
Breadth-First Search (BFS)	0.01291	106	56
Depth-First Search (DFS)	0.00810	64	56
Uniform Cost Search (UCS)	0.01604	106	56
A* Search (Manhattan Heuristic)	0.01085	105	56
A* Search (Euclidean Heuristic)	0.01983	105	56

Table 2: Performance Comparison of Search Algorithms on Maze 2

3.3.3 Maze 3

This is a very simple maze, designed to highlight the similarities and core behaviors of the different search agents. It is immediately evident that A*, which combines both cost and heuristic estimation, achieves the best efficiency in terms of nodes expanded and time. Among the A* variants, both the Manhattan and Euclidean heuristics perform equally well, since the simplicity of the layout minimizes the difference between them.

On the other hand, we observe that Uniform Cost Search (UCS) continues to expand the same number of nodes as Breadth-First Search (BFS), as seen in previous mazes. Although Dijkstra's algorithm (UCS) and BFS are built upon different principles — one prioritizing cumulative cost and the other exploring level by level — they behave identically in environments where all actions have equal cost. In such cases, UCS essentially degenerates into BFS, explaining the identical number of expanded nodes and path cost.

DFS, in contrast, expands more nodes than necessary, even in this small maze, due to its depth-oriented exploration. Although it still finds the optimal path in this instance, it does so less efficiently, reaffirming the inconsistency of DFS when optimality and efficiency are both desired.

This maze reinforces the expected behavior of each algorithm and demonstrates how A* can outperform others in both time and node expansion, even when all algorithms find the same optimal solution.

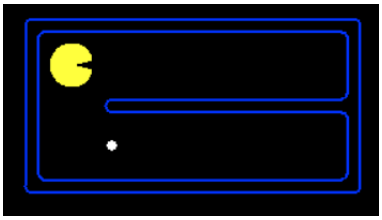


Figure 4: Maze 3

Algorithm	Time (s)	Nodes Expanded	Path Cost
Breadth-First Search (BFS)	0.00212	5	3
Depth-First Search (DFS)	0.00291	10	3
Uniform Cost Search (UCS)	0.00170	5	3
A* Search (Manhattan Heuristic)	0.00136	3	3
A* Search (Euclidean Heuristic)	0.00243	3	3

Table 3: Performance Comparison of Search Algorithms on Maze 3

3.3.4 Maze 4

Our initial goal in this maze was to demonstrate that BFS would visit fewer nodes than A* using the Manhattan distance heuristic. To achieve this, I created a maze with multiple dead ends and paths with higher costs surrounding the goal. The idea was to "trick" the A* agent into expanding many unnecessary nodes, under the assumption that its heuristic might lead it to explore more nodes than BFS, which is simply based on exploring all possible paths level by level.

However, as we can see from the results, this did not happen. The reason for this lies in the properties of the Manhattan distance heuristic used in A*. The Manhattan heuristic is admissible, meaning that it never overestimates the true cost to the goal. Additionally, it is consistent (or monotonic), satisfying the condition that the heuristic difference between any two nodes is always less than or equal to the cost of the step between them.

Because of these properties, A* with the Manhattan heuristic expands a subset of the nodes expanded by BFS, meaning that A* will always expand the same number of nodes or fewer than BFS. In this specific maze, despite its complexity, A* performed as expected and expanded fewer nodes than BFS, disproving the assumption that A* would visit more nodes in the case of complex dead ends or higher-cost paths.

Therefore, the results align with the theoretical expectations, where A* using an admissible and consistent heuristic like Manhattan distance cannot expand more nodes than BFS, even in a maze with many potential dead ends.

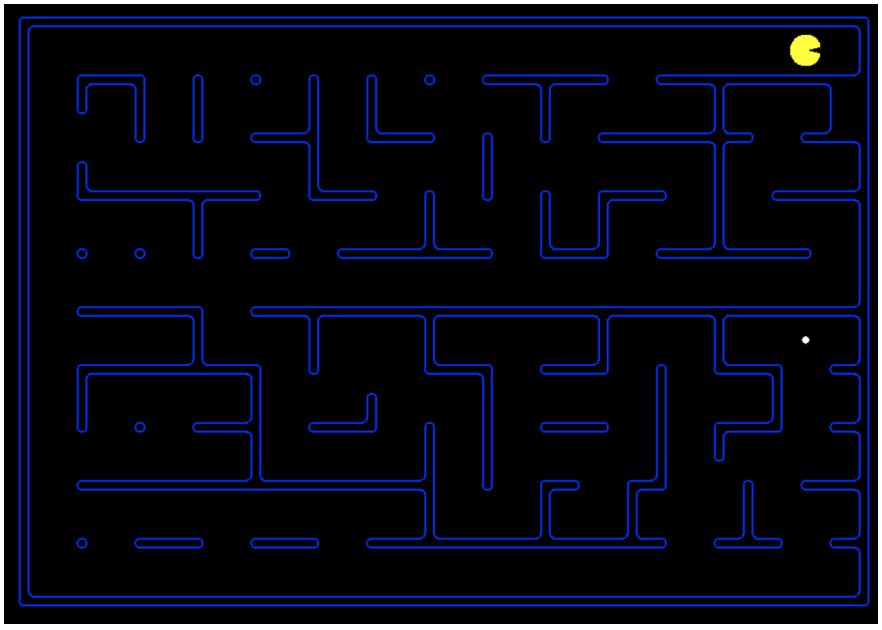


Figure 5: Maze 4

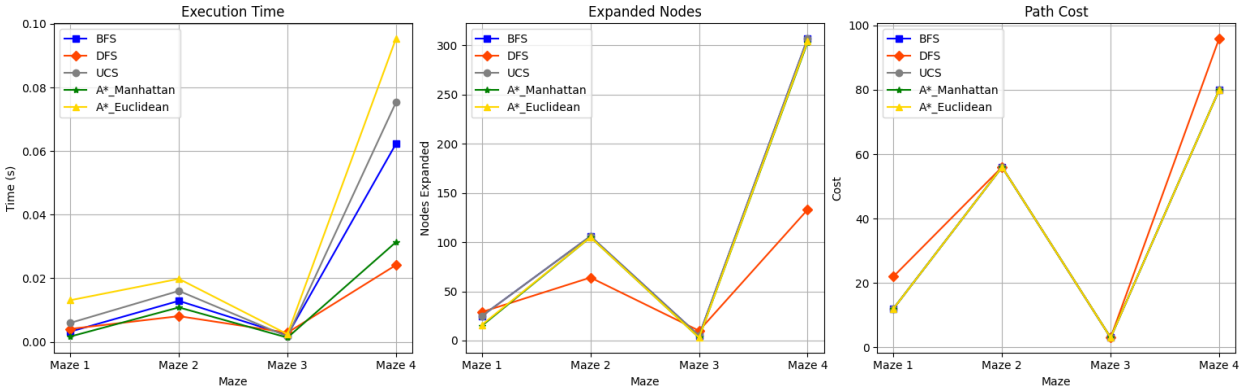
Table 4: Performance Comparison of Search Algorithms on Maze 4

Algorithm	Time (s)	Nodes Expanded	Path Cost
Breadth-First Search (BFS)	0.06232	307	80
Depth-First Search (DFS)	0.02418	133	96
Uniform Cost Search (UCS)	0.07536	307	80
A* Search (Manhattan Heuristic)	0.03140	303	80
A* Search (Euclidean Heuristic)	0.09543	304	80

3.3.5 Overall Observations

- In simpler mazes, **DFS** tends to outperform other algorithms in terms of time, as it quickly dives down one path before backtracking. However, its effectiveness depends on luck, as it may not always find the optimal path.
- **A* with Manhattan** performs very well in terms of node expansions, consistently expanding fewer nodes than BFS and UCS, making it more efficient in larger, more complex mazes.
- **BFS** and **UCS** show similar performances in terms of time and nodes expanded, with BFS always finding the optimal path but at the cost of expanding many unnecessary nodes.
- **A* with Euclidean** performs similarly to **A* with Manhattan**, but in certain cases, it might expand slightly more nodes due to the properties of the Euclidean heuristic.

In conclusion, the choice of search algorithm significantly impacts the efficiency of pathfinding in different environments. While **DFS** might be the fastest in certain cases, it is less reliable for finding the optimal path, especially in complex mazes. **A* search with Manhattan or Euclidean heuristics** generally provides a balanced approach, expanding fewer nodes and guaranteeing optimality. **BFS** and **UCS** ensure the optimal path but at the cost of expanding more nodes and taking longer to compute the solution.



4 Problem 1 Part 2: slight problem modification

4.1 Implementing Diagonal Moves in the Algorithm

In order to implememt diagonal moves in the algorithm I modified these parts of the code:

- Update the `Directions` class with constants for each direction:
- Added diagonal directions (`NORTHEAST`, `SOUTHWEST`, etc.) in relevant parts of the code to allow diagonal movements.

4.2 Comparative Analysis of Search Algorithm A* with different heuristics

In this section, I present an analysis similar to that of Section 3.3, but focusing exclusively on the A* agent using different heuristics.To highlight the strengths of each heuristic, I designed two distinct mazes that demonstrate their respective advantages.

4.2.1 Chebyshev distance

First of all, I implemented a new heuristic called `chebyshevHeuristic`. To do this, I defined a function `chebyshevHeuristic` which returns the result of a helper function called `chebyshevDistance`. This function calculates the Chebyshev distance between two points.

The Chebyshev distance between two points (x_1, y_1) and (x_2, y_2) is given by the following formula:

$$D_{\text{chebyshev}} = \max\left(|x_1 - x_2|,\, |y_1 - y_2|\right)$$

This heuristic is admissible and consistent in environments where movement in all 8 directions is allowed and each move (diagonal or cardinal) has the same cost.

To test it in the Pacman layout, run the following command:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=astar,heuristic=chebyshevHeuristic
```

4.2.2 Maze 5

All the mazes designed for this part are intentionally simple, with the aim of comparing the performance and efficiency of different heuristics. In the first maze, the layout is crafted in such a way that the Euclidean heuristic is not the most efficient, due to obstacles that hinder diagonal movements. The Manhattan distance, being optimal in environments without diagonal paths, results in the fewest node expansions. The Chebyshev heuristic also performs comparably well, while the Euclidean heuristic requires more node expansions to reach the goal. All implementations are extremely fast, with only negligible differences in execution time between them.

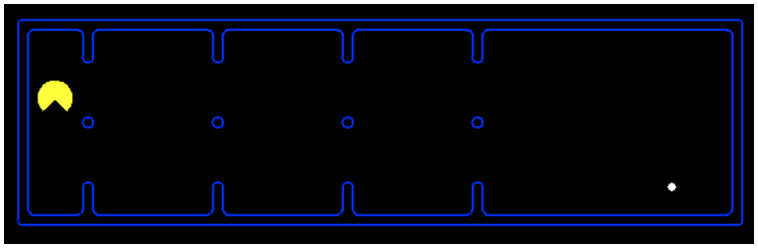


Figure 6: Maze 5

Heuristic	Time (s)	Nodes Expanded	Path Cost
Manhattan	0.00302	19	19
Euclidean	0.00392	26	19
Chebyshev	0.00379	19	19

Table 5: Performance of Heuristics on Maze 5

4.2.3 Maze 6

Although diagonal moves are possible in this maze, the Manhattan and Euclidean heuristics yield similar performance. However, this does not imply that they are always equivalent in general scenarios. Interestingly, in this specific maze, the Chebyshev heuristic ends up visiting nearly twice as many nodes as the Euclidean heuristic. In other words, it is significantly less efficient in this case.

This occurs because, while Chebyshev is admissible under the current cost model (where all moves, including diagonals, have equal cost), it may lead the A* agent to explore broader areas when the optimal

path is not aligned along pure diagonals or straight lines. In contrast, the Euclidean heuristic provides a smoother gradient toward the goal, which helps the search focus more tightly along the optimal path.

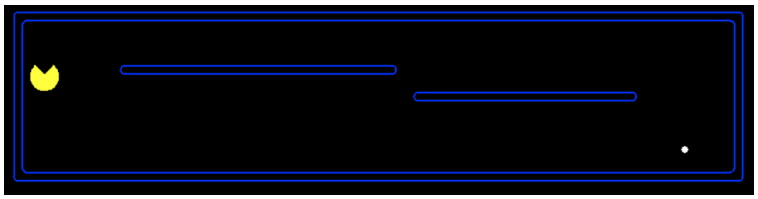


Figure 7: Maze 6

Heuristic	Time (s)	Nodes Expanded	Path Cost
Manhattan	0.00355	24	24
Euclidean	0.00485	24	24
Chebyshev	0.00580	51	24

Table 6: Performance of Heuristics on Maze 6

4.2.4 Maze 7

In this maze, where the cost of moving in any of the eight directions is uniform (e.g., 1), the Chebyshev heuristic performs the best. This is because the Chebyshev distance takes both horizontal/vertical and diagonal moves into account, providing a more accurate estimate of the number of steps required to reach the goal.

On the other hand, the Manhattan heuristic only considers horizontal and vertical movements, which can underestimate the required steps in mazes where diagonal movement would be more efficient. Similarly, the Euclidean heuristic calculates the straight-line distance to the goal but doesn't account for the number of actual steps needed, missing the path efficiency that Chebyshev provides.

These conclusions are supported by the results in the table below, where the Chebyshev heuristic expands the fewest nodes, followed by Manhattan, and finally, Euclidean.

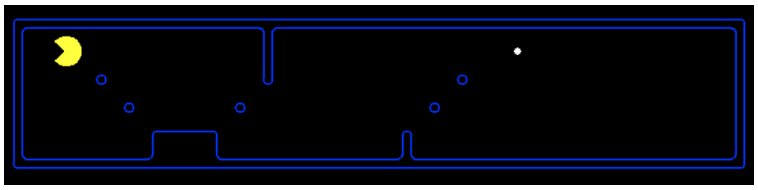


Figure 8: Maze 7

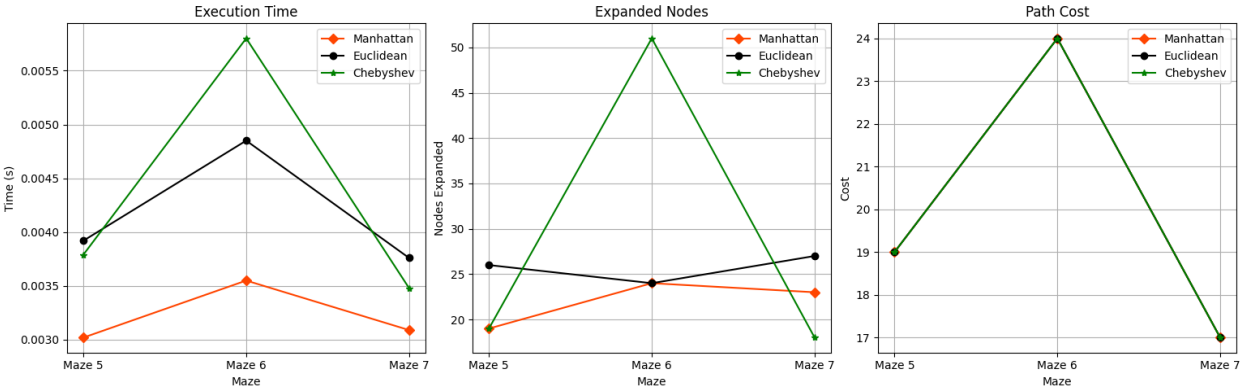
Heuristic	Time (s)	Nodes Expanded	Path Cost
Manhattan	0.00309	23	17
Euclidean	0.00376	27	17
Chebyshev	0.00348	18	17

Table 7: Performance of Heuristics on Maze 7

4.2.5 Overall Observations

Overall, each heuristic has its strengths and weaknesses. The Manhattan heuristic is well-suited for simple grids with only horizontal and vertical moves. The Euclidean heuristic is useful for long-range estimates in open spaces, but may be inefficient in complex mazes with obstacles. The Chebyshev heuristic stands out

as the most balanced, performing well in both diagonal and non-diagonal environments. It strikes a good balance between accuracy and efficiency, making it the best choice for mazes where diagonal movements are allowed and beneficial.



5 Problem 2: Eating all the food dots

5.1 Understanding the Code

The `FoodSearchProblem` class defines a search problem where Pacman must eat all the food dots in a maze. It is structured to be used with generic search algorithms such as A*, BFS, and others. To facilitate this, the class implements several functions that initialize the problem, generate successor states for each possible action, and compute the associated cost of those actions. Below is an explanation of the main components and methods:

- `__init__(self, startingGameState)`: Initializes the problem by storing Pacman's starting position and the food grid. It also stores the walls layout and initializes a counter for the number of expanded nodes. The state space is defined as a tuple containing Pacman's position and the current food grid.
- `getStartState(self)`: Returns the initial state of the problem, which includes Pacman's starting position and the initial configuration of the food grid.
- `isGoalState(self, state)`: Checks whether all food has been consumed. The goal state is reached when the count of remaining food dots is zero, i.e., `state[1].count() == 0`.
- `getSuccessors(self, state)`: Returns a list of all possible successor states from the current state. Each successor includes the new position, the updated food grid (with any food eaten marked as False), the action taken, and the cost of the move. Movements are restricted to legal directions (North, South, East, West) and are blocked by walls.
- `getCostOfActions(self, actions)`: Computes the total cost of a sequence of actions. If any action in the sequence is illegal (e.g., running into a wall), the function returns a very high cost (`999999`). Otherwise, it returns the sum of step costs, typically 1 per move.

This design supports flexibility, allowing various search strategies to be applied to the same problem definition, and enables the use of heuristics for more efficient pathfinding.

5.2 Problem Analysis and State Space Representation

5.2.1 State Space

In the `FoodSearchProblem`, a state is represented by a tuple consisting of:

- Pacman's current position as coordinates (x, y), and

- A `foodGrid` representing the locations of the remaining food.

This is defined in the constructor of the class:

```
self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())
```

The state space is composed of:

1. All valid positions that Pacman can occupy (excluding walls), and
2. All possible configurations of food presence (each food location can either have food or not).

Let:

- S be the number of valid (non-wall) positions Pacman can occupy.
- N be the number of possible food positions.

Then, the size of the state space is given by:

$$\text{State space size} = S \times 2^N$$

This reflects that for each valid position of Pacman, there are 2^N possible combinations of remaining food (present or eaten) across the board.

5.2.2 Initial and Goal States

Initial State: The initial state is defined in the constructor:

```
self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())
```

It includes:

- The starting position of Pacman.
- The full initial configuration of the food grid.

Goal State: A goal state is reached when all food has been collected. This is checked in the method:

```
return state[1].count() == 0
```

That is, the search ends when the total number of remaining food items is zero.

5.2.3 Operators

Operators are the possible actions Pacman can take at any state. These are defined as:

```
Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST
```

In the `getSuccessors` method, each action is validated to ensure it does not lead into a wall:

```
if not self.walls[nextx][nexty]:
    nextFood = state[1].copy()
    nextFood[nextx][nexty] = False
    successors.append((((nextx, nexty), nextFood), direction, 1))
```

Each action results in a new state where:

- Pacman's position is updated based on the action taken.
- The food grid is updated (a food dot is removed if Pacman eats it).
- The step cost is 1 per move.

Thus, the operators move Pacman one step in a valid direction and update the game state accordingly.

5.3 Agent Algorithms and Heuristics

In the `FoodSearchProblem`, three different agents have been implemented, each using a different strategy to collect all the food dots in the maze. These agents are located in the `searchAgents.py` file and are identified as:

- AGENT 1: `AStarFoodSearchAgent_FoodManhattanDistance`
- AGENT 2: `AStarFoodSearchAgent_FoodMazeDistance`
- AGENT 3: `DotSearchAgent`

Each of these agents employs a specific search algorithm and heuristic to guide Pacman toward the goal state. Below is a detailed explanation of how each agent operates.

5.3.1 AGENT 1: astar FoodManhattan

This agent uses the A* search algorithm combined with a heuristic based on the Manhattan distance to the farthest food dot.

- **Algorithm:** A* Search
- **Heuristic:** `foodHeuristicManhattan`

The heuristic function calculates the Manhattan distance from Pacman's current position to each food dot and returns the maximum value.

5.3.2 AGENT 2: astar FoodMazeDistance

This agent also uses the A* search algorithm but employs a more informed heuristic based on the actual maze distance between Pacman and the food dots.

- **Algorithm:** A* Search
- **Heuristic:** `foodHeuristicMazeDistance`

The maze distance is computed by performing a breadth-first search (BFS) from Pacman's position to each food dot and returning the maximum distance. This heuristic is more accurate because it accounts for walls and the true shortest path, not just straight-line distance.

5.3.3 AGENT 3: DotSearch sequences

This agent does not use a single global search. Instead, it repeatedly searches for the path to the closest food dot using a greedy strategy.

- **Algorithm:** Repeated A* or BFS to nearest food dot
- **Heuristic:** None (uses a separate problem: `AnyFoodSearchProblem`)

The agent performs the following loop:

Algorithm 1 DotSearchAgent Strategy

```
Initialize actions as an empty list
Set currentState to the initial game state
while currentState still has food do
    Find path to closest food dot using BFS
    Append the path to actions
    for all action in path do
        Execute action to generate new currentState
    end for
end while
Return actions
```

5.4 Comparison of the behavior of agents

The three agents analyzed above were applied to various search mazes in order to compare their efficiency under different conditions. The results will be discussed individually for each maze, followed by some overall observations and conclusions at the end.

5.4.1 Tricky Search Maze

In this tricky maze, the results reveal distinct trade-offs among the three agents. The A* agent with Maze Distance heuristic achieved the optimal path cost while expanding the fewest number of nodes. However, this came at the cost of a very high execution time, highlighting the computational complexity of this agent due to the repeated use of Breadth-First Search to calculate maze distances.

The A* agent with Manhattan Distance heuristic also found an optimal path in terms of cost, but it expanded nearly three times as many nodes as the Maze Distance agent. This indicates that although Manhattan Distance is faster to compute, it is less informed about the actual structure of the maze, leading to a less efficient search.

Lastly, the DotSearch Agent, despite lacking a sophisticated heuristic, performed remarkably well in terms of speed and node expansion. It expanded a minimal number of nodes and had the shortest execution time by far. However, this came with a slight increase in the path cost, suggesting a more greedy and less globally optimal strategy.

These results raise an important consideration: depending on the context, the "best" agent may vary. If optimal path quality is essential and time is not a constraint, the Maze Distance agent is preferable. If speed and resource efficiency are more critical, DotSearch may be more practical for this maze. Therefore, the choice of agent depends heavily on the trade-off between computational cost and path optimality.

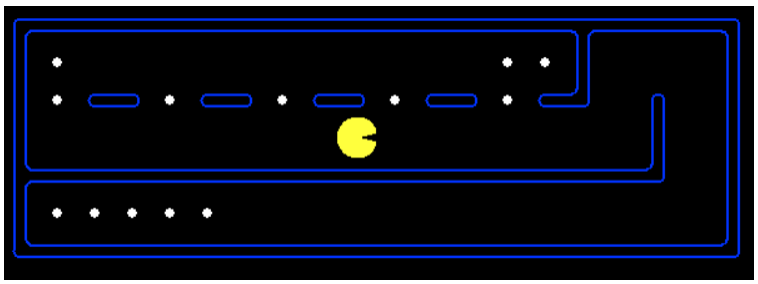


Figure 9: Tricky Search Maze

Agent	Time (s)	Nodes Expanded	Path Cost
A* with Manhattan Distance	17.29210	9402	60
A* with Maze Distance	122.54883	3712	60
DotSearch (Greedy Closest Dot)	0.01111	132	68

Table 8: Agent Performance on `trickySearch` Layout

5.4.2 Tiny Search Maze

In the `tinySearch` layout, all agents successfully completed the task, but with significant differences in performance.

The two A* agents found the optimal path with a cost of 27. However, the A* agent using the Maze Distance heuristic required substantially more time to compute, despite expanding only slightly more nodes than the Manhattan-based version. This reflects the high computational overhead of calculating accurate maze distances at each step.

On the other hand, the DotSearch agent, while producing a suboptimal path (cost = 31), executed extremely quickly and expanded the fewest nodes by far. Its greedy approach, focusing only on the nearest food dot, results in very efficient performance for small-scale problems like this.

This comparison clearly demonstrates the trade-off between optimality and efficiency: while A* with an informed heuristic ensures optimal paths, simpler agents like DotSearch can offer practical advantages in small or time-sensitive scenarios.

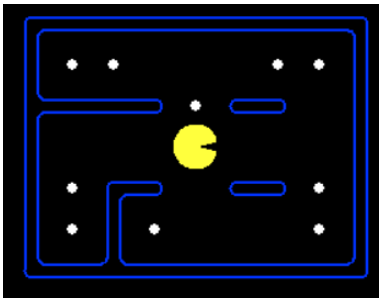


Figure 10: Tiny Search Maze

Agent	Time (s)	Nodes Expanded	Path Cost
A* with Manhattan Distance	0.48367	1812	27
A* with Maze Distance	17.27893	1932	27
DotSearch (Greedy Closest Dot)	0.00314	73	31

Table 9: Agent Performance on `tinySearch` Layout

5.4.3 Small Search Maze

The results on the `SmallSearch` maze are consistent with trends observed in previous scenarios. Both A* agents found the optimal path, with the Manhattan heuristic again expanding more nodes than the Maze Distance version, though the latter required considerably more time to compute.

Notably, DotSearch maintained its efficiency in terms of time and node expansion, but at the cost of a significantly longer path (cost = 48), which is the highest deviation from the optimal among all tested mazes so far. This highlights a limitation of its greedy approach in mazes where food dots are more widely spread.

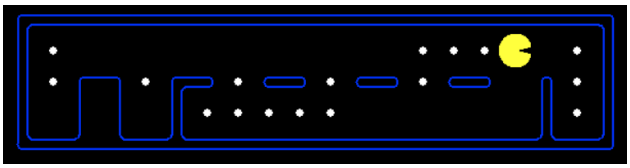


Figure 11: Small Search Maze

Agent	Time (s)	Nodes Expanded	Path Cost
A* with Manhattan Distance	6.25413	5611	34
A* with Maze Distance	98.71100	4175	34
DotSearch (Greedy Closest Dot)	0.00388	105	48

Table 10: Agent Performance on `SmallSearch` Layout

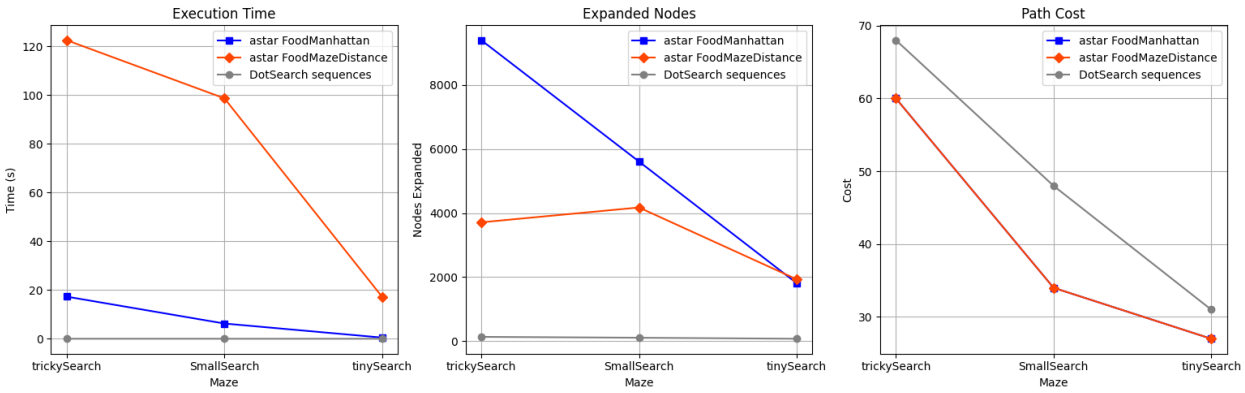
5.4.4 Big Search Maze and Mazes with Many Dots

As observed earlier, the first two agents—A* with Manhattan Distance and A* with Maze Distance—are significantly more computationally intensive. When I attempted to run them on the `BigSearch` maze, the execution time exceeded half an hour, and I had to terminate the process manually.

This experience highlighted a key limitation: mazes with a large number of food dots greatly increase the problem's complexity, making it difficult for these agents to complete execution within a reasonable timeframe. While they are capable of finding optimal paths, their practical usefulness becomes questionable on machines with limited processing power. In such scenarios, their high computational cost outweighs the benefits of optimality.

5.4.5 Overall Observations

As the graphs indicate, smaller mazes with fewer food dots result in smaller differences in execution time across the agents. Additionally, the variations in path cost are minimal. Both the Manhattan and Maze Distance heuristics consistently find the optimal path. However, they do so at the cost of a large number of node expansions and high execution time—an overhead that becomes unmanageable in larger mazes.



6 Conclusions

This project explored the implementation and evaluation of various search algorithms and heuristics within the context of the Pacman maze environment. It was divided into two main parts: the first focused on classical search strategies (DFS, BFS, UCS, and A*), and the second on comparing specialized agents designed to collect all food dots in a maze.

From the experiments conducted in Part 1, it became clear that each search algorithm has its own strengths and weaknesses, which are influenced by the structure and complexity of the maze. Depth-First Search (DFS), while often fast, lacks optimality and consistency. Breadth-First Search (BFS) and Uniform Cost Search (UCS) are both reliable in finding optimal paths but can be computationally expensive due to the high number of expanded nodes. A* Search, when paired with an appropriate heuristic, consistently delivered strong performance, balancing optimality and efficiency.

In Part 2, by enabling diagonal movement, the problem was slightly modified to reflect a more flexible navigation space. Through the design of custom mazes, the effectiveness of different heuristics—Manhattan, Euclidean, and Chebyshev—was thoroughly examined. The results showed that Chebyshev distance performs best in scenarios where diagonal movement is allowed and equally weighted. Manhattan distance proved efficient in structured, grid-based layouts without significant diagonal influence, while Euclidean distance offered an intuitive yet sometimes less efficient estimate due to its continuous-space nature.

In the final section, the three food-collecting agents demonstrated distinct trade-offs between optimality, efficiency, and computational cost. The A* agent with Maze Distance consistently found optimal paths with relatively low node expansions but incurred significant execution time. The Manhattan-based A* agent maintained optimality while being faster, though it expanded substantially more nodes. The DotSearchAgent, while not optimal in path cost, delivered exceptional performance in terms of speed and node efficiency, making it a practical choice for simpler or time-sensitive scenarios. Ultimately, the best agent depends on the specific priorities of the task—whether optimality, speed, or computational resources are the driving concern.

Key Takeaways:

- A* Search remains the most robust and flexible search method, particularly when equipped with a well-chosen heuristic.
- Heuristics must be aligned with the movement rules and maze structure to be effective.
- The trade-off between computational efficiency and path optimality is a central factor when selecting a search strategy or agent.
- Simple agents can outperform more complex ones in specific scenarios, especially when computation time is a critical constraint.

In conclusion, intelligent selection of search strategies and heuristics is crucial for solving pathfinding problems effectively. The Pacman framework provides a rich environment for experimenting with these concepts and gaining insights into the design and evaluation of AI-based agents.

7 Submission Files

The final submission for this project includes the following components:

- This final report.
- A folder containing the updated code files `game.py` and `searchAgents.py` for Problem 1, Part 2. All modifications made to enable diagonal movement are implemented exclusively in these two files within the overall Pacman game framework.
- A folder named `layout_57506`, which contains all the custom-designed mazes. Each file (e.g., `layout_try_1.lay`) corresponds to a specific maze scenario (e.g. Maze 1) used for testing and evaluation.