VNUHCM - UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

LAB REPORT

COURSE: Introduction to AI

LAB 1: PAC-MAN

Instructor:
Nguyen Ngoc Thao
Nguyen Thanh Tinh
Nguyen Hai Dang
Nguyen Tran Duy Minh

Students:

Tran Huu Khang	23127203	23CLC08
Le Trung Kien	23127075	23CLC08
Nguyen Huu Anh Tri	23127130	23CLC08
Nguyen Van Minh	23127423	23CLC08

Ho Chi Minh City, March 14th 2025

Contents

1 Project Planning and Task Distribution

1.1 Task Distribution

Name - ID	Task	Completion
Nguyen Huu Anh Tri - 23127130	Draw the maps and characters	100%
	Implement the UCS search algorithm	
	Calculate the statistics for experiments	
Le Trung Kien - 23127075	Design the characters (including ghosts, pacman)	100%
	Implement the DFS algorithm	
	Draw the measurements and GameOver's screen	
Nguyen Van Minh - 23127423	Using threads to handle parrallel execution	100%
	Implement the BFS algorithm	
	Collect data for experiments	
Tran Huu Khang - 23127203	Record the video demo	100%
	Implement the A* search algorithm	
	Draw the graph for the experiments	

1.2 Project planning

1.2.1 Programming Environment and Tools Used

In this project, we used Python 3.11.9 as the programming language. The following libraries were utilized:

- `pip`: Package installer for Python.
- `pygame`: Library for creating video games.
- `typing`: Provides type hints.
- `sys`: Provides access to some variables used or maintained by the interpreter.
- `enum`: Support for enumerations.
- `time`: Provides various time-related functions.
- `threading`: Higher-level threading interface.
- `heapq`: Provides an implementation of the heap queue algorithm.
- `csv`: Support for CSV file reading and writing.
- `os`: Provides a way of using operating system dependent functionality.
- `collections`: Provides specialized container datatypes.

1.2.2 Code Structure and Key Functions

The code is structured into several classes and functions to handle different aspects of the game:

- **GameManager**: Manages the overall game, including initialization, starting, stopping, and drawing the game elements.
- **Pacman**: Represents the Pac-Man character and its interactions.
- **Maze**: Represents the game maze, including walls, dots, and gates.
- **Ghost**: Base class for different types of ghosts and their search algorithms (BlueGhost - BFS, OrangeGhost - UCS search, PinkGhost - DFS, RedGhost - A* search).
- **GhostThread**: Handles the parallel execution of ghost movements with their own searching algorithms.
- **ExperimentLogger**: Logs the search performance statistics.
- **Record**: Manages the recording of game states and statistics.
- **GameResult**: Displays the game result at the end of the game.

1.2.3 Handling Parallel Execution of Ghosts

Parallel execution of ghosts is managed using the `threading` module. Each ghost runs in its own thread, allowing them to move independently and simultaneously. This is achieved by creating a `GhostThread` for each ghost, which is started and managed by the `GameManager`. The `running` event is used to control the execution of these threads.

1.2.4 Using Ghost's Images and Pac-Man's Images

Images for the ghosts and Pac-Man are loaded using the `pygame.image.load` function. These images are then rendered on the game screen at their respective positions. The `load_image` method is called for each ghost and Pac-Man during the game start, and the `display` method is used to draw them on the screen based on their current positions.

1.2.5 User-Controlled Pac-Man Interaction

The user controls Pac-Man using keyboard inputs. The `pygame.event.get` function is used to capture key presses, which are then translated into movements for Pac-Man. The `move` method of the `Pacman` class is called to update Pac-Man's position based on the user input. The game checks for collisions with walls and ghosts to determine the outcome of each move.

2 State Space

The state space for each ghost is the set of all possible positions in the maze grid. Each position is represented as a tuple of coordinates (x, y).

2.1 Cost

The cost varies depending on the algorithm:

- **BlueGhost (BFS)**: Uniform cost for each move.
- **PinkGhost (DFS)**: Uniform cost for each move.
- **OrangeGhost (UCS)**: Cost is based on the difference in weights between the current and next positions. In order to make it different from the BFS, we generate a sub-board that marks each runnable position randomly with a value from 1 to 4. And the weight between two values will be counted like as the absolute difference between the values of the next and current positions plus one: $|value(next\_pos) - value(current\_pos)| + 1$
- **RedGhost (A*)**: Cost is based on maze features (e.g., walls, dots) and uses a heuristic (Manhattan distance).

2.2 Heuristics

- **RedGhost (A*)**: Uses the Manhattan distance heuristic to estimate the cost to the target (e.g., $|x_1 - x_2| + |y_1 - y_2|$).

3 Algorithm Descriptions

3.1 Some definitions

- Neighbors: next 4 positions in 4 directions: up, down, left, right
- Neighbors' priority: left, right, up, down.

3.2 Input and output

- Input: Pacman's position
- Output: Path reaching Pacman's position

3.3 Breadth-First Search (BFS) - BlueGhost [3]

Breadth-First Search (BFS) is a systematic graph traversal algorithm used in AI to explore a search space level by level, making it ideal for finding the shortest path in unweighted graphs like a Pac-Man maze. Starting from an initial node (e.g., a ghost's position), BFS uses a queue to visit all neighbors at the current depth before moving deeper, ensuring completeness and optimality for the number of steps. Here is a brief description of the procedure, using a FIFO queue as a frontier:

- Begin at the initial node (e.g., the starting position) and add it to a queue. Mark it as visited.
- For every node popped from the frontier: If it's the goal node, stop and return the path. Otherwise, find all unvisited and valid neighbors of the current node. Add all of them to the queue, mark them as visited, and optionally track their parent for path reconstruction. Early-stopping can also be applied, by terminating the algorithm after figuring out a goal node as a neighbor.

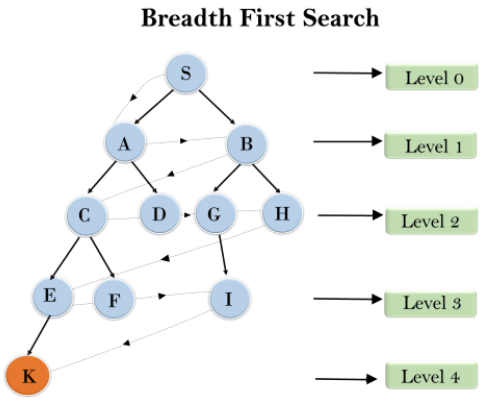

Figure 1: BFS illustration

3.4 Depth-First Search (DFS) - PinkGhost [4]

Depth-First Search (DFS) is a graph traversal algorithm used in AI to explore a search space by diving deep into each branch before backtracking, making it suitable for scenarios where the solution is deep in the search tree. Starting from an initial node (e.g., a ghost's position), DFS uses a stack to explore as far as possible along each branch before backtracking, ensuring completeness but not necessarily optimality. Here is a brief description of the procedure, using a LIFO stack as a frontier:

- Begin at the initial node (e.g., the starting position) and add it to a stack. Mark it as visited.
- For every node popped from the frontier: If it's the goal node, stop and return the path. Otherwise, find all unvisited and valid neighbors of the current node. Add all of them to the stack, mark them as visited, and optionally track their parent for path reconstruction.

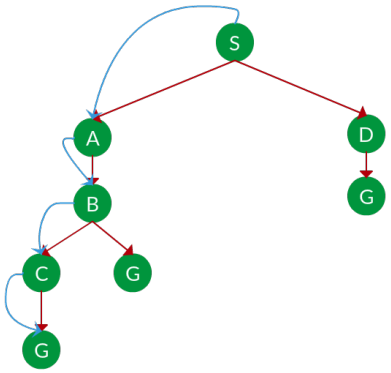

Figure 2: DFS illustration

3.5 Uniform Cost Search (UCS) - OrangeGhost [1]

Uniform Cost Search (UCS) is a graph search algorithm in AI that extends Breadth-First Search (BFS) by considering the cost of paths, making it optimal for weighted graphs where edges have varying costs. Unlike BFS, which assumes uniform step costs, UCS uses a priority queue to explore nodes in ascending order of total path cost from the starting point, ensuring it finds the least-cost path to the goal. While complete and optimal like BFS in finite spaces, UCS has a time complexity of $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$ (where b is the branching factor, C^* is the optimal cost, and ε is the minimum edge cost) and a space complexity of $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$, making it more resource-intensive than BFS in unweighted scenarios but versatile for weighted ones.

- Begin at the initial node (e.g., the starting position), add it to a frontier (e.g., priority queue) with a cost of 0, and mark it as visited with its cost.
- For every node with the lowest cumulative path cost popped from the frontier: If it is the goal node, stop and return the path. Otherwise, find all unvisited neighbors, calculate their cumulative costs (current node cost + edge cost to neighbor). Add each neighbor to the priority queue with its cost, update if a lower-cost path is found, and track parents for path reconstruction.

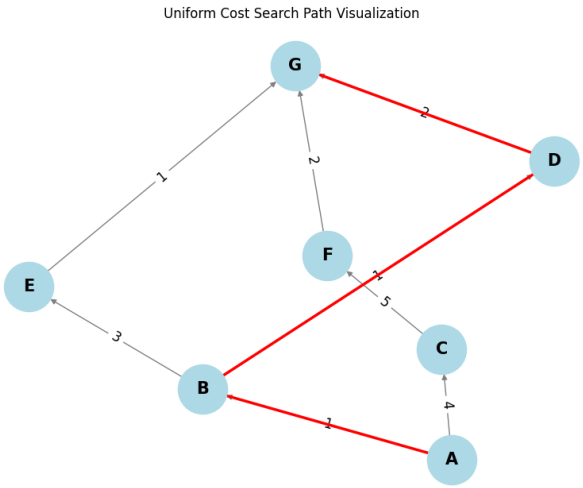

Figure 3: UCS illustration

3.6 A* Search - RedGhost [2]

A* Search is a heuristic-based graph search algorithm in AI that combines the strengths of Uniform Cost Search (UCS) and Greedy Best-First Search to find the least-cost path to the goal efficiently. A* uses a priority queue to explore nodes based on the sum of the path cost from the start and a heuristic estimate of the remaining cost to the goal, ensuring both completeness and optimality. The heuristic function (Manhattan distance) guides the search towards the goal, making A* more efficient than UCS in many scenarios. Here is a brief description:

- Begin at the initial node (e.g., the starting position), add it to a frontier (e.g., priority queue) with its heuristic cost, and mark it as visited with its cost.
- For every node with the lowest estimated total cost (path cost + heuristic) popped from the frontier: If it is the goal node, stop and return the path. Otherwise, find all unvisited neighbors, calculate their tentative scores (current node cost + edge cost to neighbor + heuristic). Add each neighbor to the priority queue with its score, update if a lower-cost path is found, and track parents for path reconstruction.

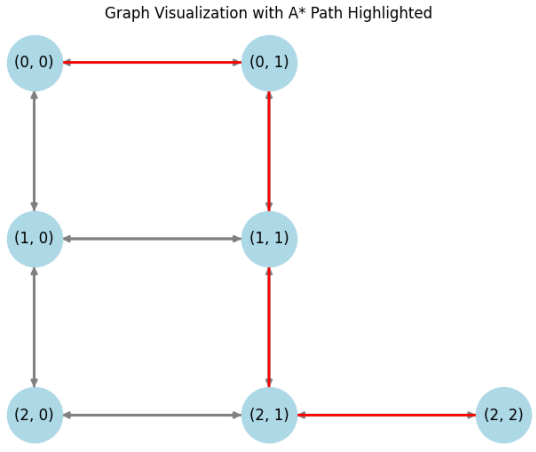

Figure 4: A* illustration with Manhattan distance as a heuristic function

4 Experiments

4.1 Method to count the necessary measurements

In each search algorithm, we will add some variables and functions for these calculations:

- **Expanded Nodes**: The number of nodes that have been expanded during the search process. This is counted by a variable incremented each time a node is popped from the frontier.
- **Search Time**: The total time taken to complete the search. This is calculated as the difference between the end time and the start time using the `time` module.
- **Memory Usage**: The maximum memory usage during the search process. This is measured as sum of the sizes of the containers and the elements they reference using the `sys.getsizeof` function from the `sys` module, and updating the maximum memory usage observed.

4.2 Results

Since the calculations rely on the location of Pacman and Ghost, we focus on some pairs of significant locations, including the four corners of the map and the middle of the maps. When taking experiments on a ghost, we adjust that ghost and the Pacman around the former positions. As the results, each ghosts will be tested with Pacman for 8 pairs of locations (4 of them are the Ghost in the middle and the Pacman at different corners; the other 4 are the opposite situation: The Pacman in the middle and the Ghost at different corners)

In order to compare these search algorithms, we will make 3 comparative tables for each measurement.

4.2.1 Number of expanded nodes

Provided that Pacman is located at position (15,15), the numbers of expanded nodes of the ghosts at the 4 corners of the gameboard are presented in the table below.

Position	Blue Ghost	Pink Ghost	Orange Ghost	Red Ghost
(1,1)	231	343	223	95
(28,1)	220	150	233	29
(1,29)	286	417	289	95
(28,29)	218	149	219	38

Table 1: Number of expanded nodes when Pacman is located at position (15,15)

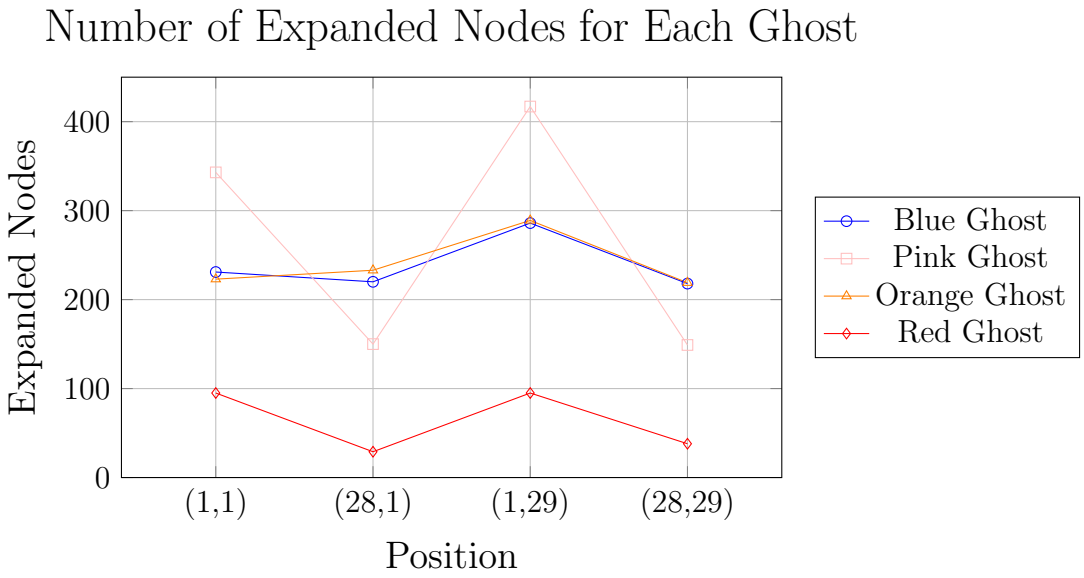

Figure 5: Number of expanded nodes when Pacman is located at position (15,15)

- We can see that BFS (Blue Ghost) and UCS (Orange Ghost) almost have the same number of expanded nodes since their categories to search for the next move is quite simillar to each other.
- For BFS (Blue Ghost) and DFS(Pink Ghost), we can hardly know which one have more or less number of expanded nodes as the DFS is not stable. DFS try its best to find the goal when expanding the path without any category
- Because using a heuristic of Mahattan distance, the A* search (Red Ghost) has more information about the next moves andsufficiently expand the nodes that can get to the goal the fastest. Therefore, it has the least expanded nodes compare to other algorithms.

Provided that the ghosts are located at position (15,15), the number of expanded nodes of the ghosts when Pacman is located at the 4 corners of the gameboard are presented in the table below.

Position	**Blue Ghost**	**Pink Ghost**	**Orange Ghost**	**Red Ghost**
(1,1)	514	297	518	37
(28,1)	524	147	515	79
(1,29)	513	332	503	30
(28,29)	527	93	509	86

Table 2: Number of expanded nodes when the ghosts are located at position (15,15)

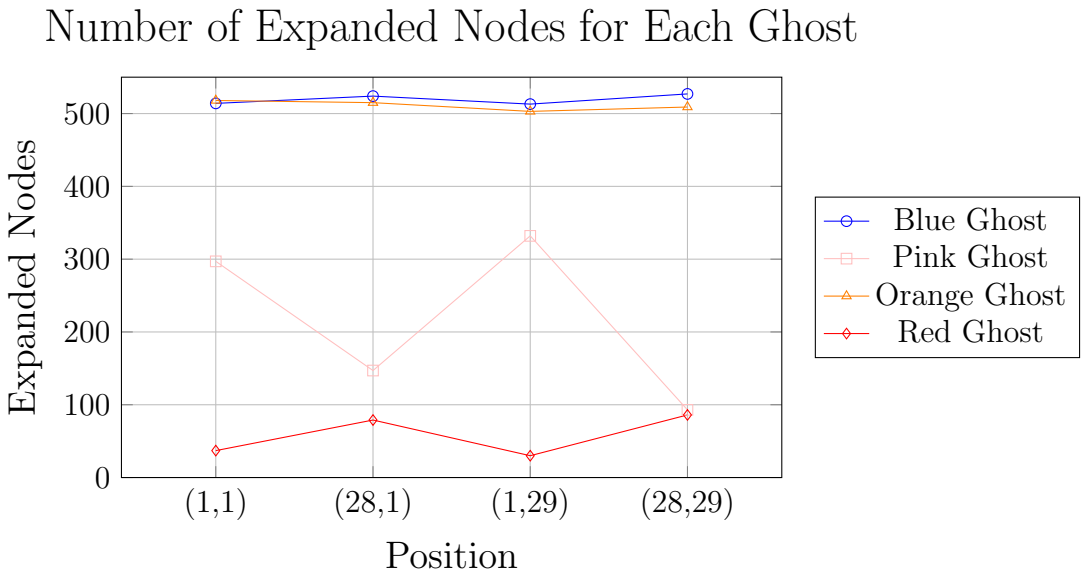

Figure 6: Number of expanded nodes when the ghosts are located at position (15,15)

- Even though changing the location, the number of expanded nodes of BFS(Blue Ghost) and UCS (Orange Ghost) are quite simmilliar (around 500 nodes). Moreover, the A* search still remain its efficiency and its number of expanded nodes is smallest
- The DFS at this situation is better than the BFS. In this situation, the BFS has to expanded many nodes at the same time and reach to almost every positions in the map. Meanwhile, the DFS only have to expand step by step and the map is not so large so it have less expanded nodes than BFS.

4.2.2 Time Search

Provided that Pacman is located at position (15,15), the searching times of the ghosts at the 4 corners of the gameboard are presented in seconds in the table below.

Position	**Blue Ghost**	**Pink Ghost**	**Orange Ghost**	**Red Ghost**
(1,1)	0.016279	0.054013	0.009698	0.01017
(28,1)	0.011961	0.018562	0.012012	0.00103
(1,29)	0.021451	0.084319	0.031853	0.011004
(28,29)	0.013555	0.01259	0.019757	0.001154

Table 3: Searching time (in seconds) when Pacman is located at position (15,15)

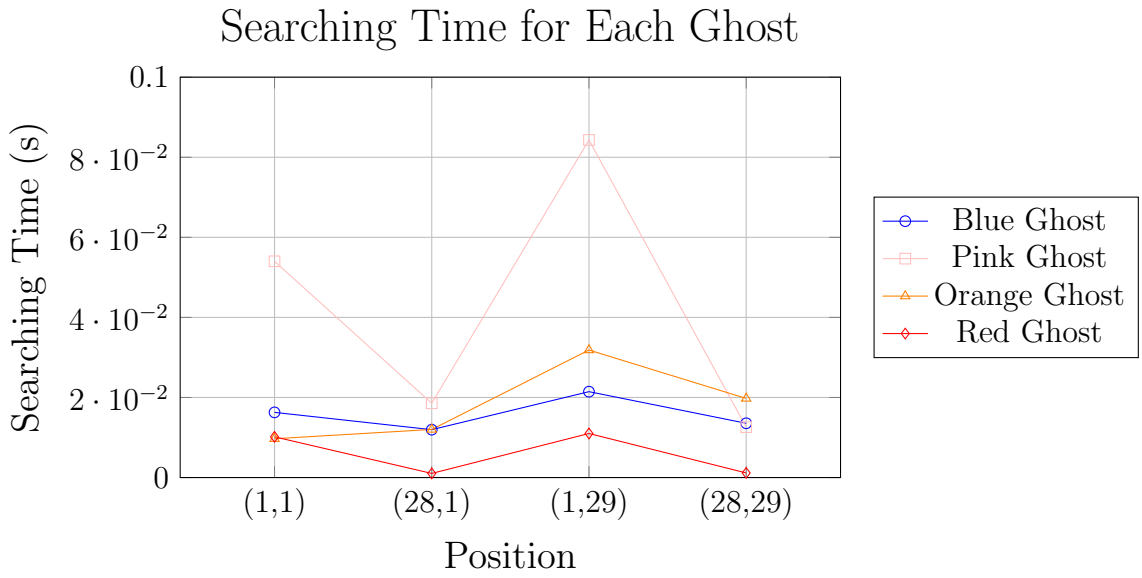

Figure 7: Searching time (in seconds) when Pacman is located at position (15,15)

- Since the A* search (Red Ghost) have the least number of expanded nodes, the algorithm reach to the goal the fastest.
- Even though UCS (Orange Ghost) and BFS (Blue Ghost) have simmilliar stargetry but the UCS sometimes takes more time than BFS as the define weight for the graph is not so efficient. It make the cost for the UCS become bigger than the cost of BFS.
- Among all algorithms, DFS (Pink Ghost) take the longest time to the goal as it expands every possible moves and moves directly without considering any conditions. In this situation, it also have the biggest number of expanded nodes in the first table above.

Provided that the ghosts are located at position (15,15), the searching times of the ghosts when Pacman is located at the 4 corners of the gameboard are presented in seconds in the table below.

Position	**Blue Ghost**	**Pink Ghost**	**Orange Ghost**	**Red Ghost**
(1,1)	0.070312	0.044761	0.072248	0.001383
(28,1)	0.071674	0.015652	0.074116	0.004889
(1,29)	0.088091	0.052062	0.063995	0.00107
(28,29)	0.074444	0.004445	0.076907	0.004846

Table 4: Searching time (in seconds) when the ghosts are located at position (15,15)

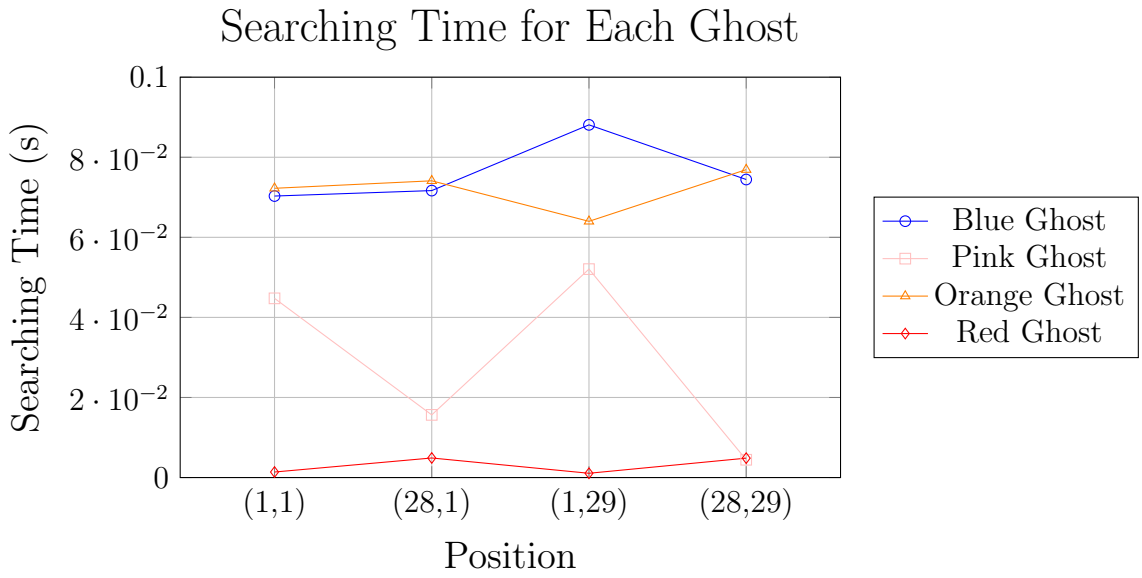

Figure 8: Searching time (in seconds) when the ghosts are located at position (15,15)

- In this situation, the BFS (Blue Ghost) and UCS (Orange Ghost) have more expanded nodes than the DFS (Pink Ghost) as metinoed in the second table. We can the DFS takes less time than these two algorithms even though it have no stragetry to chose the neccessary nodes among the expanded ones.
- The A* search (Red Ghost) still remains the fastest among all.

4.2.3 Memory Usage

Provided that Pacman is located at position (15,15), the memory of the ghosts at the 4 corners of the gameboard are presented in MBs the table below.

Position	**Blue Ghost**	**Pink Ghost**	**Orange Ghost**	**Red Ghost**
(1,1)	0.02608	0.077	0.024056	0.036104
(28,1)	0.024896	0.031352	0.024536	0.016112
(1,29)	0.029488	0.072536	0.028392	0.035176
(28,29)	0.024896	0.032816	0.023872	0.0174

Table 5: Memory usage (in MBs) when Pacman is located at position (15,15)

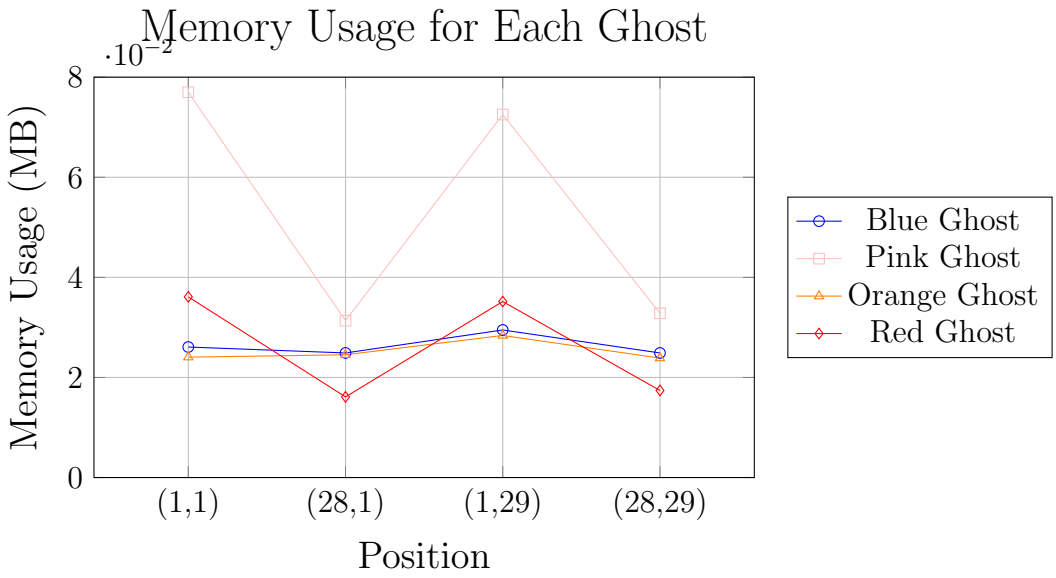

Figure 9: Memory usage (in MBs) when Pacman is located at position (15,15)

- In this situation, the memory cost for DFS (Pink Ghost) is the greatest among all as it has to expand more nodes than the others.
- The other algorithms fluctuate and it is very hard to conclude which one is better and which is not.

Provided that the ghosts are located at position (15,15), the memory usages of the ghosts when Pacman is located at the 4 corners of the gameboard are presented in the table below.

Position	**Blue Ghost**	**Pink Ghost**	**Orange Ghost**	**Red Ghost**
(1,1)	0.065568	0.072912	0.06452	0.01864
(28,1)	0.065568	0.028216	0.06452	0.03364
(1,29)	0.065568	0.072912	0.064024	0.01628
(28,29)	0.065568	0.02656	0.064424	0.034424

Table 6: Memory usage (in MBs) when the ghosts are located at position (15,15)

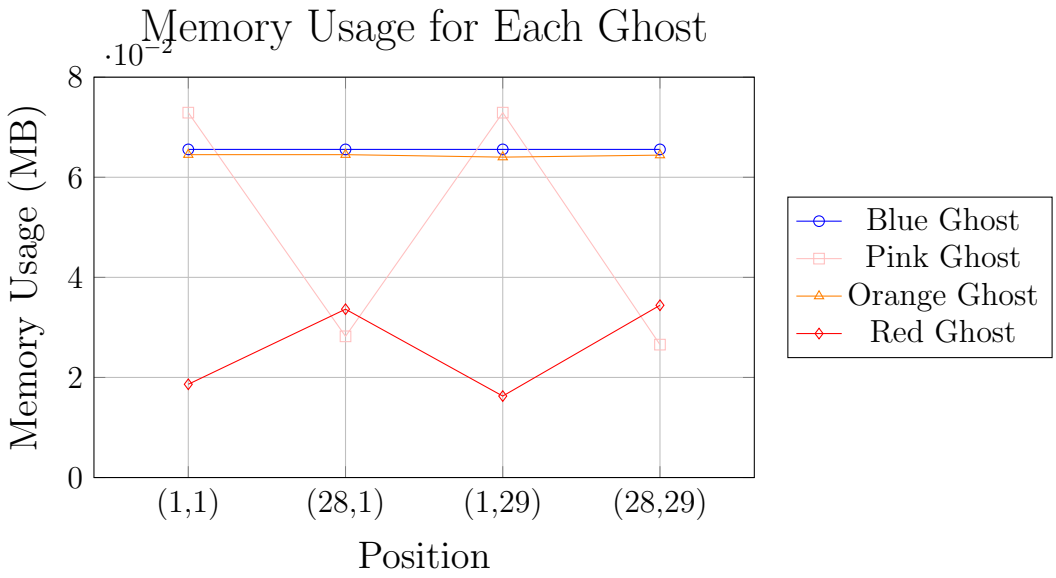

Figure 10: Memory usage (in MB) when the ghosts are located at position (15,15)

- The A* search (Red Ghost) can be considered the most efficient in using memory but it still sometimes use more than DFS (Pink Ghost).
- The memory of BFS (Blue Ghost) and UCS (Orange Ghost) almost the same all the time but the UCS seems to be a little better than BFS.

4.2.4 Summary

The analysis of the number of expanded nodes, search time, and memory usage for the four search algorithms (BFS, DFS, UCS, A*) reveals the following:

- **A* Search (Red Ghost):** Consistently shows the lowest number of expanded nodes, search time, and memory usage, making it the most efficient algorithm. This efficiency is due to the use of heuristics that guide the search towards the goal more effectively.
- **BFS (Blue Ghost):** Expands a moderate to a large number of nodes and has moderate search time and memory usage. BFS explores nodes level by level, which can be less efficient compared to heuristic-based searches.
- **DFS (Pink Ghost):** Shows high variability in the number of expanded nodes, search time, and memory usage. DFS can be efficient if the goal is found early but can also be very inefficient if it explores deep paths that do not lead to the goal.
- **UCS (Orange Ghost):** Similar to BFS in terms of the number of expanded nodes, search time, and memory usage.

5 Source code and video demo

- The source code for this project is available on GitHub at the following link: https://github.com/AnhTtis/Pac-man.git
- The demo video is uploaded on Youtube with the following link: https://youtu.be/2xVyLPzBdh4?si=4Ab21ZGmYxA9KYBf

References

[1] Ariel Felner. "Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm". In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 2. 1. 2011, pp. 47–51.

[2] R Gayathri. "Comparative analysis of various uninformed searching algorithms in AI". In: *IJCSMC* 8.6 (2019), pp. 57–65.

[3] Jonhariono Sihotang. "Analysis of shortest path determination by utilizing breadth first search algorithm". In: *Jurnal Info Sains: Informatika dan Sains* 10.2 (2020), pp. 1–5.

[4] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.