VNU-HCM UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

PROJECT REPORT

PROJECT 1: PACMAN

23CLC10

INSTRUCTOR

NGUYỄN TIẾN HUY

NGUYỄN THANH TÌNH

PHẠM TRẦN DUY MINH

-o0o-

STUDENT

23127049 ĐỖ HOÀNG DUY HƯNG

23127427 VŨ HOÀNG MINH

23127451 NGUYỄN ĐĂNG PHÔN

23127517 NGUYỄN NAM VIỆT

CSC14003 - INTRODUCTION TO AI

HO CHI MINH CITY, 2025

Table of Contents

1 INTRODUCTION

Pac-Man is a legendary arcade game where the player controls Pac-Man to navigate a maze, collect food pellets, and avoid being caught by ghosts. The game's challenge comes from the intelligent behavior of the ghosts, which are programmed to chase Pac-Man using different movement strategies.

In this project, we focus on implementing and analyzing ghost movement using various **search algorithms** to determine the optimal path to reach Pac-Man. Each ghost follows a distinct search strategy:

- **Blue Ghost** → BFS (Breadth-First Search)
- **Pink Ghost** → DFS (Depth-First Search)
- **Orange Ghost** → UCS (Uniform-Cost Search)
- **Red Ghost** → A* (A-Star Search)

After implementation, we evaluate the performance of these algorithms based on:

1. **Search time** – How fast the algorithm finds a path.
2. **Memory usage** – The amount of memory consumed during execution.
3. **Number of expanded nodes** – The number of nodes the algorithm visits before finding the solution.

This project aims to **compare the efficiency of search algorithms in real-time pathfinding scenarios**, providing insights into their advantages and limitations when applied in a dynamic environment like Pac-Man.

2 TEAM MEMBERS CONTRIBUTIONS

Level	Requirement	Member	Percentage
1	Blue Ghost using BFS	Nguyễn Nam Việt	100%
2	Pink Ghost using DFS	Vũ Hoàng Minh	100%
3	Orange Ghost using UCS	Nguyễn Đăng Phôn	100%
4	Red Ghost using A*	Đỗ Hoàng Duy Hưng	100%
5	Parallel Execution	Vũ Hoàng Minh Nguyễn Nam Việt	100%
6	User-Controlled Pac-Man	Nguyễn Đăng Phôn Đỗ Hoàng Duy Hưng	100%
7	Writing Report	Vũ Hoàng Minh Nguyễn Nam Việt Nguyễn Đăng Phôn Đỗ Hoàng Duy Hưng	100%

Table 1: Team Members contributions

Each team member contributed to different aspects of the project. Below is the demo video: **Click here to watch the demo video!**

3 EMPLEMENTATION

3.1 Level 1: Blue Ghost using BFS

3.1.1 Explanation of the Algorithm

- **Function bfs (self, start, goal):**
  - Uses a **Deque** data structure to initialize the first element as a tuple consisting of the starting position coordinates and an empty list representing the path from the start to the current cell.
  - The variable **visited** is used to store the positions that have been traversed to avoid infinite loops.
  - The while loop runs until the **queue** is empty, retrieving elements according to the **FIFO (First In First Out)** principle.
  - After dequeuing an element, the algorithm checks whether the retrieved coordinates match the **goal coordinates**. If they do, the function returns the path list with the **goal coordinates** appended; otherwise, it proceeds to the next step.
  - Next, the function checks if the current coordinates are already in the **visited list**. If the tile is already visited, the loop continues with the next element; if not, the current coordinates are added to **visited**.
  - The algorithm then iterates over the neighboring cells **(up, down, left, right)**, meaning that the priority order for exploring neighbors **is up, down, left,** then right. For each neighbor, it verifies whether the cell is within the map boundaries and has an acceptable value **(1, 2, 9, 10)**. If the neighbor meets these conditions, a new tuple containing that cell and the updated path (with the current tile added) is added to the queue.
  - If no valid path is found after processing all elements in the queue, the function returns an empty list (indicating that no path exists).

3.1.2 Test case BFS:

Test time	Pac-Man position	Search time (sec)	Memory usage (KB)	Number of expanded nodes
1st	Top-left	0.000301	48.00	225
2nd	Top-right	0.000345	64.00	268
3rd	Center of the map	0.000343	100.00	282
4th	Bottom-left	0.0005001	80.00	611
5th	Bottom-right	0.000516	80.00	623

Table 2: Test results for BFS algorithm on Blue Ghost

3.2 Level 2: Pink Ghost using DFS

3.2.1 Explanation of the Algorithm

- **Function dfs (self, start, goal):**
  - Use a **stack** to initialize the first element as a tuple containing the starting coordinates and an empty list representing the path from the start to the current cell.
  - The variable **visited** is used to store the positions that have been traversed to avoid infinite loops.
  - The `while` loop runs until the **stack** is empty, retrieving elements based on the **LIFO (Last In First Out)** principle.
  - After popping an element from the **stack**, we check if the retrieved coordinates match the **goal**. If they do, the function returns the path list appended with the **goal** coordinates; otherwise, it proceeds to the next step.
  - Next, the function checks if the current coordinates are already in the **visited list**. If the tile has been visited, the loop continues with the next element; if not, the current coordinates are added to the `visited` list.
  - It then iterates through the neighboring cells **(up, down, left, right)** with the priority order of **right, left, down, then up**. After that, it checks whether the neighbor is within the map boundaries and has an acceptable value **(1, 2, 9, 10)**. If the neighbor satisfies these conditions, a new tuple containing that cell and the

updated path (with the current cell added) is pushed onto the stack.

– If, after iterating through all the elements in the stack, no valid path is found, the function returns an empty list (indicating that no path was found).

3.2.2 Test case DFS:

Test time	**Pac-Man position**	**Search time (sec)**	**Memory usage (KB)**	**Number of expanded nodes**
1st	Top-left	0.000614	224.00	400
2nd	Top-right	0.000752	184.00	625
3rd	Center of the map	0.000479	152.00	297
4th	Bottom-left	0.000316	128.00	142
5th	Bottom-right	0.000389	32.00	93

Table 3: Test results for DFS algorithm on Pink Ghost

3.3 Level 3: Orange Ghost using UCS

3.3.1 Explanation of the Algorithm

- **Function ucs (self, start, goal):**
  - Uses a **priority queue** (heapq) to always expand the **lowest-cost** path first, ensuring the optimal solution.
  - **Initialize queue:**
    * The queue initially contains a tuple **(cost, position, path)**, where:
      · cost = 0 (starting cost).
      · position = start (starting position).
      · path = [] (initially empty).
  - **Visited set:** A set named `visited` is used to track **explored positions,** preventing redundant calculations and infinite loops.
  - **While loop execution:**
    * **Extract the lowest-cost node** from the priority queue.
    * If the extracted position **matches the goal**, return the computed path.
    * If the position has already been visited, skip it.
    * Otherwise, mark the current position as visited.
  - **Exploring neighboring cells:**
    * The algorithm explores **four possible moves: up, down, left, and right.**
    * Each direction has a predefined movement **cost.**
    * A move is valid if:
      · The new position remains **within the map boundaries.**
      · The destination cell is **not an obstacle.**
  - **Updating the priority queue:**
    * If a move is valid, compute its **total cost** (current cost + move cost).
    * Push the new position, total cost, and updated path into the priority queue.
  - **Handling no valid path:** If the queue is **empty** and the goal hasn't been reached, return an **empty list**, indicating that no path exists.

3.3.2 Test case UCS:

Test time	Pac-Man position	Search time (sec)	Memory usage (KB)	Number of expanded nodes
1st	Top-left	0.000272	44.00	225
2nd	Top-right	0.000312	44.00	273
3rd	Center of the map	0.000327	44.00	291
4th	Bottom-left	0.000632	76.00	611
5th	Bottom-right	0.000647	84.00	625

Table 4: Test results for UCS algorithm on Orange Ghost

3.4 Level 4: Red Ghost using A*

3.4.1 Explanation of the Algorithm

- **Function astar (self, start, goal):**
  - Uses a **priority queue** (heapq) to always expand the node with the **lowest total estimated cost** (f_score = g_score + heuristic). The value of heuristic function is the mahattan distance between the red ghost and pacman. Manhattan distance is :

$$\sum_{i=1}^{k} |x_i - y_i|$$

    - The queue initially contains a tuple **(cost, position)**, where:
      - cost = 0 (starting cost).
      - position = start (starting position).
  - **Tracking structures:**
    - `came_from`: A dictionary storing the previous node for reconstructing the path.
    - `g_score`: A dictionary storing the actual cost from the start node to each explored node.

    - `f_score`: A dictionary storing the estimated total cost (g_score + heuristic).
  - **While loop execution:**
    - **Extract the lowest f_score node** from the priority queue.
    - If the extracted node **matches the goal,** reconstruct and return the path.
    - If the node has already been visited, skip it.
    - Otherwise, process its neighboring cells.
  - **Exploring neighboring cells:**
    - The algorithm explores **four possible moves: up, down, left, and right.**
    - A move is **valid** if:
      - The new position is **within map boundaries.**
      - The destination cell is **not an obstacle.**
  - **Updating the priority queue:**
    - If a move is valid, compute its **tentative g_score** (current g_score + move cost).
    - If this new path is **better** than any previously recorded path, update the tracking structures.
    - Push the new node into the priority queue with the updated f_score.
  - **Handling no valid path:**
    - If the queue is **empty** and the goal hasn't been reached, return an **empty list,** indicating that no valid path exists.

3.4.2 Test case A*

Test time	Pac-Man position	Search time (sec)	Memory usage (KB)	Number of expanded nodes
1st	Top-left	0.000256	4.00	17
2nd	Top-right	0.000191	8.00	31
3rd	Center of the map	0.000380	8.00	31
4th	Bottom-left	0.000226	12.00	57
5th	Bottom-right	0.000234	12.00	61

Table 5: Test results for A* algorithm on Red Ghost

3.5 Level 5: Parallel Execution

3.5.1 Requirement

The implementation ensures that all ghosts (Blue, Pink, Orange, and Red) move simultaneously in the same maze. Each ghost follows its respective search algorithm to chase Pac-Man and executes independently. Additionally, no two ghosts can occupy the same position at the same time.

3.5.2 Ghost Movement and Path-Finding Algorithms

Each ghost utilizes a distinct search algorithm to navigate towards Pac-Man:

- **Blue Ghost**: Uses **BFS** (Breadth-First Search).
- **Pink Ghost**: Uses **DFS** (Depth-First Search).
- **Orange Ghost**: Uses **UCS** (Uniform-Cost Search).
- **Red Ghost**: Uses **A*** (A-Star Algorithm).

3.5.3 Parallel Execution Mechanism

To ensure all ghosts move simultaneously while preventing collisions, the following mechanisms are implemented:

- **Independent Paths**: Each ghost follows a unique path to Pac-Man based on its respective search algorithm. This minimizes the chance of overlapping movements.

- **Distance Monitoring Between Ghosts**: At each movement step, every ghost calculates its distance from the others. If a ghost is about to collide with another, the second ghost is teleported to a different valid tile.
- **Ghost Teleportation**: When teleportation occurs, the ghost must select a valid position. Additionally, its movement-related variables, such as `self.path` and `self.count`, are reset to ensure accurate path recalculations.

3.6 Level 6: User-controlled Pac-Man

3.6.1 Requirements

Enable interactive gameplay by allowing the player to control Pac-Man's movement while the ghosts actively chase him. Ensure real-time updates by making each ghost recalculate its path continuously based on Pac-Man's changing position.

3.6.2 Pac-Man Moves

The movement of Pac-Man in this implementation is controlled by the player using keyboard inputs. The player object is initialized with a random valid starting position on the game board, ensuring that it only spawns in available positions.

Pac-Man's movement is determined by the `move` function within the Player class. This function updates Pac-Man's position based on the current direction of movement. The directions available are:

- Up (Press W)
- Down (Press S)
- Left (Press A)
- Right (Press D)

Each time the player presses a movement key, Pac-Man moves a fixed distance in the chosen direction while avoiding obstacles like walls. The player's position is also updated to ensure smooth transitions between grid spaces.

Additionally, the **draw_player** function updates Pac-Man's animation depending on the direction in which it is moving. Different sprite images are used to create the effect of Pac-Man "opening" and "closing" his mouth as it moves.

3.6.3 How the Game Activates

- Each ghost has a different search algorithm to find a path to Pac-Man:
  - **Blue Ghost**: Uses **BFS** (Breadth-First Search).
  - **Pink Ghost**: Uses **DFS** (Depth-First Search).
  - **Orange Ghost**: Uses **UCS** (Uniform-Cost Search).
  - **Red Ghost**: Uses **A*** (A-Star Algorithm).
- The game will continue until Pac-Man is caught by one of the ghosts.
- The path of each ghost is recalculated continuously as Pac-Man moves or ghosts teleport.

References

[1] GeeksforGeeks (2024). *A* Search Algorithm*. URL: https://www.geeksforgeeks.org/a-search-algorithm/.

[2] GeeksforGeeks (2025). *Breadth First Search or BFS for a Graph*. URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

[3] GeeksforGeeks (2025). *Depth First Search or DFS for a Graph*. URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/.

[4] GeeksforGeeks(2023). *Uniform-Cost Search (Dijkstra for large Graphs)*. URL: https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/.

[5] OpenAi. *ChatGpt*. URL: https://chatgpt.com/.

[6] plesmaster01. *PythonPacman*. URL: https://github.com/plemaster01/PythonPacman.