**VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY**

**INTERNATIONAL UNIVERSITY**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**



**Algorithms & Data Structures**

**IT013IU**

**FINAL REPORT**

**PACMAN**

Instructor:

Dr. Vi Chi Thanh

Msc. Thai Trung Tin

**Members - Student** **ID**

Nguyễn Huỳnh Ngân Anh - ITDSIU23003

Trần Quốc Bảo - ITITWE20033

**TABLE OF CONTENTS**

[**CHAPTER I: INTRODUCTION 3**](#_tffmws72rup3)

[1. Abstract 3](#_nr8acbqajes2)

[2. Objectives 3](#_gp280xegtsfb)

[3. The tools and programming languages 3](#_n0jtjxr1tayo)

[**CHAPTER II: PROJECT ANALYSIS 4**](#_4g0mw3vmu6oq)

[1. Algorithms and Data Structures Application 4](#_dfhqxeorju6y)

[1.1. Data structures: 4](#_2epqh6anx2l9)

[1.2. Algorithms: 5](#_4xaj23r6478z)

[1.2.1. Pathfinding: 5](#_1znvh7704owx)

[1.2.2. Collision detection: 6](#_mjhzueyj3fnf)

[2. Project Structure 7](#_90rzkt4jizg5)

[3. UML and Pseudo Code: 8](#_wulwchktms3u)

[3.1. UML Diagram 8](#_3l4j35cjucqv)

[3.2 Class Description 8](#_ca2qttqt276j)

[3.2.1 Pacman Class 8](#_o31oxzarleyc)

[3.2.2. Ghost Class: 9](#_bo32bpjrccv)

[3.2.3. Game Class: 10](#_6iab97ezojow)

[3.3. Pseudocode 10](#_udy36rb5mgir)

[4. User Interface: 12](#_xb5i62r1sox)

[5. Game implementation: 14](#_sqpzl68vfj9s)

[**CHAPTER III: CONCLUSION 16**](#_oivck7wuas6d)

[1. Accomplishments: 16](#_l2eflv2k96bn)

[2. Future Improvements 16](#_pvpcv1qe87ut)

[3. References 16](#_oj39bfhb286t)

# 

# CHAPTER I: INTRODUCTION

## Abstract

This project aimed to apply the knowledge gained in the Data Structures and Algorithms course to build a Pacman game. The Pacman game application provided users with an engaging and challenging experience while demonstrating the intricate process of ghosts’ movements through sophisticated algorithms, as shown in this report. The use of data structures in the game's development was explained in detail, in addition to the complex algorithm it applied. The report concludes by evaluating the end application to analyze the performance of solving algorithms and propose some future modifications.

## Objectives

Via this project, our group aimed to:

* Incorporate appropriate data structures into the development of the game.
* Construct a correct algorithm to find ghosts’ paths and try optimizing them.
* Develop various features for the game.

## The tools and programming languages

* Javascript: Main programming language for developing the Pacman application.
* HTML: Structure the web page, including the canvas element where the game is rendered and image elements for sprite sheets.
* CSS: Styles the game elements, ensuring a visually appealing presentation.

1. **Project’s contribution**

GitHub repository: https://github.com/nhnain/DSA\_Pacman.git

Contribution table:

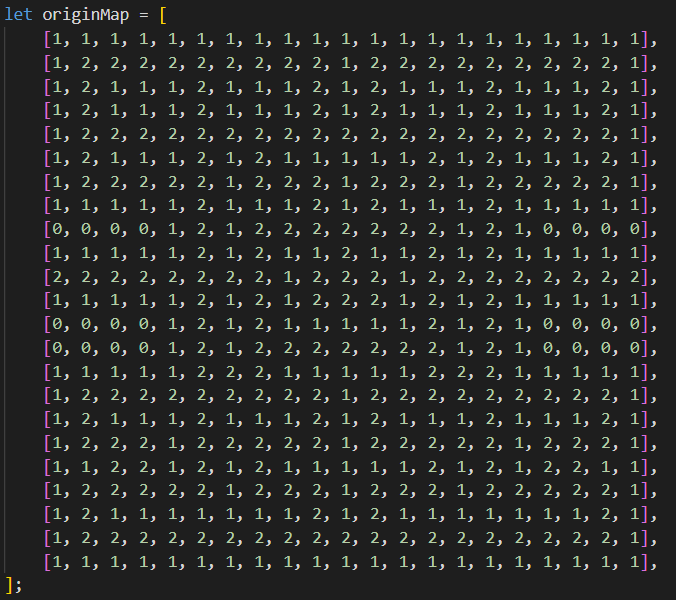
| **Name** | **Student ID** | **Contribution** |
| --- | --- | --- |
| Nguyen Huynh Ngan Anh | ITDSIU23003 | 50% |
| Tran Quoc Bao | ITITWE20033 | 50% |

# CHAPTER II: PROJECT ANALYSIS

## [Algorithms and Data Structures Application](#_epot4ma5qmb)

### 1.1. Data structures:

* 2-dimensional array: The game map is represented as a 2D array (originMap), where each element indicates whether a cell is a wall, empty space, or contains food.



* Queue: the calculateNewDirection method in the Ghost class uses a queue as part of the Breadth-First Search (BFS) algorithm to find the shortest path from the ghost's current position to a destination position on the game map
* Set: Used in the calculateNewDirection method to keep track of visited cells during the BFS algorithm.



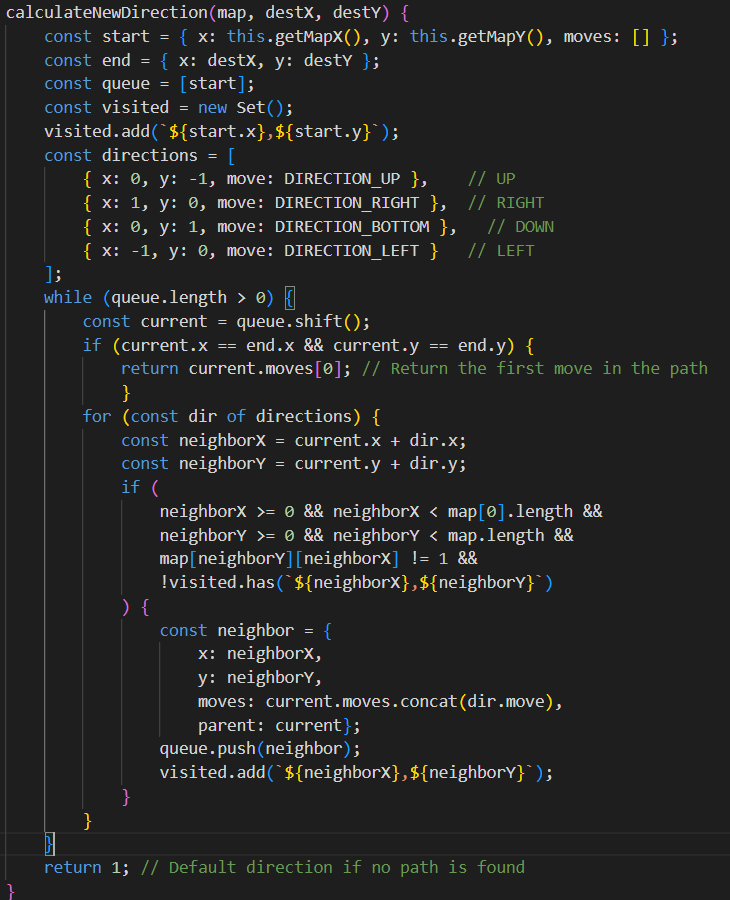


### 1.2. Algorithms:

#### 1.2.1. Pathfinding:

The BFS algorithm used in the calculateNewDirection method efficiently finds the shortest path from the ghost's current position to the destination position on the game map. The algorithm uses a queue to explore the map level by level, ensuring that the shortest path is found. The visited set helps avoid processing the same cell multiple times, and the directions array defines the possible moves.

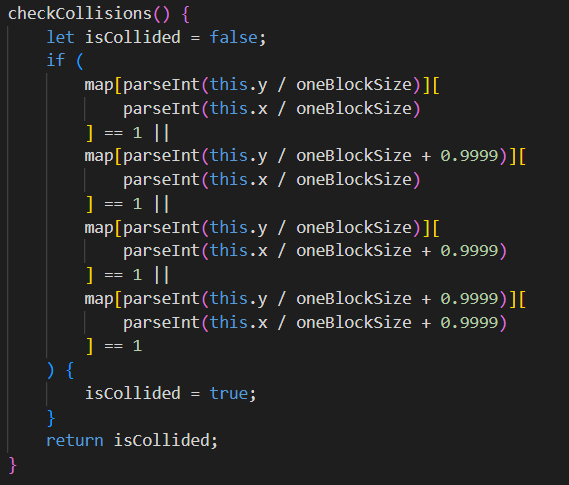
BFS is specifically used in this case as all nodes have the same value, simulating an unweighted graph. Therefore, if we change to other pathfinding algorithms like A\* or Dijkstra, the performance will be the same.



* The worst-case time complexity is O(m\*n), where m and n are the number of rows and columns, respectively. This case happens when the BFS visits every cell and checks its neighbors (up, right, down, left) in the game map.
* Space complexity is O(m\*n) due to queue and set can hold up to m\*n cells in the worst case.

#### 1.2.2. Collision detection:

Both the Pacman and Ghost classes in pacman.js and ghost.js respectively use collision detection algorithms to check for collisions with walls and other objects. This is implemented in the checkCollisions method, which has a time complexity is O(1) as it performs a constant number of operations regardless of the size of the map and space complexity is O(1) since it uses a fixed amount of additional space for variables isCollided.



The checkCollisions method needs to add 0.9999 to make sure the ghosts when moving into the place which is near the boundary between two tiles, prevent them from colliding with the walls and help them to move smoothly and precisely.  
The method will be incorrect if we add 1 instead of 0.9999, due to the fact that the logic of checkCollisions will always consider the next tile, but not the tile that ghosts collide with. This leads to the ghosts may cross the boundary, although still in the collided tile.

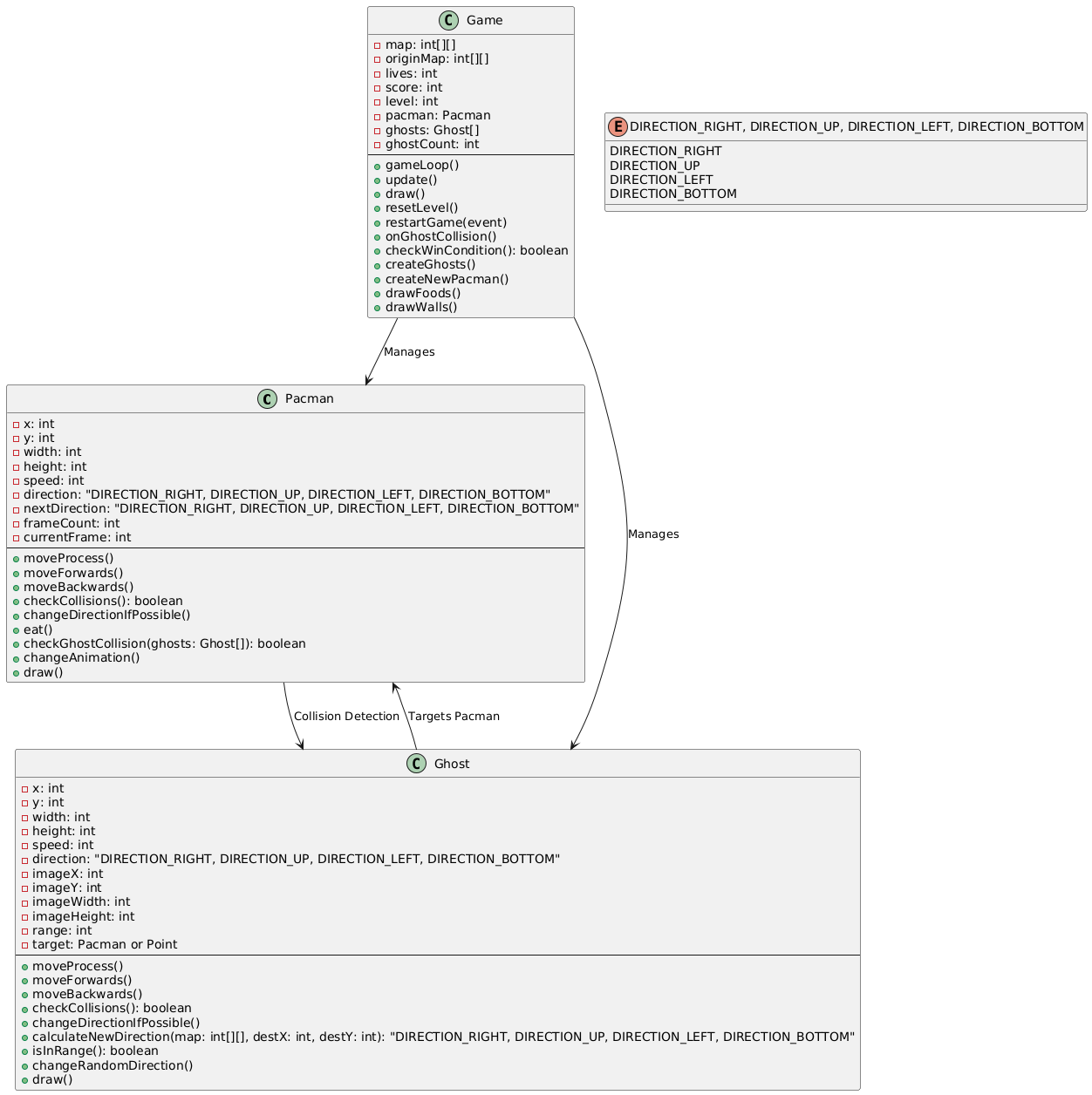
The overall time complexity of the game is O(g\*m\*n), where g, m, and n are the number of ghosts, rows, and columns, respectively as each ghost, in the worst case, will have a time complexity of O(m\*n) in the operation of finding path while space complexity is O(m\*n+g) due to storing the state of each ghost requires additional space.

## Project Structure

The project is organized into directories for core logic and assets. Key files include *game.js* for the main game loop and overall game management, *pacman.js* for defining the Pacman class and its behaviors, *ghost.js* for the Ghost class with pathfinding logic. The *assets* include *animations.gif* for Pacman’s animation frames and *ghost.png* for ghost images. The root of the project includes *index.html* to set up the game canvas and load necessary scripts and *style.css* for defining the game’s visual styles. This organization ensures modularity and clarity, making it easy to maintain and extend the project.

## UML and Pseudo Code:

### 3.1. UML Diagram

****

**Figure 1: UML Class Diagram of Pacman Game**

### 3.2 Class Description

#### 3.2.1 Pacman Class

* Purpose:
  + Represents how the player can control the character in game.
  + Manages movements, animation, and interactions with the game map and other entities.
* Attributes:
  + x, y: Position of Pacman on the map.
  + width, height: Dimensions of the Pacman sprite.
  + speed: Movement speed in pixels per frame.
  + direction: Current direction of movement (right, left, up, down).
  + frameCount, currentFrame: Handles sprite animation for Pacman.
* Key methods:
  + moveProcess: Changes direction and moves Pacman, ensuring no collisions occur.
  + eat: Consumes food at Pacman's current position, and increases the score.
  + checkCollisions: Check if Pacman collides with walls or not.
  + checkGhostCollision: Check if Ghosts collide with walls or not.
  + draw: Renders Pacman on the canvas with rotation based on the direction.
  + changeDirectionIfPossible: Changes direction based on the players.

#### 3.2.2. Ghost Class:

* Purpose:
  + Represents the enemy character in the game.
  + Adjusts its directions around the map, and sometimes chases Pacman.
* Attributes:
  + x, y: Their position on the map.
  + width, height: Dimensions of the Ghost sprite.
  + speed: Movement speed.
  + direction: Current direction of movement.
  + target: Current target location (either Pacman or a random point).
  + range: Radius to switch between random movement and chasing Pacman.
* Key methods:
  + moveProcess: Moves the Ghost and determines whether to chase Pacman or move randomly.
  + changeRandomDirection: Periodically and randomly changes the direction.
  + checkCollisions: Check if collides with walls or not.
  + calculateNewDirection: Computes the optimal direction to move towards the target using a breadth-first search.
  + draw: Renders the Ghost and its detection radius.
  + isInRange: Checks if Pacman is within the detection radius.

#### 3.2.3. Game Class:

* Purpose:
  + Handles the overall game logic, including initialization, rendering, updates, and user interactions.
  + Manages the state of the game such as levels, score, lives, and win/lose conditions.
* Key responsibilities:
  + Map initialization: Creates a 21x23 grid-based map with walls, food, and pathways.
  + Game state: Tracks the number of lives, score, and progression through levels.
  + Rendering: Draws the map, food, walls, characters, and game messages.
  + Controls: Handles user input.
  + Game loop: updates game state, checks win/loss conditions, and redraws the screen.
  + Events: Based on the keyboard input to control Pacman and manage the game states.

### 3.3. Pseudocode

Input: map (grid of the game), position of Pacman, positions of Ghosts.

Output: Movement for the Pacman.

* Function Initialize map:
* gameMap <- map
* PacmanPosition <- {startX, startY}
* GhostPositions <- list of {x, y, direction}
* Score <- 0
* Lives <- 3
* GameOver <- False
* Function GameLoop()
* While GameOver == False:

- Update Pacman Position

- Update Ghost Positions

- Check for Collisions

- Check for WinCondition

* Function UpdatePacman()
* Call moveProcess()
* Call eat()
* Function moveProcess():
* If the direction is valid (no walls in nextDirection):

- Move to the nextDirection.

* Else:

- Stay in the current position.

* Function eat():
* For each cell in gameMap:
* If gameMap[x][y] == FOOD and Pacman at (x,y):

- gameMap[x][y] <- empty

- score++

* Function updateGhosts():
* For each Ghost in GhostPositions:

- Call moveProcess().

- call calculateTargetDirection().

* Function calculateTargetDirection():
* If Pacman is within Ghost range:

- Ghost.Target <- PacmanPosition

* Else:

- Ghost.Target <- RandomTargetPosition

* Function CheckCollisions():
* For each Ghost in GhostPositions:
* If Ghost position == Pacman position:

- Lives - -

* If Lives == 0:

- GameOver <- True

* Call RestartPositions()
* Function RestartPositions():
* Reset PacmanPosition to Start
* Reset GhostPositions to their initial states
* Function checkWinCondition():
* For each cell in gameMap:

- if If FOOD exists:

Return False

* gameOver <- True
* Print “YOU WIN!”
* Function drawGame():
* Clear screen
* Draw Walls, Foods, Pacman, and Ghosts
* Draw Score and Lives
* If GameOver:

- Print “GAME OVER!”

## User Interface:

When running the index.html successfully, users are directed to the main gameplay interface. When losing, the interface will display the text ‘GAME OVER’ as below, together with a guide to restart the game again.

| **Figure 2: Main Gameplay Interface** | **Figure 3: Game Over Screen** |
| --- | --- |

If the user wins, there will be a message to inform and automatically move to the next level.

| **Figure 4: Win Message and Next Level** | **Figure 5: Move To The Next Level** |
| --- | --- |

Our team has designed the game to have 3 levels, so after completing all levels, the user wins and there will be a message to inform them as below.

****

**Figure 6: Final Win Screen**

## Game implementation:

**5.1. Object-Oriented Principles:**

* The game uses two classes Pacman, and Ghost to encapsulate related data and methods, which help to easily maintain, upgrade, and understand.
* The class Game is responsible for managing the main game logic and connecting Pacman and Ghost objects.

**5.2. Algorithms:**

* Ghosts use a Breadth-First Search (BFS) algorithm in calculateNewDirection() in the Ghost class to calculate optimal paths toward Pacman while avoiding walls.
* Collision detection:

- Pacman and Ghosts use the checkCollisions method to handle the walls.

- Pacman detects food and ghost collisions via eat() and checkGhostCollision().

**5.3. User interface:**

* Using HTML5 Canvas for rendering graphics
* Walls, food, and characters are drawn using createRect and canvasContext.
* Dynamic UI updates include displaying scores, levels, and lives.
* Win, game over, pass level screens with messages, and restart options enhance the experience of the users.

# CHAPTER III: CONCLUSION

## Accomplishments:

* **Core gameplay**: The project successfully implemented the classic Pacman game features, including Pacman movement, food consumption, and ghost interaction.
* **Algorithmic Implementation**: Smooth Pacman movement, collision detection, and ghost interaction were achieved.
* **Ghost**: The ghosts were programmed with pathfinding logic, allowing dynamic targeting and movement across the map.
* **User Feedback**: Clear win/lose messages, level indicators, and score displays were integrated to enhance the player experience.

## Future Improvements

* **Enhanced Ghost**: Implement advanced ghost movement algorithms for a more challenging experience.
* **Visual Upgrades**: Include enhanced animations and special effects for a better gaming experience.
* **More features**: Power-ups and Bonuses: Introduce power-ups like temporary ghost invincibility, speed boosts, or score multipliers to diversify gameplay.

## References

[1] Servet Gulnaroglu, “Pacman in JS,” GitHub, 2023. [Online]. Available:<https://github.com/servetgulnaroglu/pacman-js>

[2] “HTML Colors,” W3Schools. [Online]. Available:

<https://www.w3schools.com/colors/colors_picker.asp>

[3] A. H. Nhanh, “Not a Flappy Bird,” GitHub, 2023. [Online]. Available: <https://github.com/atinyanhanh/not_a_flappybird.git>