**HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION**

**FACULTY OF INFORMATION TECHNOLOGY**

Logo

Description automatically generated

**FINAL TERM PROJECT**

**Course name: Artificial Intelligence**

**TITLE OF PROJECT**

**Design and implement Snake games by applying AI algorithms**

**Lecturer name**: Assoc. Prof. Hoang Van Dung

**List of members**

|  |  |  |
| --- | --- | --- |
| **Student ID** | **Student name** | **Contribution (%)** |
| 22110010 | Nguyễn Huỳnh Quốc Bảo | % |
| 22110075 | Hà Quốc Tiến | % |
| 22110041 | Bùi Nguyễn An Khang | % |

*Ho Chi Minh City, mm/2022*

# **Acknowledgment**

Our group would like to express our heartfelt gratitude to Mr. Hoang Van Dung for your invaluable guidance and unwavering support throughout the process of completing our final report on Artificial Intelligence (AI). Your dedication to teaching and mentorship has been instrumental in deepening our understanding of fundamental AI concepts and their practical applications. Through his patient and insightful explanations, you not only resolved our queries but also encouraged us to think critically and innovatively.

Under your mentorship, we have significantly enhanced our research skills, logical reasoning, and ability to translate theoretical knowledge into practical projects. This journey has been both challenging and inspiring, culminating in the creation of a complete game—an achievement we could not have realized without his support.

We are fully aware that despite our best efforts, this project may still contain imperfections and areas for improvement. We sincerely welcome and value your feedback, as it will help us refine our work and further develop our skills for future endeavors.

Once again, we extend our deepest gratitude to Mr. Hoang Van Dung for your dedication, professionalism, and passion for education. Your encouragement and expertise have left a lasting impact on us, as well as on many generations of students. We wish you continued health, happiness, and success in his noble career as an educator, inspiring and guiding students on the exciting and ever-evolving journey of learning and discovery.

**Index**

[**Acknowledgment** 1](#_Toc183615017)

[**Abbreviation list** 3](#_Toc183615018)

[**Figure list** 4](#_Toc183615019)

[**Table list** 5](#_Toc183615020)

[**I.** **Project descriptions** 6](#_Toc183615021)

[**1.** **Reason for choosing the topic** 6](#_Toc183615022)

[**2.** **Project Objectives** 6](#_Toc183615023)

[**3.** **Scope and Target Audience** 7](#_Toc183615024)

[**II.** **Task assignment** 8](#_Toc183615025)

[**III.** **Background knowledges** 9](#_Toc183615026)

[**1.** **The programming environment** 9](#_Toc183615027)

[**2.** **Programming support library** 9](#_Toc183615028)

[**3.** **Methods and techniques** 10](#_Toc183615029)

[**3.1.** **Methods** 10](#_Toc183615030)

[**3.2.** **Techniques** 11](#_Toc183615031)

[**3.2.1.** **A\* Search Algorithm** 11](#_Toc183615032)

[**3.2.2.** **UCS Search Algorithm** 15](#_Toc183615033)

[**3.2.3.** **BFS Search Algorithm** 18](#_Toc183615034)

[**3.2.4.** **DFS Search Algorithm** 21](#_Toc183615035)

[**3.2.5.** **Greedy Search Algorithm** 24](#_Toc183615036)

[**IV.** **Design system** 27](#_Toc183615037)

[**1.** **Class building** 27](#_Toc183615038)

[**1.2.** **Fruit class** 27](#_Toc183615039)

[**1.3.** **Snake class** 28](#_Toc183615040)

[**1.4.** **AI class** 32](#_Toc183615041)

[**2.** **Interface building** 33](#_Toc183615042)

[**Conclusions** 37](#_Toc183615043)

[**1.** **Evaluate the results** 37](#_Toc183615044)

[**2.** **Development direction** 37](#_Toc183615045)

[**References** 39](#_Toc183615046)

# **Abbreviation list**

|  |  |
| --- | --- |
| **Abbreviation** | **Name** |
| UCS | Uniform-Cost Search |
| BFS | Breadth first search |
| DFS | Depth First Search |

# **Figure list**

[Figure 1: Python programming environment 9](#_Toc183616989)

[Figure 2: Pygame programming support library 9](#_Toc183616990)

[Figure 3: A\* Algorithm (Cre: datacamp.com) 11](#_Toc183616991)

[Figure 4: A\* Algorithm Cost Function (Cre: datacamp.com) 12](#_Toc183616992)

[Figure 5: A\* Algorithm in Snake Game(Part 1) 13](#_Toc183616993)

[Figure 6: A\* Algorithm in Snake Game (Part 2) 13](#_Toc183616994)

[Figure 7: A\* Algorithm in Snake Game (Part 3) 14](#_Toc183616995)

[Figure 8: UCS Algorithm in Snake Game(Part 1) 16](#_Toc183616996)

[Figure 9: UCS Algorithm in Snake Game(Part 2) 17](#_Toc183616997)

[Figure 10: UCS Algorithm in Snake Game(Part 3) 17](#_Toc183616998)

[Figure 11: BFS Algorithm in Snake Game(Part 1) 19](#_Toc183616999)

[Figure 12: BFS Algorithm in Snake Game(Part 2) 20](#_Toc183617000)

[Figure 13: BFS Algorithm in Snake Game(Part 3) 20](#_Toc183617001)

[Figure 14: DFS Algorithm in Snake Game(Part 1) 22](#_Toc183617002)

[Figure 15: DFS Algorithm in Snake Game(Part 2) 23](#_Toc183617003)

[Figure 16: DFS Algorithm in Snake Game(Part 3) 23](#_Toc183617004)

[Figure 17: Greedy Algorithm in Snake Game(Part 1) 25](#_Toc183617005)

[Figure 18: Greedy Algorithm in Snake Game(Part 2) 26](#_Toc183617006)

[Figure 19: Greedy Algorithm in Snake Game(Part 3) 26](#_Toc183617007)

[Figure 20: Fruit class 27](#_Toc183617008)

[Figure 21: Import snake model (Part 1) 28](#_Toc183617009)

[Figure 22: Import snake model (Part 2) 28](#_Toc183617010)

[Figure 23: Draw snake model (Part 1) 30](#_Toc183617011)

[Figure 24: Draw snake model (Part 2) 30](#_Toc183617012)

[Figure 25: Update head graphics 31](#_Toc183617013)

[Figure 26: Update tail graphics 31](#_Toc183617014)

[Figure 27: Create snake moverment 31](#_Toc183617015)

[Figure 28: Check free position 32](#_Toc183617016)

[Figure 29: Find save move 32](#_Toc183617017)

[Figure 30: Started interface 33](#_Toc183617018)

[Figure 31: Play snake game as a player 35](#_Toc183617019)

[Figure 32: Play snake game as AI 36](#_Toc183617020)

Table 1

# **Table list**

[Table 1: Work plan 8](#_Toc183616953)

[Table 2: Contants Information 34](#_Toc183616954)

[Table 3: Buttons Information 34](#_Toc183616955)

1. **Project descriptions**
2. **Reason for choosing the topic**

The Snake Game is a classic and widely recognized game that is simple to understand yet engaging, making it an ideal choice for our project. Its straightforward mechanics and familiar gameplay allow us to easily visualize and analyze the operational logic behind the game. Moreover, this simplicity serves as an excellent foundation for exploring and experimenting with a wide range of artificial intelligence (AI) algorithms, including pathfinding techniques like A\* and Dijkstra, as well as optimization methods.

Another advantage of the Snake Game is its accessibility. It does not require complex graphics or powerful hardware, making it easy to develop and test. This simplicity ensures that we can directly observe the steps taken by AI, allowing learners to quickly see results and better comprehend how AI processes work within a real-world application. The game provides an intuitive and interactive platform to demonstrate how AI can execute tasks and adapt to changing conditions.

The Snake Game's core mechanics, such as pathfinding and decision-making, make it highly suitable for applying various AI algorithms. Developers can experiment with different approaches to optimize the snake’s performance. For instance, they can focus on minimizing the distance required to reach food, avoiding obstacles, or even preventing the snake from "self-colliding." This provides an opportunity to understand and compare the efficiency of algorithms like BFS, DFS, or other heuristics, making the project both educational and practical.

1. **Project Objectives**

The primary goal of this project is to explore and apply artificial intelligence algorithms to enhance the development of the Snake Game. By doing so, we aim to deepen our understanding of AI operations and gain practical knowledge of how to implement these techniques effectively. The project seeks to create game elements, such as intelligent characters and dynamic scenarios, that can respond thoughtfully to player inputs, making the gameplay more engaging and challenging.

In addition to advancing our technical knowledge, the project aspires to produce a fully functional and playable game. This game will not only provide entertainment but also serve as a demonstration of how AI can be integrated into game design. By showcasing the practical applications of AI in gaming, the project highlights its potential in other areas of technology and software development.

Furthermore, this project will offer participants an opportunity to enhance their creativity and problem-solving skills. Throughout the game development process, we will encounter challenges that require innovative thinking and analytical approaches to resolve, making it a valuable learning experience. By addressing these challenges, we aim to improve our abilities to design and implement solutions effectively..

1. **Scope and Target Audience**

The Snake Game is designed to appeal to a wide range of players across all age groups, from children to adults. Its simple gameplay and user-friendly interface make it accessible to casual gamers and those new to video games, while the incorporation of intelligent AI-driven elements adds depth and challenge for more experienced players.

By keeping the game mechanics intuitive and the requirements minimal, we aim to create a product that is enjoyable for everyone, regardless of their gaming background. This broad appeal ensures that the game has the potential to reach a diverse audience and encourages players of all ages to engage with it, making it a versatile and inclusive project.

1. **Task assignment**

|  |  |  |
| --- | --- | --- |
| **Student’s name** | **Evaluate contribution** | **Taskwork** |
| Nguyễn Huỳnh Quốc Bảo | % | Create Presentation, report of the project Create the overlay of Snake Game (UI/UX) |
| Hà Quốc Tiến | % | Function coding, Create method for path finding, testing the final product |
| Bùi Nguyễn An Khang | % | Function coding, Advance method to create smarter AI for games |

Table 2: Work plan

1. **Background knowledges**
2. **The programming environment**

The programming environment that Snake game uses includes: Python, Pygame,...



Figure 1: Python programming environment

1. **Programming support library**

Using the pygame library to support programming

A cartoon snake with a toy

Description automatically generated

Figure 2: Pygame programming support library

1. **Methods and techniques**
   1. **Methods**
   * **Game State Management:** Managing the game state involves defining elements such as the initial position and length of the snake, the position of the food, and the score. At the start of the game, these parameters are initialized and continuously updated to reflect changes in the game, ensuring a seamless experience for the player.
   * **Snake Control:** The snake moves based on player input, typically through arrow keys or WASD. Each time the snake moves, a new segment is added to its head, while the last segment is removed, creating a smooth movement effect.
   * **Random Food Generation:** Food appears at random positions after the snake eats it, usually generated using a random number function. It is essential to ensure that the food does not overlap with the snake's current position, preventing errors and maintaining the game's challenge.
   * **Collision Handling:** The game ends if the snake collides with the screen boundaries or itself. This requires checking the position of the snake's head against the rest of its body to detect collisions and terminate the game when the collision condition is met.
   * **Length and Score Increase:** When the snake eats food, its length increases, and the score is updated. This can be implemented by adding a new coordinate to the end of the snake’s body and incrementing the player's score, creating a sense of progress in the game.
   * **Difficulty Scaling:** To make the game more engaging, the difficulty can gradually increase by reducing the time interval between the snake's movements or altering the screen area. This challenges players to improve their reflexes and encourages them to push their limits as they reach higher scores.
   1. **Techniques**
      1. **A\* Search Algorithm**
         1. **Definition of A\* Search Algorithm**

The A\* algorithm is a graph search method designed to find a path from a starting node to a target node or any node that satisfies a specified condition. It works by evaluating nodes using a heuristic function, which estimates the cost of the best path passing through each node. Based on this evaluation, nodes are explored in order of their heuristic ranking. This makes A\* a type of best-first search algorithm.

First introduced in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael, the algorithm was initially called the A algorithm. When equipped with a suitable heuristic, it guarantees optimal results, earning it the name A\*.



Figure 3: A\* Algorithm (Cre: datacamp.com)

A\* maintains a set of incomplete solutions, which are paths through the graph starting from the initial node. These paths are stored in a priority queue, where the priority of each path x is determined by the function f(x)=g(x)+h(x).

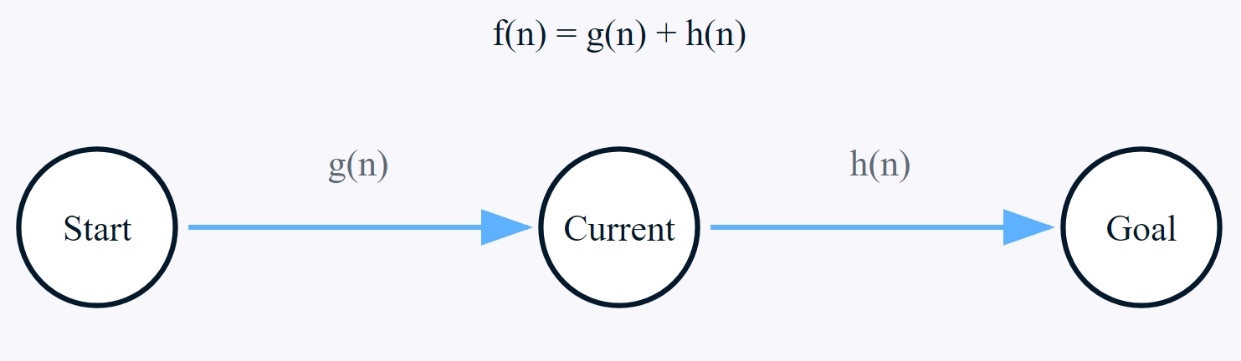


Figure 4: A\* Algorithm Cost Function (Cre: datacamp.com)

**While:**

* **g(x):** The cost of the path so far, calculated as the total weight of the edges traversed.
* **h(x):** A heuristic function estimating the minimum cost required to reach the goal from node x. For example, if "cost" is measured as distance, the straight-line distance between two points on a map can serve as a heuristic estimate for the remaining distance.

The lower the value of f(x), the higher the priority of x. To efficiently implement this priority queue, **a min-heap** structure can be used, ensuring that the path with the lowest f(x) is processed first.

* + - 1. **Algorithmic ideas**

For the Snake Game, A\* uses a heuristic estimate to evaluate which paths are most likely to guide the snake to its target, such as the food. In this case, a common heuristic could be the straight-line distance (Euclidean or Manhattan distance) between the snake's current position and the food, which approximates the What sets A\* apart from other best-first search algorithms is its ability to account for both the distance already traveled (the cost so far) and the estimated remaining distance. This dual consideration makes A\* both "complete" and "optimal" in its search. In the Snake Game, this means that if a path exists for the snake to reach the food, A\* will always find it and will choose the shortest one available.

Although A\* ensures optimality, it does not always run faster than simpler algorithms. In the Snake Game, its performance depends on factors such as the size of the grid, the complexity of obstacles, and the efficiency of the heuristics used.

* **Code:**

**A screenshot of a computer code

Description automatically generated**

Figure 5: A\* Algorithm in Snake Game(Part 1)

*A computer screen shot of a code

Description automatically generated*

Figure 6: A\* Algorithm in Snake Game (Part 2)

***A screen shot of a computer code

Description automatically generated***

Figure 7: A\* Algorithm in Snake Game (Part 3)

* + 1. **UCS Search Algorithm**
       1. **Definition of UCS Search Algorithm**

Uniform Cost Search (UCS) is an algorithm used to find the least-cost path from a start node to a goal node in a weighted graph. Unlike BFS, which explores all nodes at the same depth, UCS expands nodes based on their path cost, ensuring that the path with the lowest accumulated cost is explored first. UCS is essentially a variant of Dijkstra's algorithm and guarantees finding the optimal path in graphs with non-negative edge weights. It systematically expands the node with the least cost, making it particularly useful for problems where the goal is to minimize the cost of the path, such as in network routing or logistics problems.

UCS is based on Dijkstra’s algorithm, developed by Edsger W. Dijkstra in 1956, to find the shortest paths in a graph. It adapts this concept for uniform-cost scenarios, prioritizing paths with the lowest accumulated cost. UCS has become a foundational algorithm in AI and pathfinding due to its efficiency in solving problems in robotics, transportation, and games.

The UCS algorithm works by exploring nodes in increasing order of their path cost from the start node. Here's how it solves problems:

* **Initialization:** Begin by placing the start node into a priority queue, with a path cost of 0. Each node has a path cost associated with it, representing the total cost to reach that node from the start.
* **Exploration:** While the queue is not empty, dequeue the node with the lowest cost. For the current node, examine all its neighboring nodes. For each neighboring node, calculate the new cost to reach that node, which is the current node's cost plus the edge cost to the neighbor.
* **Update and Enqueue:** If the neighboring node has not been visited or if the newly calculated cost is lower than the previously known cost, update the cost and enqueue the node into the priority queue.
* **Goal Check:** The algorithm continues until the goal node is dequeued from the priority queue, at which point the least-cost path to the goal has been found.
  + - 1. **Algorithmic ideas**

In the Snake game, Uniform Cost Search (UCS) finds the least-cost path to the food while avoiding obstacles like walls and the snake's body. Treating the grid as a graph, UCS starts at the snake's head and uses a priority queue to explore paths based on cumulative costs. Each move incurs a fixed cost, and the algorithm evaluates all possible directions while avoiding revisiting obstacles.

UCS guarantees the optimal path to the food by expanding nodes with the lowest cost first. This method is effective even in complex scenarios, such as weighted grids or challenging terrains. By prioritizing efficiency and safety, UCS improves gameplay and enables strategic navigation in the Snake game.

* **Code:**

**A screen shot of a computer code

Description automatically generated**

Figure 8: UCS Algorithm in Snake Game(Part 1)

**A screenshot of a computer code

Description automatically generated**

Figure 9: UCS Algorithm in Snake Game(Part 2)

**A screenshot of a computer code

Description automatically generated**

Figure 10: UCS Algorithm in Snake Game(Part 3)

* + 1. **BFS Search Algorithm**
       1. **Definition of BFS Search Algorithm**

Breadth-First Search (BFS) is an algorithm used for exploring or searching a graph or tree. It starts at a specific node (referred to as the "source" node in graphs or "root" node in trees) and explores all neighboring nodes at the current depth level before moving on to nodes at the next level. BFS ensures that it explores all the nodes level by level, visiting each node in increasing order of their distance from the source node.

The concept of breadth-first search dates back to the early days of graph theory. It was first formally introduced by C. Moore in 1959, and then further developed by other mathematicians and computer scientists as graph traversal became essential in solving computational problems. The BFS algorithm became widely recognized and adopted for solving problems such as finding shortest paths, network routing, and connected components in graphs.

The BFS algorithm is implemented using a queue data structure, where nodes are explored in the order they are visited, starting from the source node. The basic steps of the BFS algorithm are as follows:

* **Initialization:** Start by enqueuing the source node and marking it as visited. A queue is used to manage the exploration process, ensuring that nodes are processed in a first-in, first-out (FIFO) manner.
* **Exploration:** While the queue is not empty, dequeue a node and visit all of its unvisited neighbors. For each unvisited neighbor, mark it as visited and enqueue it.
* **Termination:** The algorithm continues until all reachable nodes have been visited or the destination node is found. In the case of pathfinding, BFS guarantees that the first time it reaches a node, it has found the shortest path to that node.
  + - 1. **Algorithmic ideas**

In the Snake game, BFS (Breadth-First Search) helps the snake find the shortest path to the food while avoiding walls and its own body. The grid is treated as a graph, with each cell as a node, and BFS explores neighboring cells level by level, marking them as visited. It ensures the snake doesn't revisit its body or move into obstacles.

When BFS finds the food, it guarantees the shortest path and reconstructs the route by tracing back from the food to the snake's head. This allows the snake to follow the most efficient and collision-free path, ensuring safe navigation to the food.

* **Code:**

**A computer screen shot of text

Description automatically generated**

Figure 11: BFS Algorithm in Snake Game(Part 1)

**A computer screen shot of text

Description automatically generated**

Figure 12: BFS Algorithm in Snake Game(Part 2)

**A screenshot of a computer code

Description automatically generated**

Figure 13: BFS Algorithm in Snake Game(Part 3)

* + 1. **DFS Search Algorithm**
       1. **Definition of DFS Search Algorithm**

Depth-First Search (DFS) is a graph traversal algorithm that explores as far down a branch as possible before backtracking. It starts at a source node, explores each branch to its deepest point, and then returns to explore other branches. Unlike BFS, which explores level by level, DFS dives deep into one path before moving to the next. It is commonly implemented using either a stack or recursion and is particularly useful for tasks like detecting cycles, solving puzzles, and exploring connected components in graphs.

DFS was first formally defined in the mid-20th century as computer scientists began studying systematic methods for graph traversal. The algorithm is rooted in early explorations of graph theory by mathematicians such as Euler and Kirchhoff.

DFS solves problems by systematically exploring paths in a graph. The steps to implement DFS are as follows:

* **Initialization:** Start at the source node, mark it as visited, and push it onto a stack (or begin recursion).
* **Exploration:** For the current node, explore each unvisited neighboring node. If a neighbor is found, mark it as visited and push it onto the stack. Continue this process for the new node.
* **Backtracking:** When all neighbors of a node are visited, backtrack by popping nodes from the stack until an unvisited neighbor is found or the stack is empty.
* **Goal or Full Traversal:** If a specific goal is defined, stop when it is found. If performing a full traversal, continue until all nodes are visited.
  + - 1. **Algorithmic ideas**

In the Snake game, Depth-First Search (DFS) can be applied to navigate the grid and find a path to the food. The game grid is treated as a graph, where each cell is a node, and adjacent cells are connected by edges. DFS explores one path from the snake’s head as deeply as possible before backtracking to try other paths, making it suitable for finding any valid path, though not necessarily the shortest one.

DFS begins at the snake's head, marking it as visited, and recursively explores adjacent cells (up, down, left, right) that are not occupied by the snake's body or walls. If the food is found, DFS stops, returning the path. If all neighbors are visited or lead to dead ends, the algorithm backtracks to the previous node and continues exploring other directions. By prioritizing depth, DFS can find paths in scenarios where other algorithms might prioritize breadth or cost.

While DFS guarantees finding a path if one exists, it may not be the most efficient choice in the Snake game, as it does not optimize for the shortest or safest route. However, it is useful for exploring possibilities or testing the snake's ability to navigate complex environments.

* **Code:**

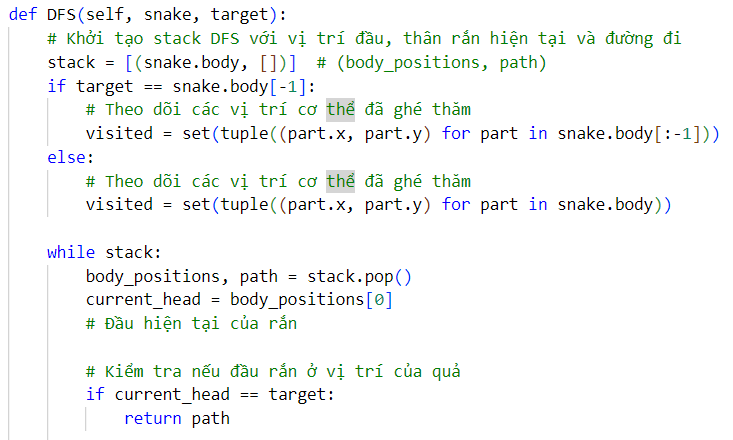
****

Figure 14: DFS Algorithm in Snake Game(Part 1)

*A computer screen shot of text

Description automatically generated*

Figure 15: DFS Algorithm in Snake Game(Part 2)

***A screenshot of a computer code

Description automatically generated***

Figure 16: DFS Algorithm in Snake Game(Part 3)

* + 1. **Greedy Search Algorithm**
       1. **Definition of Greedy Search Algorithm**

Greedy Search is an algorithmic approach used to solve optimization problems by making the locally optimal choice at each step, aiming for a globally optimal solution. The algorithm evaluates available options and selects the one that appears best based on a heuristic or specific criteria. Greedy Search does not backtrack or reconsider decisions, making it simple and fast but not always guaranteed to find the optimal solution.

The Greedy Search algorithm originated in the mid-20th century as a method for solving optimization problems like shortest paths, spanning trees, and resource allocation. Algorithms such as Kruskal’s and Prim’s, based on greedy principles, became fundamental in graph theory. Its simplicity and efficiency made it widely used, especially when heuristics are well-suited to the problem.

Greedy Search solves problems by making a sequence of decisions based on a heuristic, aiming to reduce the problem's size at each step. The process typically involves:

* **Initialization:** Start from an initial state, node, or position.
* **Heuristic Evaluation:** Use a heuristic function to evaluate all available options at the current state. The heuristic is designed to estimate the cost, value, or benefit of each option.
* **Selection:** Choose the option that appears to be the best based on the heuristic. This decision is made without considering future implications or backtracking.
* **Progression:** Move to the chosen option and repeat the process until a goal state is reached or no more options are available.
* **Result:** The solution is the path or sequence of choices made during the process.
  + - 1. **Algorithmic ideas**

In the Snake game, Greedy Search navigates to the food by choosing moves that minimize the immediate distance to the target. The grid is treated as a graph, and the algorithm evaluates potential moves (up, down, left, right) using a heuristic like Manhattan distance, selecting the closest option.

This simple, efficient approach continues until the food is reached or a dead end occurs. While effective in open grids, Greedy Search struggles with obstacles, as it lacks backtracking and ignores long-term consequences. It’s useful for basic AI but may require enhancements in complex environments.

* **Code:**

**A computer screen shot of a code

Description automatically generated**

Figure 17: Greedy Algorithm in Snake Game(Part 1)

**A screenshot of a computer code

Description automatically generated**

Figure 18: Greedy Algorithm in Snake Game(Part 2)

**A screenshot of a computer code

Description automatically generated**

Figure 19: Greedy Algorithm in Snake Game(Part 3)

1. **Design system**
2. **Class building**
   1. **Fruit class**

The **Fruit class** in the Snake game is responsible for managing the fruit's position and appearance. It determines where the fruit will spawn on the grid and provides functionality to display it on the screen. The fruit's position is randomized within the game boundaries, ensuring it appears at a valid location each time. The class supports graphical representation using an image (apple.png) or a simple colored rectangle, making it adaptable for different visual styles. This design integrates seamlessly into the grid-based gameplay and adds variety by positioning the fruit randomly for each new spawn.

**A screen shot of a computer code

Description automatically generated**

Figure 20: Fruit class

* 1. **Snake class**
     1. **Import snake model image**

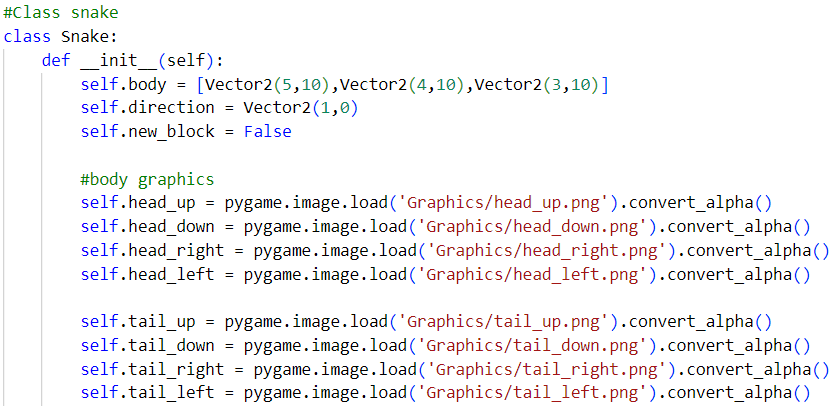
****

Figure 21: Import snake model (Part 1)

**A screen shot of a computer code

Description automatically generated**

Figure 22: Import snake model (Part 2)

* + 1. **Draw snake model**

The draw\_snake method handles the visual representation of the snake in the game, dynamically rendering its head, tail, and body segments based on their positions and orientations. This method ensures that the snake's appearance updates in real-time as it moves and grows.

The method iterates through the snake's body segments, treating the head, body, and tail differently to provide a polished graphical appearance. For the head, it uses a specific graphic that reflects the snake's current direction. The tail is similarly rendered using a distinct image, while body segments are drawn based on their alignment with adjacent segments.

Body segments are categorized as vertical, horizontal, or corner pieces, depending on the relative positions of neighboring segments. This ensures the snake's body is rendered correctly, even when it turns or loops back on itself. By calculating the position of each segment and selecting the appropriate image, the method dynamically adapts to the snake's changing shape and movement, providing a smooth and cohesive visual experience.

**A screenshot of a computer code

Description automatically generated**

Figure 23: Draw snake model (Part 1)

**A computer screen shot of text

Description automatically generated**

Figure 24: Draw snake model (Part 2)

* + 1. **Update head and tail graphics**

**A screen shot of a computer code

Description automatically generated**

Figure 25: Update head graphics

*A screen shot of a computer program

Description automatically generated*

Figure 26: Update tail graphics

* + 1. **Snake moverment**

**A computer code with black and blue text

Description automatically generated**

Figure 27: Create snake moverment

* 1. **AI class**
     1. **Check if the position is free**

**A white background with black text

Description automatically generated**

Figure 28: Check free position

* + 1. **Find safe move**

**A screenshot of a computer code

Description automatically generated**

Figure 29: Find save move

1. **Interface building**

When running this file **main.py** will open a window to choose as a player or AI (using some algorithm)

A screenshot of a video game

Description automatically generated

Figure 30: Started interface

To build the interface as above, our group first declares constants such as matrix size, project title, color code (RGB format),... The constants are declared in the table below:

|  |  |  |
| --- | --- | --- |
| **Contants Information** | | |
| **STT** | **Constants** | **Explain** |
| 1 | CONTROLLER\_WIDTH = 200 | The size of navigation bars |
| 2 | cell\_size = 40 | The size of each cell in map |
| 3 | cell\_number = 19 | Number of cells display in the total map (19x19) |
| 4 | SPEED =150 | The speed for snake moverment |
| 5 | FONT\_NAME = 'Font/Minecraft.ttf' | The file path for the font used in the game |
| 6 | LIGHT\_GREEN = (144,238,144) | Display the color of odd cells of map |
| 7 | LIME\_GREEN = (50,205,50) | Display the color of even cells of map |
| 8 | LIGHT\_GRAY = (211, 211, 211) | Display the color of the background |

Table 3: Contants Information

After having the constants, our group set up the functions used to get the position, determine what the button is and set up each button according to the user's actions. Next, our group proceeded to draw the matrix grid, draw the buttons on the interface for the user. On the main interface, there are 3 function buttons when choose played as a player 8 function buttons when choose played as AI listed in the table below:

|  |  |  |
| --- | --- | --- |
| **BUTTON FUNCTIONS Information** | | |
| **STT** | **Buttons** | **Explain** |
| 1 | RESET | Recreate the position of snake also the fruit |
| 2 | PAUSE | Alternative stop the snake game |
| 3 | BACK | Return to the main page |
| 4 | DFS | Choose DFS Algorithm Solving |
| 5 | BFS | Choose BFS Algorithm Solving |
| 6 | UCS | Choose UCS Algorithm Solving |
| 7 | GREEDY | Choose Greedy Algorithm Solving |
| 8 | A STAR | Choose A\* Algorithm Solving |

Table 4: Buttons Information

* 1. **Player Snake game interface**

**A screenshot of a game

Description automatically generated**

Figure 31: Play snake game as a player

* 1. **AI Snake game interface**

**A screenshot of a game

Description automatically generated**

Figure 32: Play snake game as AI

# **Conclusions**

1. **Evaluate the results**

In terms of algorithms, the BFS, DFS, UCS, Greedy and A\* pathfinding algorithms have been successfully implemented and applied to the Snake game. These algorithms help determine the optimal path for the snake to reach its food while avoiding obstacles. The program visually compares these algorithms, showcasing how they differ in their approach to solving the pathfinding problem in the context of the Snake game.

On the application side, the Snake game has been developed using Pygame to visually demonstrate the functionality of these algorithms. Players can interact with the game and see how each algorithm affects the snake's movement. Additionally, the game allows users to customize the game map, providing flexibility for players to modify the grid and experiment with different scenarios, enhancing both learning and gameplay experience.

1. **Development direction**

Our group's Snake game has the potential to evolve into an interactive learning tool focused on pathfinding algorithms. Players will have the ability to adjust the difficulty level of each game, with the number of obstacles, such as walls or the snake's body, increasing based on the chosen difficulty. In each level, players will manually guide the snake from the starting point to the food. The game will analyze the player's path and compare it to the paths generated by preset algorithms like BFS, DFS, or A\*. The comparison will be based on metrics like time, cost, and distance, helping players understand the performance of different algorithms in real-time.

To achieve this goal, our group has outlined several plans:

* Implement additional pathfinding algorithms to diversify the game, giving players more strategies to explore.
* Add a feature that automatically generates new game maps based on the selected difficulty level, expanding the map's size to challenge players and showcase the strengths and weaknesses of each algorithm in larger environments.
* Introduce new modes, such as step-by-step manual movement or the display of algorithm variables, to help players better understand how the algorithms work.
* Improve the user interface to make the game more engaging and integrate sound effects for various events, making the experience more immersive.
* These enhancements will not only make the Snake game more enjoyable but also serve as an interactive tool for learning and experimenting with pathfinding algorithms in a fun and educational way.

# **References**