MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

Artificial Intelligence (CSE 2225) MINI PROJECT REPORT ON

Classic Snake Game using AI Algorithms

SUBMITTED TO

Department of Computer Science & Engineering

by

Keshav Agrawal	220962436	73
Pranav Kumar Gupta	220962250	37
Ankur Monga	220962137	81
Ishan	220962270	41

Name & Signature of Evaluator

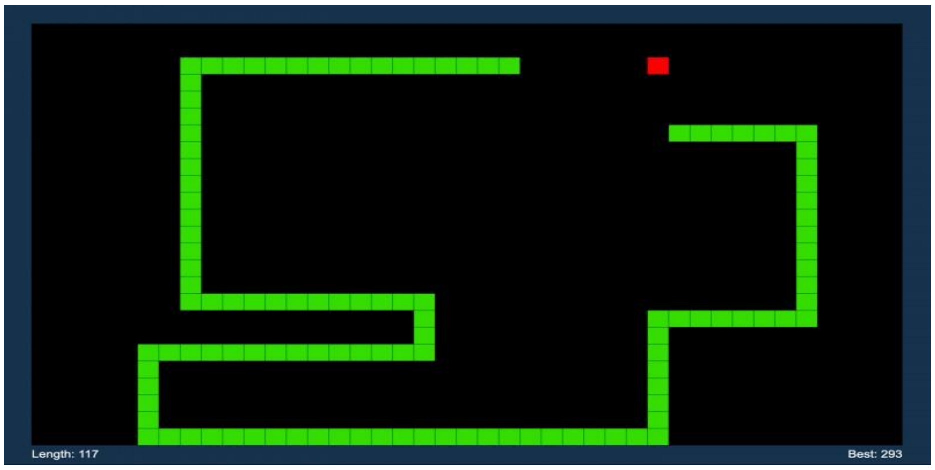
(Jan 2024 - May 2024)

Table of Contents

Chapter 1: Introduction

1. Introduction

The Snake game is an arcade game where a snake moves around a fixed grid, eating food and grows longer after eating each food particle. The player controls the snake's direction to avoid walls and collision with itself. It is a game that was very popular during the early days of the 2000s era, which gained popularity as a mobile game.



2. Problem Statement

- Initial State: The agent (snake) starts in a specific position on the grid and a food particle placed at a random position.

- Possible Actions:
    - Turn Left: Snake head turns left.
    - Turn Right: Snake head turns right.
    - Go Straight: Snake head goes straight.

- Transition Model:

  When the snake moves:
    - If it's moving upwards, its vertical coordinate decreases.
    - If it's moving downwards, its vertical coordinate increases.

    - If it's moving leftwards, its horizontal coordinate decreases.
    - If it's moving rightwards, its horizontal coordinate increases.

  Each movement update occurs based on the snake's current direction, which is controlled by the player's actions (e.g., turning left, turning right, going straight).

- Goal Test: A given state is a goal state if the snake's head occupies the same position as a food pellet, signifying that the snake has eaten the pellet and grown longer.

- Path Cost Function: In the Snake game, the path cost function is defined as the number of steps (actions) taken by the snake to reach the goal state. Each step incurs a cost of 1, so the total path cost is the sum of these individual step costs. When it collides with the wall or its own body, path cost increments by 10.

3. Objective

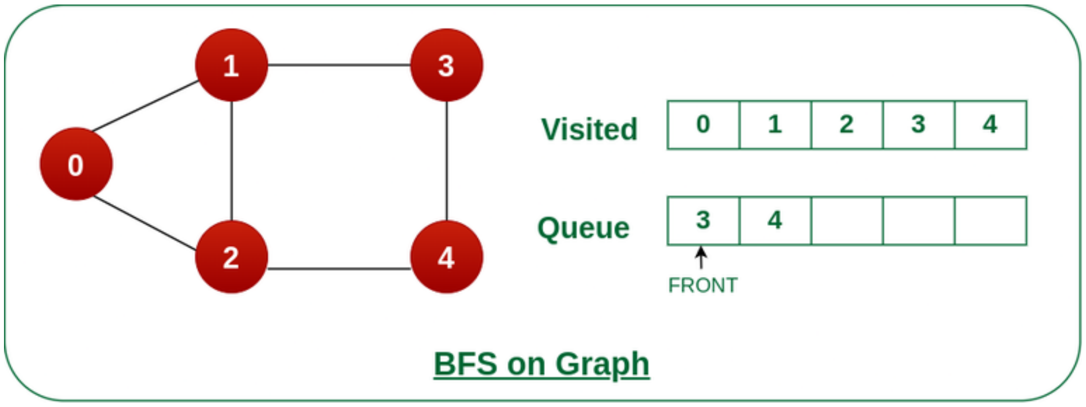
The objective of this project is to create an AI-powered solution for playing the classic snake game, leveraging a variety of algorithms including A*, BFS, DFS, and HCS. The primary goal is to develop an intelligent agent capable of navigating the game environment dynamically, avoiding obstacles, and strategically collecting food items to maximize its score.

This project aims to depict of how the AI algorithms can mimic the human tendency of playing games, by keeping the points such as the goal of eating food, avoiding clashing with the walls or the agent itself, and minimize the effort required to complete the task.

Chapter 2: Literature Review

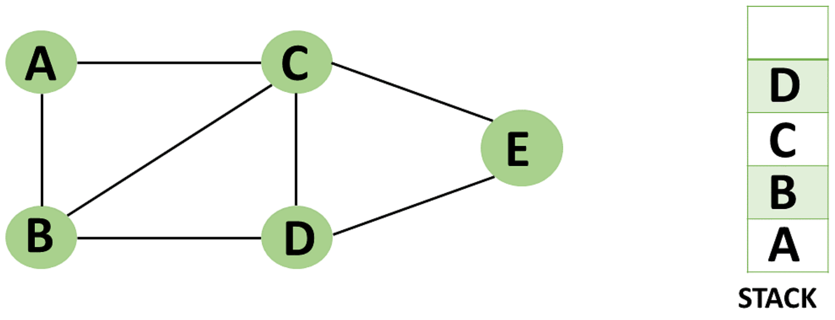
2.1. Breadth-First Search (BFS)

It is also a graph traversal algorithm in which it scans for unvisited vertices from a current node in concentric fashion. A queue is initialized with traversal starting vertex which is marked as visited. On each iteration, the algo identifies all unvisited vertices that are adjacent to the front vertex, marks them visited and adds them to the queue. Also, the front vertex is removed from the queue.



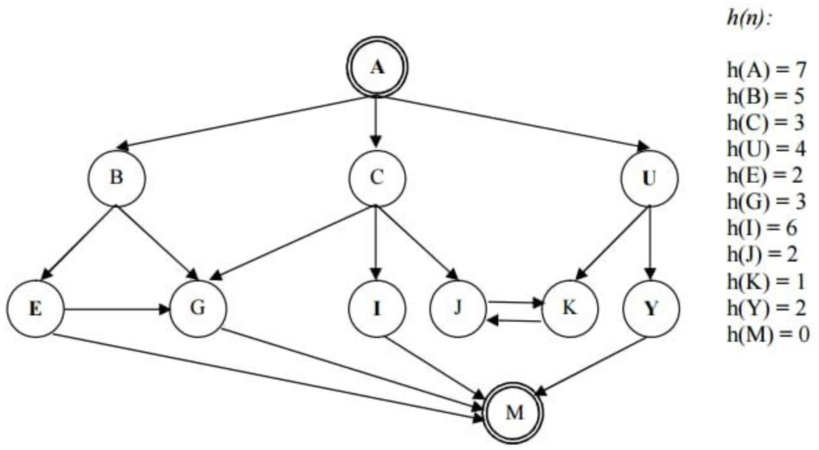
2.2. Depth-First Search (DFS)

It is a graph traversal algorithm useful in applications like finding acyclicity and connectivity of graphs. Stack data structure can be used to trace the operation of DFS. A vertex is pushed on to the stack when it is reached for the first time. A vertex is popped of the stack when it becomes dead-end. It explores as far as possible along each branch before backtracking. By maintaining a visited set, DFS avoids revisiting nodes and efficiently explores connected components.



2.3. Hill Climb Search (HCS)

Hill Climbing Search is a heuristic algorithm that iteratively improves a solution by exploring neighboring states. It evaluates these neighbors based on an objective function and selects the one with the best value. However, it can get stuck in local maxima or minima and doesn't guarantee finding the global optimum.



2.4. A* Search

A* algorithm finds the best route for traversal, considering both distances traveled and how close it is to the goal. It balances efficiency and accuracy by using smart guessing (heuristics) to decide which paths to explore first. The formula below is used to find the best node to be traversed after each iteration:

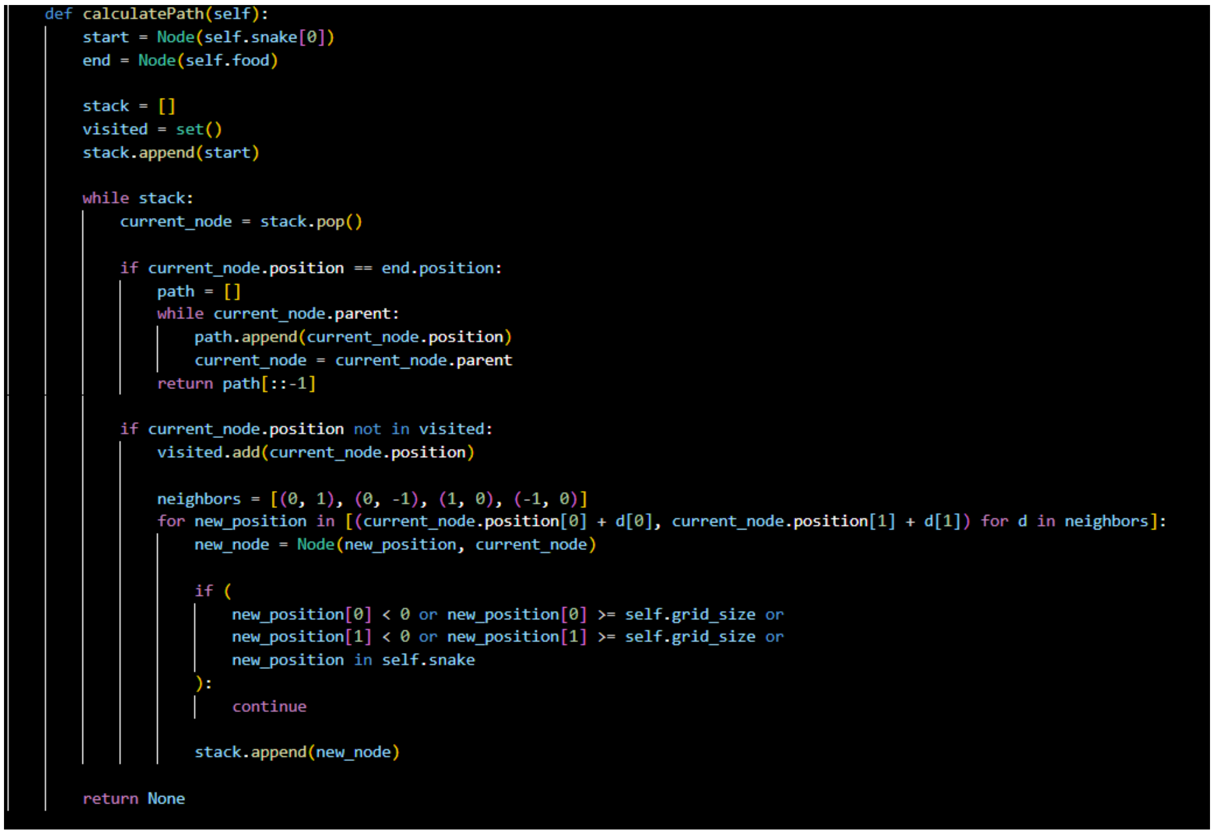
$f(n) = g(n) + h(n)$

where:

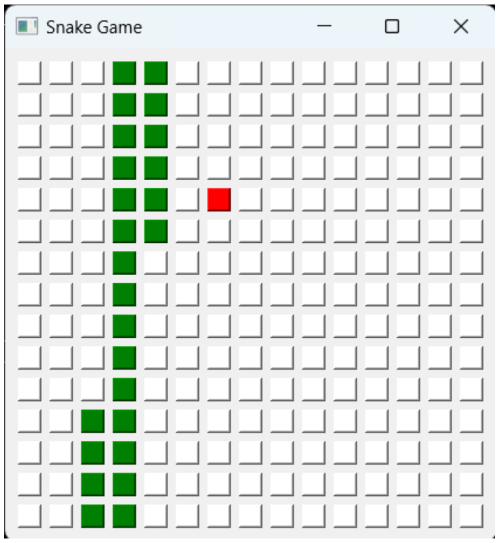
- f(n) is the total estimated cost of node ‘n’
- g(n) is the actual cost from the start node to node ‘n’
- h(n) is the estimated cost from node ‘n’

Chapter 3: Methodology

3.1. DFS Algorithm

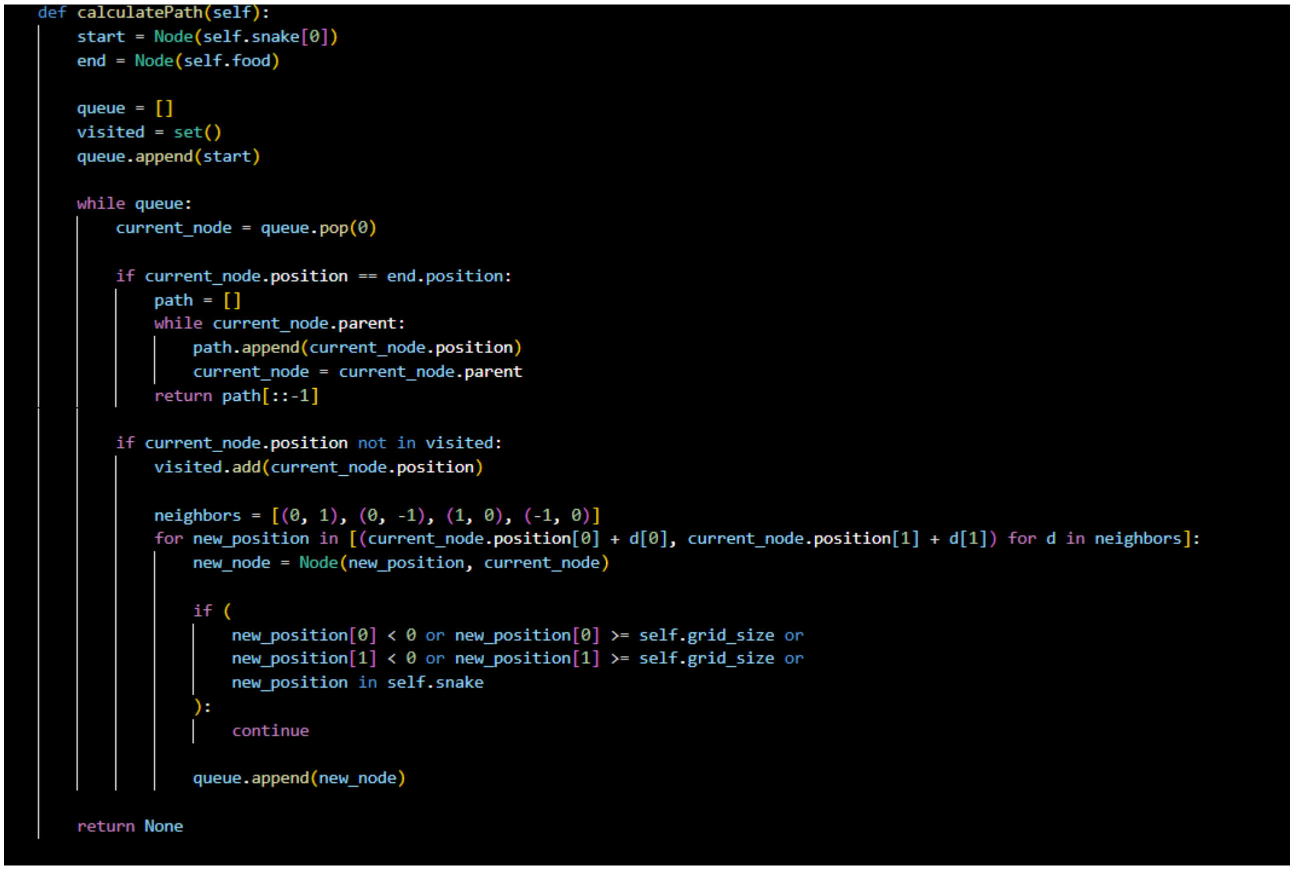


```python
def calculatePath(self):
    start = Node(self.snake[0])
    end = Node(self.food)

    stack = []
    visited = set()
    stack.append(start)

    while stack:
        current_node = stack.pop()

        if current_node.position == end.position:
            path = []
            while current_node.parent:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

        if current_node.position not in visited:
            visited.add(current_node.position)

            neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
            for new_position in [(current_node.position[0] + d[0], current_node.position[1] + d[1]) for d in neighbors]:
                new_node = Node(new_position, current_node)

                if (
                    new_position[0] < 0 or new_position[0] >= self.grid_size or
                    new_position[1] < 0 or new_position[1] >= self.grid_size or
                    new_position in self.snake
                ):
                    continue

                stack.append(new_node)

    return None
```

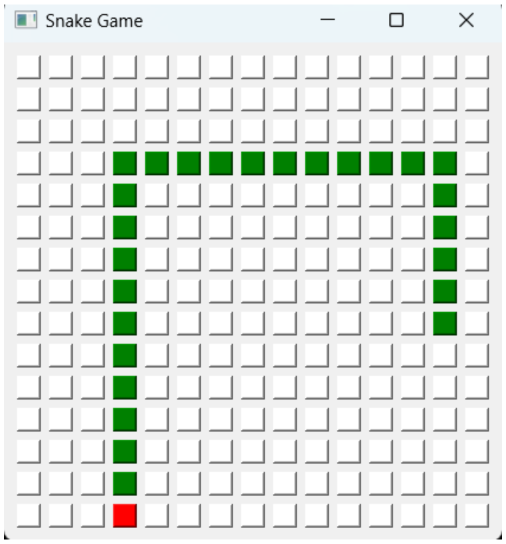


The `calculatePath` method in `SnakeGame` employs Depth-First Search (DFS) to find the shortest path from the snake's head to the food. It initializes start and end nodes, uses a stack to manage node exploration, and tracks visited nodes in a set. The neighboring nodes include the cell above, below, left, and right of the current cell. The algorithm pops nodes from the stack, checks if the current node is the goal, and backtracks if needed. It continues until the goal is reached or no valid path exists. DFS explores paths deeply, prioritizing unvisited nodes, but may not guarantee the shortest path compared to other algorithms like A*.

3.2. BFS Algorithm

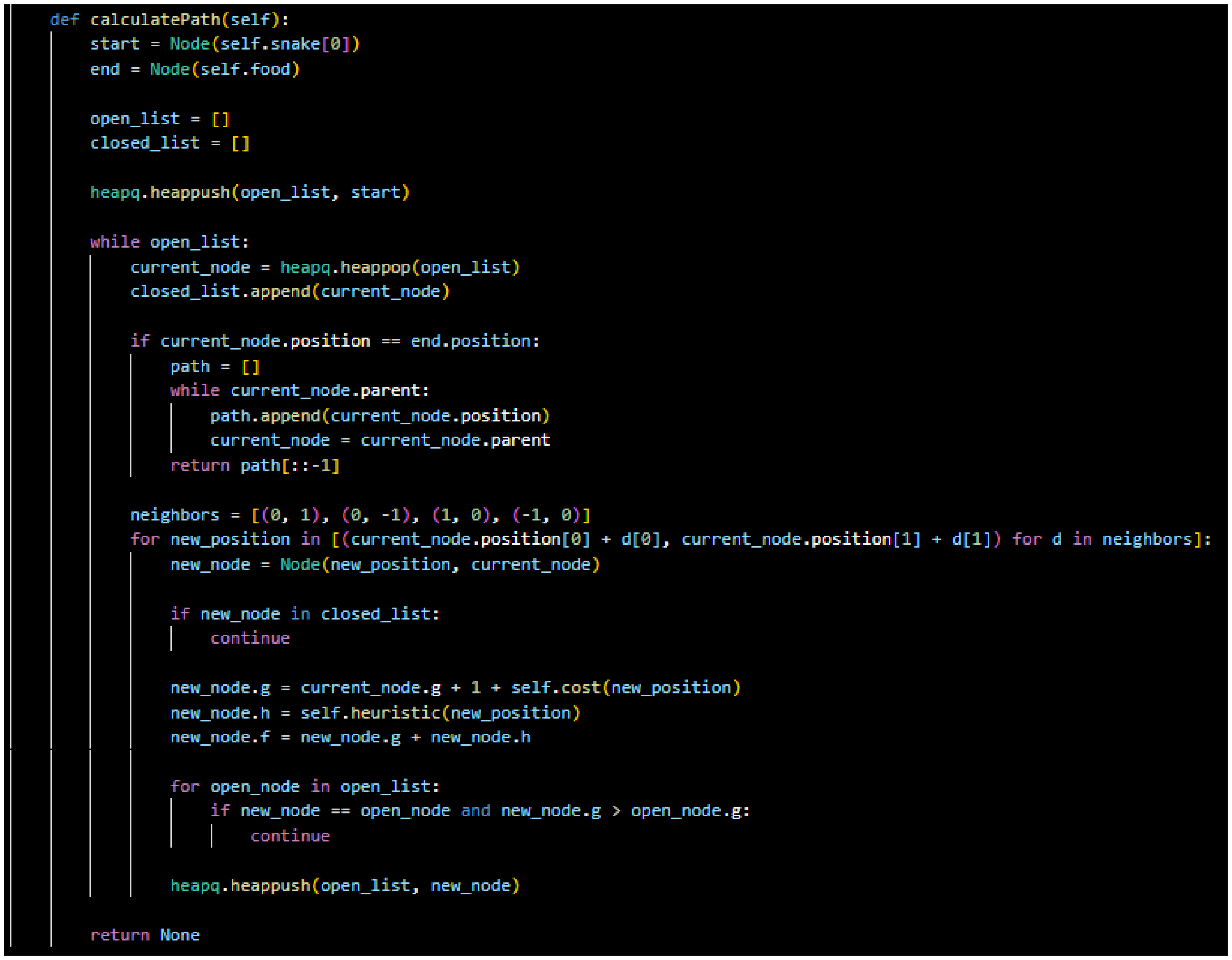


```
def calculatePath(self):
    start = Node(self.snake[0])
    end = Node(self.food)

    queue = []
    visited = set()
    queue.append(start)

    while queue:
        current_node = queue.pop(0)

        if current_node.position == end.position:
            path = []
            while current_node.parent:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

        if current_node.position not in visited:
            visited.add(current_node.position)

            neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
            for new_position in [(current_node.position[0] + d[0], current_node.position[1] + d[1]) for d in neighbors]:
                new_node = Node(new_position, current_node)

                if (
                    new_position[0] < 0 or new_position[0] >= self.grid_size or
                    new_position[1] < 0 or new_position[1] >= self.grid_size or
                    new_position in self.snake
                ):
                    continue

                queue.append(new_node)

    return None
```

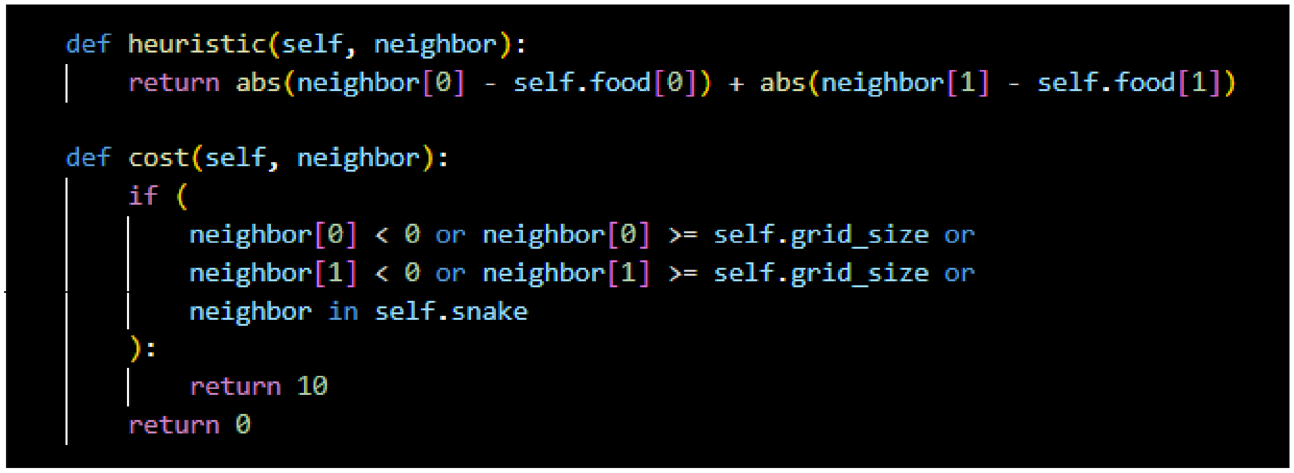


The `calculatePath` method in `SnakeGame` uses Breadth-First Search (BFS) to find the shortest path from the snake's head to the food. It initializes start and end nodes, manages a queue for node exploration, and tracks visited nodes in a set. The neighboring nodes include the cell above, below, left, and right of the current cell. The algorithm dequeues nodes, checks if the current node is the goal, and backtracks if needed. It continues until the goal is reached or no valid path exists. BFS explores all nodes at a given depth level before moving deeper, ensuring it finds the shortest path. It reconstructs and returns the shortest path if found, or None otherwise.

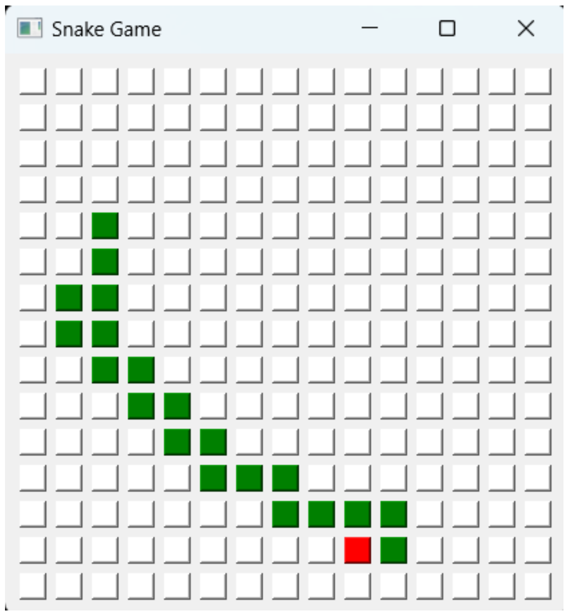
3.3. A* Algorithm



```python
def calculatePath(self):
    start = Node(self.snake[0])
    end = Node(self.food)

    open_list = []
    closed_list = []

    heapq.heappush(open_list, start)

    while open_list:
        current_node = heapq.heappop(open_list)
        closed_list.append(current_node)

        if current_node.position == end.position:
            path = []
            while current_node.parent:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

        neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        for new_position in [(current_node.position[0] + d[0], current_node.position[1] + d[1]) for d in neighbors]:
            new_node = Node(new_position, current_node)

            if new_node in closed_list:
                continue

            new_node.g = current_node.g + 1 + self.cost(new_position)
            new_node.h = self.heuristic(new_position)
            new_node.f = new_node.g + new_node.h

            for open_node in open_list:
                if new_node == open_node and new_node.g > open_node.g:
                    continue

            heapq.heappush(open_list, new_node)

    return None
```



```python
def heuristic(self, neighbor):
    return abs(neighbor[0] - self.food[0]) + abs(neighbor[1] - self.food[1])

def cost(self, neighbor):
    if (
        neighbor[0] < 0 or neighbor[0] >= self.grid_size or
        neighbor[1] < 0 or neighbor[1] >= self.grid_size or
        neighbor in self.snake
    ):
        return 10
    return 0
```



The `calculatePath` method in the `SnakeGame` class implements the A* algorithm for finding the shortest path from the snake's head to the food in the game grid. It initializes start and end nodes, manages open and closed lists to track node evaluation, and employs a loop to prioritize nodes based on their total cost, f(n).
The neighboring nodes include the cell above, below, left, and right of the current cell. This cost evaluation considers neighboring positions and updates nodes' costs accordingly. If a path is found, the method reconstructs it by tracing back from the end node to the start node through each node's parent pointers.

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

$g(n)$ = Path cost {

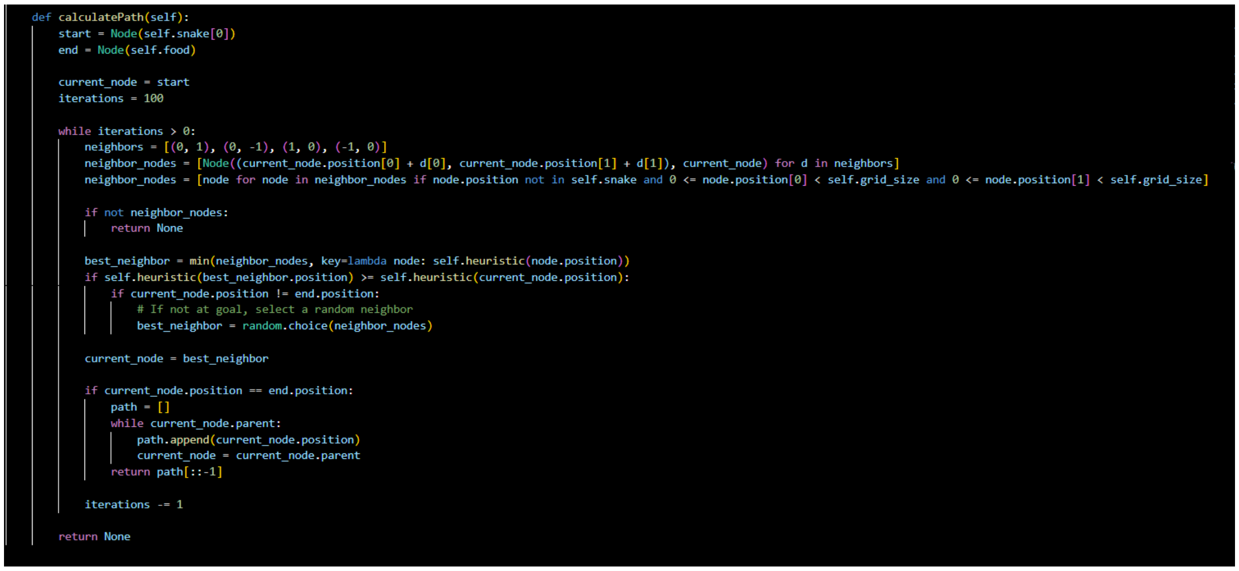
+1; if move is by one unit

+10; if hits wall or itself

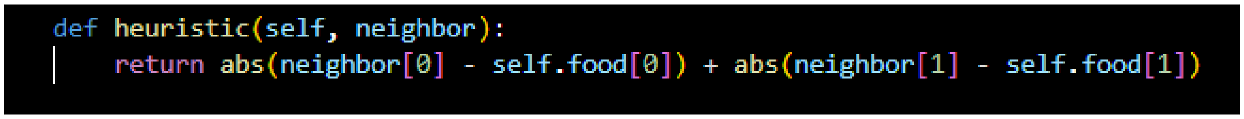
}

$h(n)$ = Heuristic cost (Manhattan distance)

3.4. Hill Climb Search (HCS) Algorithm



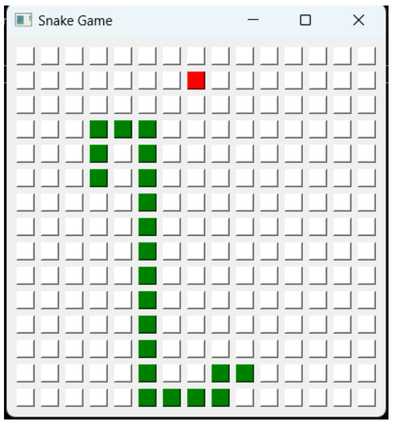
```python
def calculatePath(self):
    start = Node(self.snake[0])
    end = Node(self.food)

    current_node = start
    iterations = 100

    while iterations > 0:
        neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        neighbor_nodes = [Node((current_node.position[0] + d[0], current_node.position[1] + d[1]), current_node) for d in neighbors]
        neighbor_nodes = [node for node in neighbor_nodes if node.position not in self.snake and 0 <= node.position[0] < self.grid_size and 0 <= node.position[1] < self.grid_size]

        if not neighbor_nodes:
            return None

        best_neighbor = min(neighbor_nodes, key=lambda node: self.heuristic(node.position))
        if self.heuristic(best_neighbor.position) >= self.heuristic(current_node.position):
            if current_node.position != end.position:
                # If not at goal, select a random neighbor
                best_neighbor = random.choice(neighbor_nodes)

        current_node = best_neighbor

        if current_node.position == end.position:
            path = []
            while current_node.parent:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

        iterations -= 1

    return None
```



```python
def heuristic(self, neighbor):
    return abs(neighbor[0] - self.food[0]) + abs(neighbor[1] - self.food[1])
```

The `calculatePath` method in the `SnakeGame` class implements the Hill Climbing algorithm for pathfinding. It starts at the snake's head, evaluates the neighboring nodes which includes the cell above, below, left, and right of the current cell, and selects the one closest to the food.

The algorithm has four possible moves and checks which of them is better than the current node and one of these better nodes is randomly selected as the next node. This process repeats for a limited number of iterations until reaching the food or determining that no path exists.



Chapter 4: Result and Discussion

The UI illustrates a snake game simulation where different algorithms' results are displayed through individual codes. The simulation runs autonomously until manually stopped by the user or any collision. Each algorithm's outcome is presented similarly within the UI, showcasing how they guide the snake towards the food. This setup enables users to compare and analyze the performance of various algorithms in solving the game's pathfinding puzzle. It offers a hands-on way to observe and understand the efficiency and effectiveness of different strategies, aiding in learning and decision-making related to algorithm selection for similar tasks.

Comparative Analysis:

- Efficiency: A* emerged as the most efficient algorithm, consistently finding optimal paths with minimal computational overhead.

- Exploration: BFS excelled in exploration, covering a large portion of the search space quickly, but sometimes sacrificed optimality.

- Optimality: A* prioritized finding the shortest path, resulting in higher scores compared to BFS and DFS.

- Completeness: HCS is not able to always find a path, as it sometimes collides with itself or the wall due to stochasticity. Hence, it is not a complete algorithm.

Chapter 5: Conclusion and Future Enhancements

Conclusion

The project demonstrated how well different searching algorithms, including HCS, BFS, DFS, and A*, worked to solve the traditional snake game.

Every algorithm exhibited distinct advantages and disadvantages, which enhanced our comprehension of how well they functioned in pathfinding assignments within the gaming setting.

The best algorithm, A*, was able to efficiently navigate the snake through the game grid by consistently determining the best paths. It was able to avoid obstacles and achieve high scores due to its ability to handle variable edge costs and use heuristic information.

While BFS explored quickly, it sometimes compromised optimality for breadth, and DFS's depth-first exploration approach made it difficult to identify the best solutions.

Future Enhancements:

Enhanced Gameplay Dynamics: Introduce additional challenges such as dynamic obstacles, varying food item values, and multiple snakes to create a more dynamic and challenging gaming environment.

Advanced Heuristics: Explore the integration of more sophisticated heuristic functions in A* search to further improve pathfinding efficiency and adaptability to different game scenarios.
Reinforcement Learning: Implement reinforcement learning techniques to enable the snake agent to learn and adapt its strategies over time, enhancing its performance through experience and interaction with the game environment.

Multi-Agent Systems: Extend the project to incorporate multiple snake agents competing or collaborating in the same game space, leading to more complex and strategic gameplay dynamics.
User Interface and Interactivity: Develop a user-friendly interface with customizable settings, visualizations, and interactive features to enhance user engagement and provide insights into the AI decision-making process.

Real-Time Optimization: Optimize algorithms for real-time performance to enable seamless gameplay experiences, especially in scenarios with larger game grids or complex dynamics.

References:

1) Geeksforgeeks.com

2) Norvig, P., & Russell, S. J. (2019). Artificial Intelligence: A Modern Approach (4th ed.). Upper Saddle River, NJ: Pearson.

3) JavaTpoint. (n.d.). Hill Climbing Algorithm in AI. Retrieved from https://www.javatpoint.com/hill-climbing-algorithm-in-ai

4) Python documentation

5) www.lighthouse.manipal.edu