Final Project – ISTA 555

Snake Escape: The Ghost Challenge

Udit Chaudhary, 23842120

Introduction

The overall project is a Python implementation of the classic Snake game with a twist – the addition of blue fire(Ghost) as obstacles. The snake that moves around the board to collect food while avoiding collisions with both the edges of the board and the blue fire using Artificial Intelligence concepts. The game features multiple runs, with each run increasing in difficulty. The objective is to collect a certain number of food items in each run to progress to the next one. The project aims to provide an entertaining gaming experience while incorporating elements of AI to control the snake's movement intelligently.

Relevant AI Concepts or Methods:

The project utilizes several AI concepts and methods to control the snake's movement and decision-making process:

- **Pathfinding Algorithms:** The A* search algorithm is employed to find the shortest path from the snake's current position to the nearest food item while avoiding obstacles such as the snake's body and blue fire.
- **Heuristic Functions:** A heuristic function is used to estimate the cost of reaching the goal from a given position, guiding the A* search algorithm towards the most promising paths.
- **Search Algorithms:** The code implements various search algorithms such as breadth-first search (BFS) and iterative deepening depth-first search (IDDFS) to explore possible paths from the snake's position to the food.
- **Prediction:** The game predicts the future positions of blue fire to anticipate potential collisions and adjust the snake's path accordingly.

Overview about the Search Algorithm used:

A* Search Algorithm:

- A* is a widely used pathfinding algorithm in artificial intelligence and robotics.
- It combines the advantages of both uniform cost search and greedy best-first search by using a heuristic to guide its search.
- A* evaluates nodes based on two cost functions: the cost of reaching a node from the start and a heuristic estimate of the cost to reach the goal from that node.
- It maintains a priority queue of nodes to explore, prioritizing nodes with lower estimated total cost.
- A* guarantees finding the shortest path in a weighted graph under certain conditions, making it efficient for many pathfinding problems.

Code Chunk –

```
def a_star_search(self, start, goal, max_depth=None):

        visited = set()

        pq = [(0 + self.heuristic_distance(start, goal), 0, start, [])]

        while pq:

            _, cost, current_tile, path = heappop(pq)

            if current_tile == goal:

                return path + [goal]

            if current_tile in visited:

                continue

            visited.add(current_tile)

            for neighbor in self.adjacent_tiles(current_tile):

                if neighbor not in visited and neighbor not in
self.snake_body_as_tiles():

                    new_cost = cost + 1
                    priority = new_cost + self.heuristic_distance(neighbor, goal)
                    heappush(pq, (priority, new_cost, neighbor, path +
[current_tile]))
        return []
```

Breadth-First Search (BFS):

- BFS is a simple graph traversal algorithm that explores all the neighbor nodes at the present depth before moving on to the nodes at the next depth level.
- It starts at the root node and explores all the neighbor nodes at the present depth level before moving to the next level.
- BFS guarantees finding the shortest path in an unweighted graph from the starting node to all other nodes.
- It uses a queue data structure to keep track of the nodes to be explored, ensuring that nodes are explored in the order they are discovered.

Code Chunk –

```
def bfs_search_with_obstacles(self, start, goal, max_depth=None):
        visited = set()
        queue = deque([(start, [])])
        while queue:
            current_tile, path = queue.popleft()
            if current_tile == goal:
                return path + [goal]
            if current_tile in visited:
                continue
            visited.add(current_tile)
            for neighbor in self.adjacent_tiles(current_tile):
                if neighbor not in visited and neighbor not in
self.snake_body_as_tiles() and neighbor not in [box.position for box in
self.black_boxes]:
                    queue.append((neighbor, path + [current_tile]))
        return []
```

Iterative Deepening Depth-First Search (IDDFS):

- IDDFS is a combination of depth-first search (DFS) and breadth-first search (BFS) algorithms.
- It iteratively performs a depth limited DFS with increasing depth limits until the goal is found.
- IDDFS avoids the memory overhead of BFS by only storing information about nodes within the current depth limit.
- It guarantees finding the shortest path in a tree with uniform edge costs.
- IDDFS is particularly useful when memory usage is a concern and when the depth of the solution is unknown.

Code Chunk –

```
def iddfs_search_with_obstacles(self, start, goal, max_depth):
        for depth in range(max_depth):
            visited = set()
            stack = [(start, 0, [])]
            while stack:
                current_tile, current_depth, path = stack.pop()
                if current_tile == goal:
                    return path + [goal]
                if current_depth < depth:
                    if current_tile in visited:
                        continue
                    visited.add(current_tile)
                    for neighbor in self.adjacent_tiles(current_tile):
                        if neighbor not in visited and neighbor not in
self.snake_body_as_tiles() and neighbor not in [box.position for box in
self.black_boxes]:
                            stack.append((neighbor, current_depth + 1, path +
[current_tile]))
        return []
```

Description of Code and Programming Challenges:

The code is organized into classes representing the game elements such as the board, snake, food, and blue fire. It makes use of Pygame, a popular library for game development in Python, to handle graphics and user input. One notable challenge in the code is implementing intelligent snake movement while considering obstacles and avoiding collisions. This involves implementing pathfinding algorithms and predicting future positions of blue fire to generate optimal paths for the snake to follow. Additionally, managing game states, such as tracking the number of runs completed, time taken per run, moves taken per run and food items collected, required careful handling to ensure smooth gameplay.

Comparison:

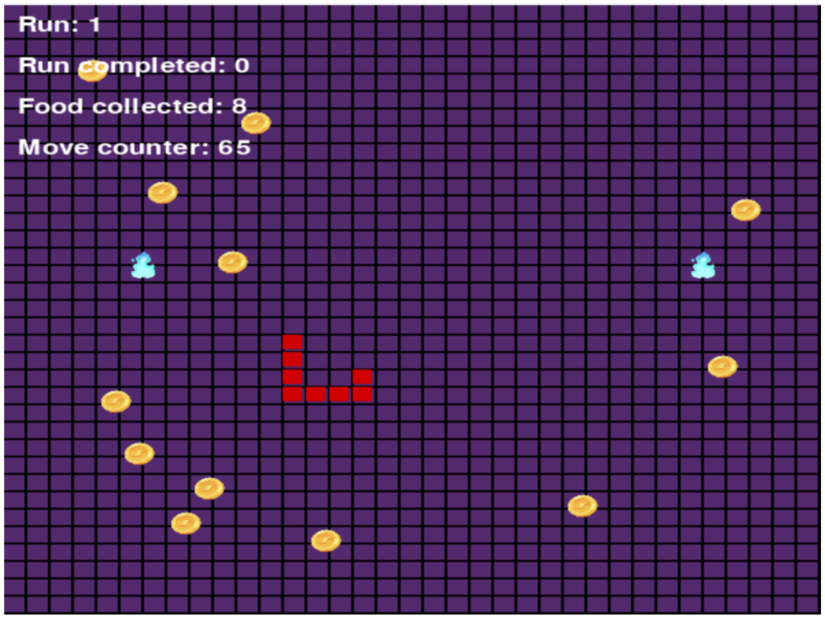
Test Run Result:

		BFS	A*	IDDFS
Test Run 1	Food Collected	20	20	13
	Moves taken	186	188	204
	Time taken	9.43	9.31	13.71
Test Run 2	Food Collected	20	20	20
	Moves taken	203	205	321
	Time taken	10.67	10.19	31.71
Test Run 3	Food Collected	19	20	4
	Moves taken	201	204	31
	Time taken	10.55	10.10	4.38

Test Run 4	Food Collected	4	4	4
	Moves taken	34	39	35
	Time taken	1.73	1.924	6.97
Test Run 5	Food Collected	20	20	6
	Moves taken	218	218	94
	Time taken	11.94	10.38	7.57
Won		**3**	**4**	**1**

Test Result Analysis:

The A* search algorithm generally performs the best across most test runs, achieving the highest number of wins and collecting the most food items. BFS also performs well, but in some cases, it falls short compared to A* in terms of the number of food items collected and the time taken. IDDFS shows varying performance, sometimes achieving fewer wins, and collecting fewer food items compared to A* and BFS. Overall, the performance of each algorithm depends on factors such as the complexity of the game board, the distribution of food items, and the placement of obstacles (blue fire). Adjusting parameters such as move delay and search depth could potentially improve the performance of each algorithm.



```
Run: 1
Run completed: 0
Food collected: 8
Move counter: 65
```

Conclusion:

In conclusion, the provided code implements a Snake game with an AI-controlled snake that utilizes various search algorithms (BFS, A*, IDDFS) to navigate the game board, collect food items, and avoid obstacles. The game features dynamic obstacles (blue fire) and multiple runs, each presenting unique challenges. The test results reveal the performance of each search algorithm in terms of food collected, moves taken, time elapsed, and the number of wins achieved across different game configurations. Overall, the A* algorithm demonstrates superior performance, consistently achieving the highest number of wins and collecting the most food items. However, the effectiveness of each algorithm varies depending on factors such as the complexity of the game board layout and the distribution of food items. Further optimization and tuning of parameters could enhance the gameplay experience and algorithm efficiency.

Reference:

- Iterative Deepening Depth-First Search (IDDFS):
  https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
  https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/
- Pygame Documentation: https://www.pygame.org/docs/