PAA REPORT

SNAKE EATER

Members of group:

1. Ahmad Danindra Nugroho (5025211259)

A. Definition

In the video game genre of "Snake," the player controls a line that expands after eating something, usually an apple, making the snake its main obstacle. In the second variation, a player tries to consume objects by striking them with the snake's head. The snake grows longer as more food is consumed. But the difference is this time I used BFS or breadth-first search which uses a queue data structure to find the shortest path. So this time the snake game will move automatically like an AI to finish the game by locating the shortest way the snake can take. When pressing the play button, the snake game will start automatically from the beginning to the end of the game.

B. Source Code

```python
# Breadth First Search Algorithm
def bfs(self, s, e):  # Find shortest path between (start_position, end_position)
    q = [s]  # Queue
    visited = {tuple(pos): False for pos in GRID}

    visited[s] = True

    # Prev is used to find the parent node of each node to create a feasible path
    prev = {tuple(pos): None for pos in GRID}

    while q:  # While queue is not empty
        node = q.pop(0)
        neighbors = ADJACENCY_DICT[node]
        for next_node in neighbors:
            if self.is_position_free(next_node) and not visited[tuple(next_node)]:
                q.append(tuple(next_node))
                visited[tuple(next_node)] = True
                prev[tuple(next_node)] = node

    path = list()
    p_node = e  # Starting from end node, we will find the parent node of each node

    start_node_found = False
    while not start_node_found:
        if prev[p_node] is None:
            return []
        p_node = prev[p_node]
        if p_node == s:
            path.append(e)
            return path
        path.insert(0, p_node)
```

```
def get_available_neighbors(self, pos):
    valid_neighbors = []
    neighbors = get_neighbors(tuple(pos))
    for n in neighbors:
        if self.is_position_free(n) and self.apple.pos != n:
            valid_neighbors.append(tuple(n))
    return valid_neighbors

def longest_path_to_tail(self):
    neighbors = self.get_available_neighbors(self.head.pos)
    path = []
    if neighbors:
        dis = -9999
        for n in neighbors:
            if distance(n, self.squares[-1].pos) > dis:
                v_snake = self.create_virtual_snake()
                v_snake.go_to(n)
                v_snake.move()
                if v_snake.eating_apple():
                    v_snake.add_square()
                if v_snake.get_path_to_tail():
                    path.append(n)
                    dis = distance(n, self.squares[-1].pos)
        if path:
            return [path[-1]]
```

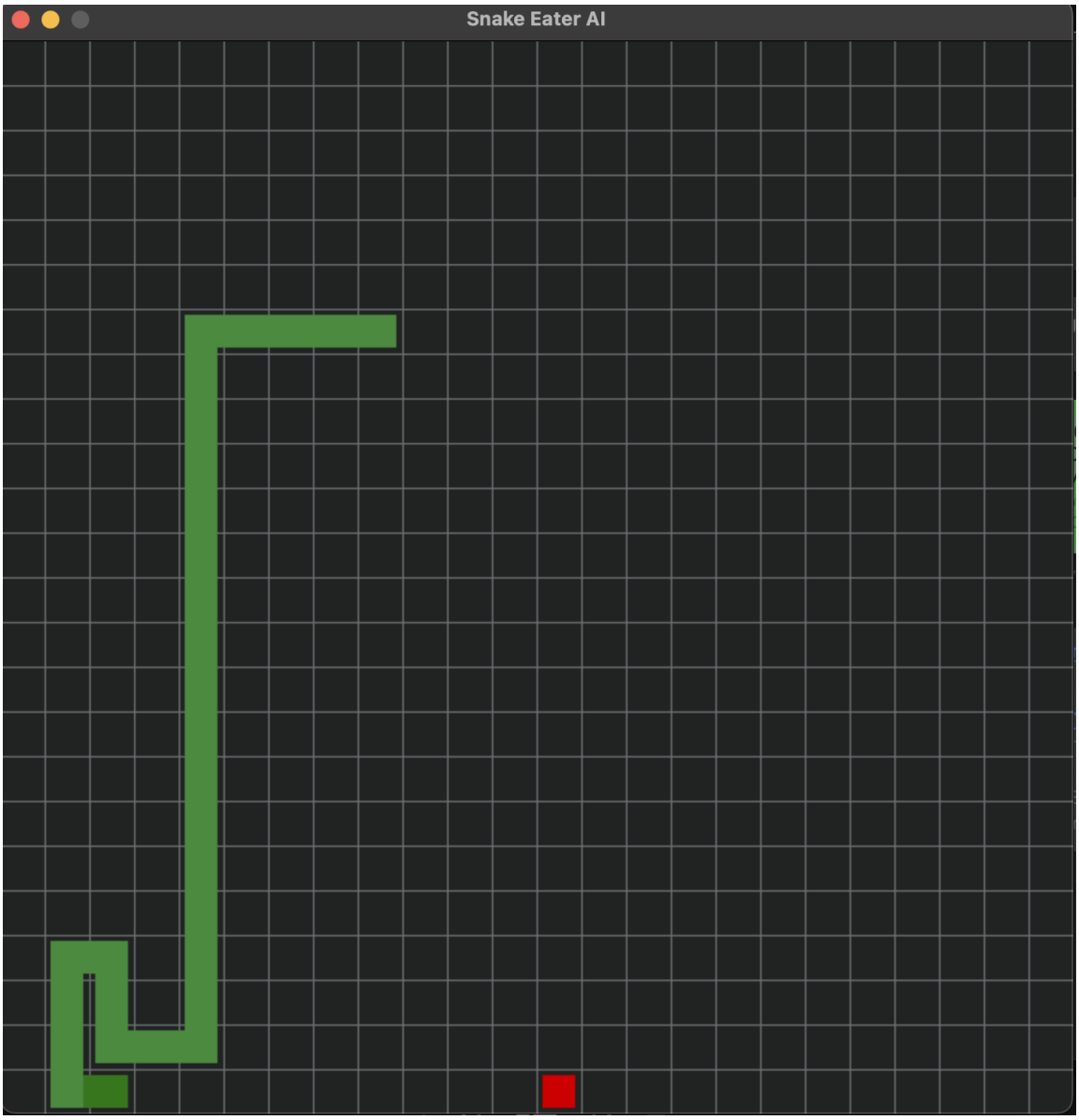
```
        return []  # Path not available

    def create_virtual_snake(self):  # Creates a copy of snake (same size, same position, etc..)
        v_snake = Snake(self.surface)
        for i in range(len(self.squares) - len(v_snake.squares)):
            v_snake.add_square()

        for i, sqr in enumerate(v_snake.squares):
            sqr.pos = deepcopy(self.squares[i].pos)
            sqr.dir = deepcopy(self.squares[i].dir)

        v_snake.dir = deepcopy(self.dir)
        v_snake.turns = deepcopy(self.turns)
        v_snake.apple.pos = deepcopy(self.apple.pos)
        v_snake.apple.is_apple = True
        v_snake.is_virtual_snake = True

        return v_snake

    def get_path_to_tail(self):
        tail_pos = deepcopy(self.squares[-1].pos)
        self.squares.pop(-1)
        path = self.bfs(tuple(self.head.pos), tuple(tail_pos))
        self.add_square()
        return path
```

```
def any_safe_move(self):
    neighbors = self.get_available_neighbors(self.head.pos)
    path = []
    if neighbors:
        path.append(neighbors[randrange(len(neighbors))])
        v_snake = self.create_virtual_snake()
        for move in path:
            v_snake.go_to(move)
            v_snake.move()
        if v_snake.get_path_to_tail():
            return path
        else:
            return self.get_path_to_tail()

def set_path(self):
    # If there is only 1 apple left for snake to win and it's adjacent to head
    if self.score == SNAKE_MAX_LENGTH - 1 and self.apple.pos in get_neighbors(self.head.pos):
        winning_path = [tuple(self.apple.pos)]
        print('Snake is about to win..')
        return winning_path

    v_snake = self.create_virtual_snake()

    # Let the virtual snake check if path to apple is available
    path_1 = v_snake.bfs(tuple(v_snake.head.pos), tuple(v_snake.apple.pos))
```

```
    # This will be the path to virtual snake tail after it follows path_1
    path_2 = []

    if path_1:
        for pos in path_1:
            v_snake.go_to(pos)
            v_snake.move()

        v_snake.add_square()  # Because it will eat an apple
        path_2 = v_snake.get_path_to_tail()

    if path_2:  # If there is a path between v_snake and it's tail
        return path_1  # Choose BFS path to apple (Fastest and shortest path)

    # If path_1 or path_2 not available, test these 3 conditions:
    # 1- Make sure that the longest path to tail is available
    # 2- If score is even, choose longest_path_to_tail() to follow the tail, if odd use any_safe_move()
    # 3- Change the follow tail method if the snake gets stuck in a loop
    if self.longest_path_to_tail() and \
            self.score % 2 == 0 and \
            self.moves_without_eating < MAX_MOVES_WITHOUT_EATING / 2:
        # Choose longest path to tail
        return self.longest_path_to_tail()

    # Play any possible safe move and make sure path to tail is available
    if self.any_safe_move():
        return self.any_safe_move()
```

```
    # If path to tail is available
    if self.get_path_to_tail():
        # Choose shortest path to tail
        return self.get_path_to_tail()

    # Snake couldn't find a path and will probably die
    print('No available path, snake in danger!')

def update(self):
    self.handle_events()

    self.path = self.set_path()
    if self.path:
        self.go_to(self.path[0])

    self.draw()
    self.move()

    if self.score == ROWS * ROWS - INITIAL_SNAKE_LENGTH:  # If snake wins the game
        self.won_game = True
        print("Snake won the game after {} moves"
              .format(self.total_moves))

        pygame.time.wait(1000 * WAIT_SECONDS_AFTER_WIN)
        return 1

    self.total_moves += 1
```

C. Output



D. Analysis

The theory we used in our program is BFS or Breadth First Search. **BFS, Breadth-First Search,** is a vertex-based technique for finding the shortest path in the graph. It uses a Queue data structure that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

So I came up with a gameplay that was as follows:
I'm going to use the Breadth-first search algorithm to find the shortest path between the head of the snake and the apple. I would create a dummy snake with the same specifications as the original snake, and let it follow this path. After the dummy snake reaches the apple. I am going to test the path between

the head of the dummy snake, and the tail of the dummy snake. If the path is available, then the first path is safe and it will be the path of the original snake. But if it is not available, I let the original snake follow its own tail. So I programmed two methods through which the tail can be followed, so that the snake's movement is more diverse. The first way is by choosing the longest path between the snake's head and the snake's tail. The second way is by choosing a path with safe random moves. So that the path is available between the head and the tail.

So in this algorithm, first I declare BFS which is useful for finding the shortest path the snake will go through. Inside BFS there is code that will find the parent of each node in order to create a feasible path. I used Adjaency dictionary for the node and declare it as neighbor.

```
# Breadth First Search Algorithm
def bfs(self, s, e):  # Find shortest path between (start_position, end_position)
    q = [s]  # Queue
    visited = {tuple(pos): False for pos in GRID}

    visited[s] = True

    # Prev is used to find the parent node of each node to create a feasible path
    prev = {tuple(pos): None for pos in GRID}

    while q:  # While queue is not empty
        node = q.pop(0)
        neighbors = ADJACENCY_DICT[node]
        for next_node in neighbors:
            if self.is_position_free(next_node) and not visited[tuple(next_node)]:
                q.append(tuple(next_node))
                visited[tuple(next_node)] = True
                prev[tuple(next_node)] = node

    path = list()
    p_node = e  # Starting from end node, we will find the parent node of each node

    start_node_found = False
    while not start_node_found:
        if prev[p_node] is None:
            return []
        p_node = prev[p_node]
        if p_node == s:
            path.append(e)
            return path
        path.insert(0, p_node)
```

After that i declare longest_path_to_tail to search the longest path to the snake tail and if [path[-1]] the path arent available.

```
def get_available_neighbors(self, pos):
    valid_neighbors = []
    neighbors = get_neighbors(tuple(pos))
    for n in neighbors:
        if self.is_position_free(n) and self.apple.pos != n:
            valid_neighbors.append(tuple(n))
    return valid_neighbors

def longest_path_to_tail(self):
    neighbors = self.get_available_neighbors(self.head.pos)
    path = []
    if neighbors:
        dis = -9999
        for n in neighbors:
            if distance(n, self.squares[-1].pos) > dis:
                v_snake = self.create_virtual_snake()
                v_snake.go_to(n)
                v_snake.move()
                if v_snake.eating_apple():
                    v_snake.add_square()
                if v_snake.get_path_to_tail():
                    path.append(n)
                    dis = distance(n, self.squares[-1].pos)
        if path:
            return [path[-1]]
```

Like i said before, i would like to make a dummy snake and i declare it to create_virtual_snake. This code will create the copy of the original snake. Thenceforth i make a code to get the path, safe move for the original snake, and set the path for the original snake.

```
        return []  # Path not available

    def create_virtual_snake(self):  # Creates a copy of snake (same size, same position, etc..)
        v_snake = Snake(self.surface)
        for i in range(len(self.squares) - len(v_snake.squares)):
            v_snake.add_square()

        for i, sqr in enumerate(v_snake.squares):
            sqr.pos = deepcopy(self.squares[i].pos)
            sqr.dir = deepcopy(self.squares[i].dir)

        v_snake.dir = deepcopy(self.dir)
        v_snake.turns = deepcopy(self.turns)
        v_snake.apple.pos = deepcopy(self.apple.pos)
        v_snake.apple.is_apple = True
        v_snake.is_virtual_snake = True

        return v_snake

    def get_path_to_tail(self):
        tail_pos = deepcopy(self.squares[-1].pos)
        self.squares.pop(-1)
        path = self.bfs(tuple(self.head.pos), tuple(tail_pos))
        self.add_square()
        return path
```

```
def any_safe_move(self):
    neighbors = self.get_available_neighbors(self.head.pos)
    path = []
    if neighbors:
        path.append(neighbors[randrange(len(neighbors))])
        v_snake = self.create_virtual_snake()
        for move in path:
            v_snake.go_to(move)
            v_snake.move()
        if v_snake.get_path_to_tail():
            return path
        else:
            return self.get_path_to_tail()

def set_path(self):
    # If there is only 1 apple left for snake to win and it's adjacent to head
    if self.score == SNAKE_MAX_LENGTH - 1 and self.apple.pos in get_neighbors(self.head.pos):
        winning_path = [tuple(self.apple.pos)]
        print('Snake is about to win..')
        return winning_path

    v_snake = self.create_virtual_snake()

    # Let the virtual snake check if path to apple is available
    path_1 = v_snake.bfs(tuple(v_snake.head.pos), tuple(v_snake.apple.pos))
```

In the set_path code, there will be code that will let the virtual snake to check if path to apple is available and if there is a path between the dummy snake and it's tail then choose BFS path to apple (the fastest and shortest path)
But there is a condition :
If path_1 or path_2 are not available

1. the longest path to tail must available
2. if the score is even choose the code longest_path_to_tail and if odd then use any_safe_move()
3. change the follow tail method if the snake stuck in a loop

```
# This will be the path to virtual snake tail after it follows path_1
path_2 = []

if path_1:
    for pos in path_1:
        v_snake.go_to(pos)
        v_snake.move()

    v_snake.add_square()  # Because it will eat an apple
    path_2 = v_snake.get_path_to_tail()

if path_2:  # If there is a path between v_snake and it's tail
    return path_1  # Choose BFS path to apple (Fastest and shortest path)

# If path_1 or path_2 not available, test these 3 conditions:
# 1- Make sure that the longest path to tail is available
# 2- If score is even, choose longest_path_to_tail() to follow the tail, if odd use any_safe_move()
# 3- Change the follow tail method if the snake gets stuck in a loop
if self.longest_path_to_tail() and \
        self.score % 2 == 0 and \
        self.moves_without_eating < MAX_MOVES_WITHOUT_EATING / 2:
    # Choose longest path to tail
    return self.longest_path_to_tail()

# Play any possible safe move and make sure path to tail is available
if self.any_safe_move():
    return self.any_safe_move()
```

and if the code cant find a path then the snake will die. After that, i create a condition if the snake win the game.

```
    # If path to tail is available
    if self.get_path_to_tail():
        # Choose shortest path to tail
        return self.get_path_to_tail()

    # Snake couldn't find a path and will probably die
    print('No available path, snake in danger!')

def update(self):
    self.handle_events()

    self.path = self.set_path()
    if self.path:
        self.go_to(self.path[0])

    self.draw()
    self.move()

    if self.score == ROWS * ROWS - INITIAL_SNAKE_LENGTH:  # If snake wins the game
        self.won_game = True
        print("Snake won the game after {} moves"
              .format(self.total_moves))

        pygame.time.wait(1000 * WAIT_SECONDS_AFTER_WIN)
        return 1

    self.total_moves += 1
```

“By the name of Allah (God) Almighty, herewith I pledge and truly declare that I have solved quiz 2 by myself, did not do any cheating by any means, did not do any plagiarism, and did not accept anybody’s help by any means. I am going to accept all of the consequences by any means if it has proven that I have done any cheating and/or plagiarism.”

Surabaya, 22 November 2022

Ahmad Danindra Nugroho