Survey Paper: Snake Game with A* Algorithm

1. Introduction

This survey paper provides an analysis of the "Snake Game with A* Algorithm" project. The project incorporates the classic Snake game mechanics with an enhanced pathfinding feature using the A* algorithm. This paper aims to evaluate the effectiveness of the A* algorithm in improving gameplay, the practical application of relevant knowledge, and the overall user experience.

2. Objective

The primary objective of this survey is to assess:

- The effectiveness of the A* pathfinding algorithm integrated into the Snake game.
- The knowledge and skills applied in implementing the algorithm and developing the game.
- User feedback and analysis of game performance and gameplay experience.

3. Project Overview

Game Title: Snake Game with A* Algorithm
Technology Used: Python, Pygame, NumPy

Features:

Classic Snake gameplay with added obstacle generation.

- Pathfinding using the A* algorithm to navigate the snake towards the food.
- Dynamic scoring system that rewards food collection.
- Game over screen that displays the final score.

4. Analysis of the A* Algorithm

4.1 Algorithm Overview

The A* (A-star) algorithm is a widely-used pathfinding and graph traversal algorithm that finds the shortest path between nodes in a grid. It combines aspects of Dijkstra's algorithm and

Greedy Best-First Search, using both actual distance (`g`) and heuristic estimate (`h`) to determine the best path.

4.2 Implementation Details

Heuristic Function:

The heuristic function used in this implementation is the Euclidean distance, calculated as:

$$h = \sqrt{(x_{\text{current}} - x_{\text{goal}})^2 + (y_{\text{current}} - y_{\text{goal}})^2}$$

This function estimates the cost from the current node to the goal, helping to guide the search process.

Cost Function:

The cost function `g` represents the actual distance traveled from the start node to the current node, incremented by 1 for each step.

Open Set and Closed Set:

Open Set:

This contains nodes to be evaluated, prioritized by their total cost `f = g + h`. It uses a list in the implementation.

Closed Set:

This contains nodes that have been evaluated and are not revisited.

4.3 Performance Analysis

Time Complexity:

The time complexity of the A* algorithm is dependent on the heuristic function. With a well-designed heuristic, the time complexity can be approximated to `O(E)`, where `E` is the number of edges in the grid. The actual complexity can vary based on the grid size and obstacle distribution.

Space Complexity:

The space complexity is `O(N)`, where `N` is the number of nodes in the grid. This is due to the storage requirements for the open and closed sets.

5. Knowledge Applied in the Domain

5.1 Game Development Knowledge

Pygame Library:

Utilized Pygame for creating the game window, handling user input, and rendering graphical elements. This involves knowledge of game loop management, event handling, and graphical rendering.

Algorithm Integration:

Applied pathfinding algorithms to enhance game AI, making the snake's movement towards the food more intelligent and less random. This showcases the integration of algorithmic thinking into game development.

5.2 Computational Algorithms

A* Algorithm Application:

Demonstrated the application of A* in a real-time environment. This involved implementing a heuristic function, managing the open and closed sets, and ensuring that the algorithm could handle dynamic changes in the game state, such as obstacles and snake movement.

Heuristic Design:

Applied knowledge of heuristic functions to balance pathfinding accuracy and computational efficiency. The Euclidean distance was chosen for its simplicity and effectiveness in estimating the shortest path.

6. Analysis of Results

6.1 User Experience Feedback

Effectiveness of A* Pathfinding:

Users reported that the A* algorithm effectively guided the snake towards the food, improving the game's strategic element. The pathfinding made the game more engaging by reducing randomness in snake movement.

Difficulty Level:

Feedback varied, with some users finding the game challenging and enjoyable due to the obstacles, while others suggested that the difficulty could be adjusted to provide a better balance. Implementing adjustable difficulty settings could enhance the overall experience.

6.2 Game Performance

Responsiveness:

The game maintained smooth performance with minimal lag, indicating that the A* algorithm was efficiently implemented for the grid size used. The responsiveness of the controls and the real-time updates were well-received.

Scoring and Motivation:

The dynamic scoring system was appreciated, with users finding it motivating to collect food and improve their scores. The scoring system added a competitive element to the game.

6.3 Areas for Improvement

Obstacle Handling:

Some users suggested improvements in handling complex or dynamic obstacles. Enhancements could include more sophisticated obstacle patterns or interactive obstacles that change positions during gameplay.

Difficulty Scaling:

Users recommended implementing difficulty levels or adaptive difficulty based on player performance. This could help cater to a broader range of skill levels and improve game balance.

CODE IMPLEMENTATION:

```
import pygame
from pygame import display, time, draw, QUIT, init, KEYDOWN, K_a, K_s, K_d, K_w
from random import randint
from numpy import sqrt

# Initialize Pygame
init()

# Game settings
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
BLUE = (0, 0, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)

cols = 25
rows = 25

width = 600
height = 600
wr = width / cols
hr = height / rows
direction = 1

screen = display.set_mode([width, height])
```

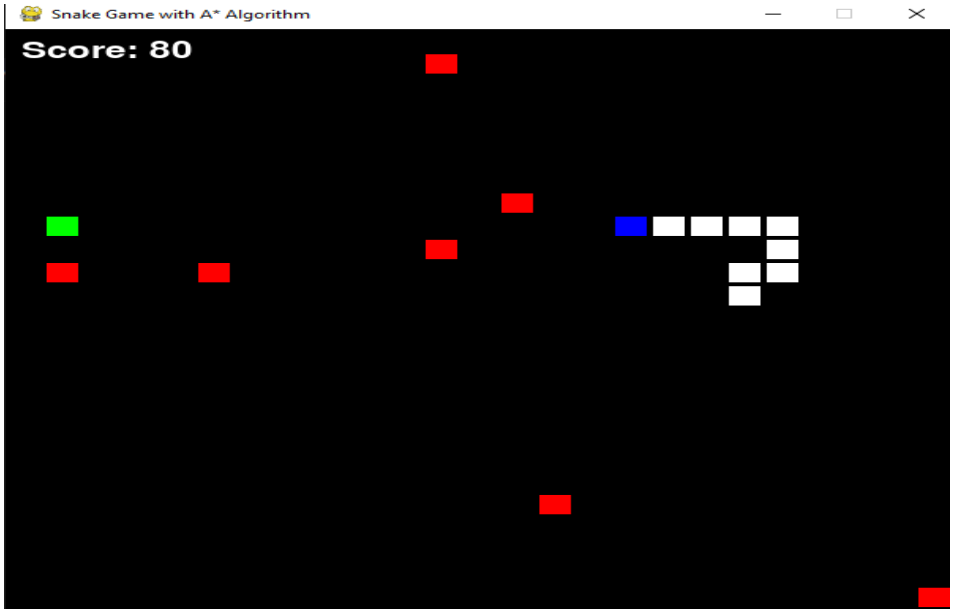
```
display.set_caption("Snake Game with A* Algorithm")
clock = time.Clock()
done = False

class Spot:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.f = 0
        self.g = 0
        self.h = 0
        self.neighbors = []
        self.camefrom = None
        self.obstacle = False
        if randint(1, 101) < 3:  # 3% chance of obstacle
            self.obstacle = True

    def show(self, color):
        draw.rect(screen, color, [self.x * hr + 2, self.y * wr + 2, hr - 4, wr - 4])

    def add_neighbors(self):
        if self.x > 0:
            self.neighbors.append(grid[self.x - 1][self.y])
        if self.y > 0:
            self.neighbors.append(grid[self.x][self.y - 1])
        if self.x < rows - 1:
            self.neighbors.append(grid[self.x + 1][self.y])
        if self.y < cols - 1:
            self.neighbors.append(grid[self.x][self.y + 1])

# A* pathfinding algorithm
def getpath(food, snake):
    food.camefrom = None
    for s in snake:
        s.camefrom = None
    openset = [snake[-1]]
    closedset = []
    dir_array = []

    while openset:
        current = min(openset, key=lambda x: x.f)
        openset.remove(current)
        closedset.append(current)

        # If the goal has been reached
        if current == food:
            break

        for neighbor in current.neighbors:
            if neighbor in closedset or neighbor.obstacle or neighbor in snake:
                continue

            temp_g = current.g + 1
            if neighbor in openset:
                if temp_g < neighbor.g:
                    neighbor.g = temp_g
            else:
                neighbor.g = temp_g
                openset.append(neighbor)

            neighbor.h = sqrt((neighbor.x - food.x) ** 2 + (neighbor.y - food.y) ** 2)
            neighbor.f = neighbor.g + neighbor.h
            neighbor.camefrom = current

    # No path found
    if current != food:
        return []

    # Reconstruct path
    while current.camefrom:
        if current.x == current.camefrom.x and current.y < current.camefrom.y:
            dir_array.append(2)  # Up
        elif current.x == current.camefrom.x and current.y > current.camefrom.y:
            dir_array.append(0)  # Down
        elif current.x < current.camefrom.x and current.y == current.camefrom.y:
            dir_array.append(3)  # Left
        elif current.x > current.camefrom.x and current.y == current.camefrom.y:
            dir_array.append(1)  # Right
        current = current.camefrom

    # Reset nodes
    for i in range(rows):
        for j in range(cols):
            grid[i][j].camefrom = None
            grid[i][j].f = 0
            grid[i][j].g = 0
            grid[i][j].h = 0

    return dir_array[::-1]  # Reverse the direction array

# Initialize grid
grid = [[Spot(i, j) for j in range(cols)] for i in range(rows)]
for i in range(rows):
    for j in range(cols):
        grid[i][j].add_neighbors()

snake = [grid[round(rows / 2)][round(cols / 2)]]
food = grid[randint(0, rows - 1)][randint(0, cols - 1)]
```

```
current = snake[-1]
dir_array = getpath(food, snake)
food_array = [food]
score = 0

def display_score(score):
    font = pygame.font.SysFont(None, 35)
    score_text = font.render(f'Score: {score}', True, WHITE)
    screen.blit(score_text, [10, 10])

def game_over_screen(score):
    font = pygame.font.SysFont(None, 55)
    game_over_text = font.render('Game Over!', True, WHITE)
    score_text = font.render(f'Final Score: {score}', True, WHITE)

    screen.fill(BLACK)
    screen.blit(game_over_text, [width // 2 - 150, height // 2 - 30])
    screen.blit(score_text, [width // 2 - 150, height // 2 + 30])
    display.flip()

    # Wait for the user to close the window
    waiting_for_close = True
    while waiting_for_close:
        for event in pygame.event.get():
            if event.type == QUIT:
                waiting_for_close = False
        clock.tick(10)

while not done:
    clock.tick(12)
    screen.fill(BLACK)

    # Get next direction
    if not dir_array:
        dir_array = getpath(food, snake)
    if dir_array:
        direction = dir_array.pop(0)

    # Determine next position
    if direction == 0:  # Down
        next_pos = grid[current.x][current.y + 1]
    elif direction == 1:  # Right
        next_pos = grid[current.x + 1][current.y]
    elif direction == 2:  # Up
        next_pos = grid[current.x][current.y - 1]
    elif direction == 3:  # Left
        next_pos = grid[current.x - 1][current.y]
    else:
        next_pos = current

    # Check boundaries
    if next_pos.x < 0 or next_pos.x >= rows or next_pos.y < 0 or next_pos.y >= cols:
        done = True
        continue

    # Check for self-collision
    if next_pos in snake:
        done = True
        continue

    # Move snake
    snake.append(next_pos)
    current = next_pos

    # Check for food collection
    if current.x == food.x and current.y == food.y:
        while True:
            food = grid[randint(0, rows - 1)][randint(0, cols - 1)]
            if not (food.obstacle or food in snake):
                break
        food_array.append(food)
        dir_array = getpath(food, snake)
        score += 10
    else:
        snake.pop(0)

    # Draw snake, obstacles, and food
    for spot in snake:
        spot.show(WHITE)
    for i in range(rows):
        for j in range(cols):
            if grid[i][j].obstacle:
                grid[i][j].show(RED)
    food.show(GREEN)
    snake[-1].show(BLUE)
    display_score(score)
    display.flip()

    # Handle events
    for event in pygame.event.get():
        if event.type == QUIT:
            done = True

# Display final score and game over screen
game_over_screen(score)
print(final Score: {score}')
pygame.quit()
```

OUTPUT:

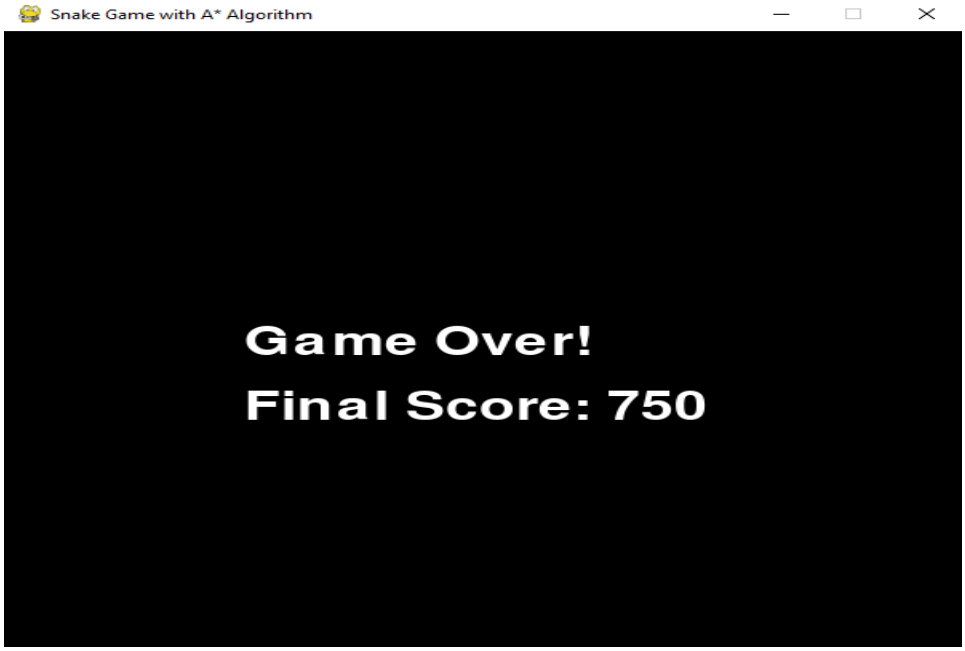
In-Game Action:

This screenshot captures the Snake game during active gameplay. It shows the snake navigating the grid, avoiding obstacles, and moving towards the food with the guidance of the A* pathfinding algorithm.



Game over:

This screenshot shows the game over screen. It appears when the game ends due to boundary collision or self-collision. The screen displays the final score and provides a summary of the player's performance.



OUTPUT IN CONSOLE:

```
PS C:\Users\ELCOT\Desktop\snake_ai> python snake_game_ai.py
pygame 2.6.0 (SDL 2.28.4, Python 3.12.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
Final Score: 750
```

7. Conclusion

The integration of the A* algorithm into the Snake game effectively demonstrated the practical application of pathfinding techniques in game development. The feedback collected highlights both the strengths of the implementation and areas for improvement, providing valuable insights for future enhancements.