International Business Academy

In collaboration with

Bsc(Hons) in Informatics

# Portfolio – Blog of Weekly Tasks

| | |
|---|---|
| **Student Name:** | **Module Leader:** |
| Anurag sah | Lene Erbs |
| Word Count: 1278 | KOL304CR |
| 26/03/2025 | |

## Contents

## 1. Introduction:

This report presents an analysis of three AI techniques used in real-time games. The study aims to evaluate their implementation, strengths, and challenges. The portfolio-based approach allows for a practical exploration of AI concepts relevant to game development.

## 2. AI Techniques Overview

The following three AI techniques have been selected for analysis:

### 2.1 Brute Force Search

**Description:** It is the basic technique of brute force search in AI, where you systematically explore all possibilities to find a solution. It is usually used in problem solving scenarios: where all possible solutions need to be determined to be optimal. Although simple and easy to use, the complexity of brute force search in exponential time makes it impractical for complex problems. In games, this technique is often used for search for extreme movement or problem solving in board games.

**Implementation:**

- Implemented a basic AI that exhaustively checks all possible moves in a game scenario.
- Applied to a simple number guessing game AI to determine optimal moves.
- Added player mode and brute force mode to play game.

**Example Code in Python:**

```python
# Function for the brute force mode

def brute_force_guess(target):

    guess = 1
    attempts = 0
    start_time = time.time()
    while guess != target:
        print(f"Brute force trying: {guess}")
        guess += 1
        attempts += 1
    end_time = time.time()
    print(f"Brute force found the number {target} in {attempts} attempts!")
    print(f"Time taken: {end_time - start_time:.2f} seconds.")
```

**Reflection:**

- Handling user input
- Implement the Brute force mode
- Game mode selection i.e. Player mode or Brute force mode

**Strength of Brute force**

- Simplicity: Easy to implement and understand.
- Guaranteed Result: Always finds the correct answer by checking all possibilities.
- Transparency: Process is visible, useful for educational purposes.
- No Complex Logic: No need for advanced algorithms or feedback.

**Weakness of Brute Force**

- Inefficiency: Performs a linear search, slow for large ranges.
- Lack of Intelligence: Doesn't use feedback to optimize guesses.
- Poor User Experience: Can be tedious or frustrating to watch for large ranges.
- Time Consumption: Takes too long for large search spaces or time-sensitive tasks.

## 2.2 Pathfinding with A* Algorithm

**Description:** In modern gaming, the pathfinding algorithm that is most commonly used in real time gaming is A-Star. A* is used to find the most proficient way to connect two points on a graph, and it does this as effectively as possible. A* combines the strength of Dijkstra's algorithm and the greedy best-first search by taking into account both the estimated cost and the actual cost. Merging these factors makes A* suitable for dynamic AI navigation in games.

**Implementation:**

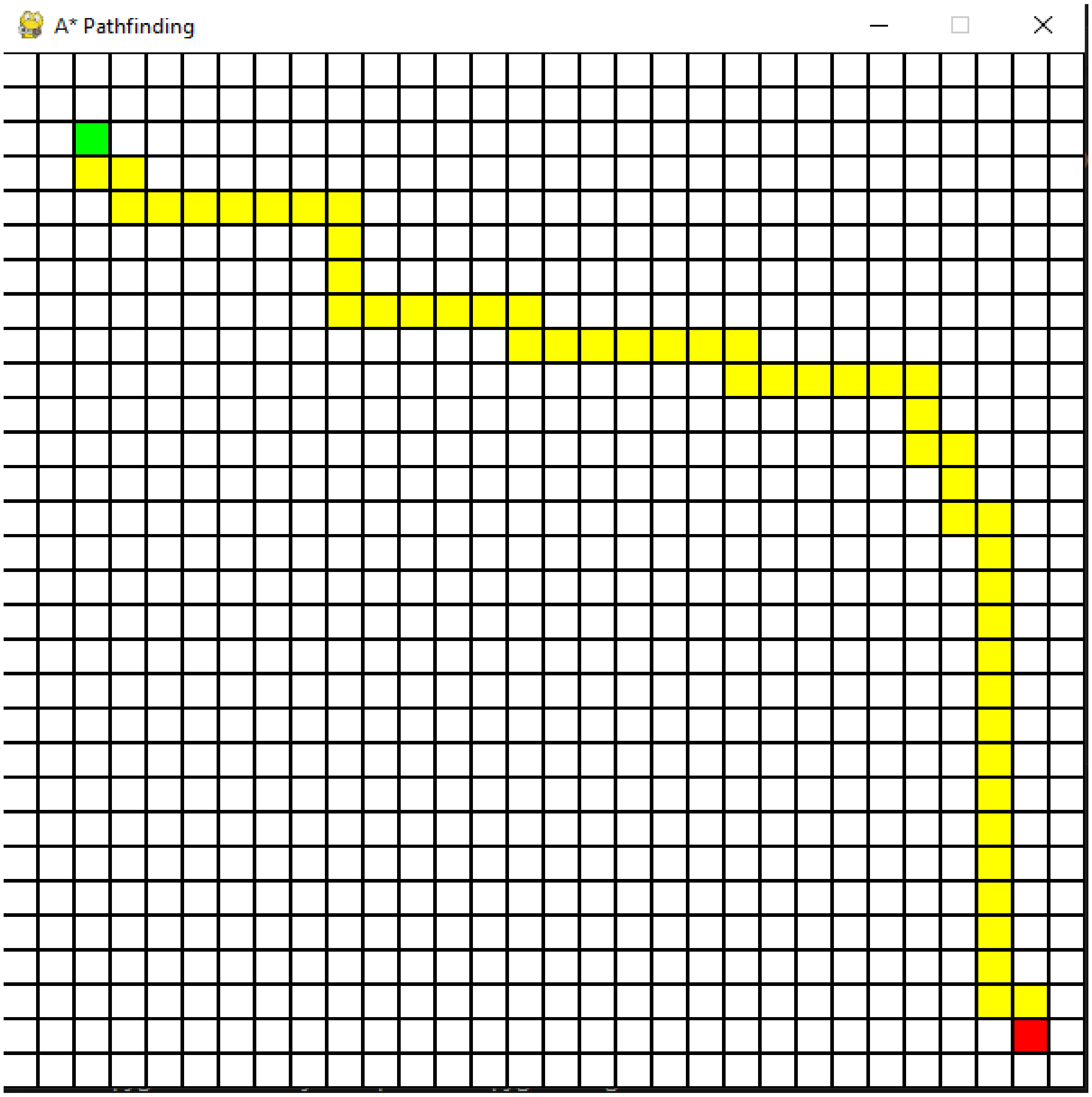
- Applied A* to navigate NPCs in a 2D grid environment.
- Integrated heuristic-based path calculations for efficiency.
- Visualized paths using debug lines.

**Example Code in Python:**

```python
# A* Algorithm

def a_star(grid, start, end):
    open_set = []
    closed_set = set()

    start.g = 0
    start.h = abs(start.row - end.row) + abs(start.col - end.col)
    start.f = start.g + start.h

    heapq.heappush(open_set, start)

    while open_set:
        current_node = heapq.heappop(open_set)
        closed_set.add(current_node)

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

        for neighbor in get_neighbors(current_node, grid):
            if neighbor in closed_set or neighbor.is_obstacle:
                continue

            tentative_g = current_node.g + 1

            if tentative_g < neighbor.g:
                neighbor.parent = current_node
                neighbor.g = tentative_g
                neighbor.h = abs(neighbor.row - end.row) + abs(neighbor.col -
end.col)
                neighbor.f = neighbor.g + neighbor.h

                if neighbor not in open_set:
                    heapq.heappush(open_set, neighbor)

    return []
```

**Reflection:**

- The implementation is straightforward, making it easy to understand the core logic of A*.
- Real-time interaction allows users to set start, end, and obstacle points, making the visualization interactive and engaging.
- Colors are used to distinguish between start, goal, obstacles, and the path, which helps in visualizing how A* works.



**Strength of A***

- A* is **optimal** when the heuristic is admissible.
- A* is **efficient**, exploring fewer nodes compared to algorithms like Dijkstra.
- The algorithm is **flexible**, allowing for different heuristic functions based on the problem.
- It is a **complete** algorithm, meaning it will always find a solution if one exists.

**Weakness of A***

- A* can be **memory-intensive**, storing all visited nodes in the open and closed sets.
- Its **computational complexity** can be high, especially if the heuristic isn't well-designed.
- A* requires a **good heuristic**; a poor one can lead to inefficient performance.
- The algorithm isn't ideal for **dynamic environments** where obstacles can change.

## 2.3 Dijkstra's Algorithm

**Description:** Dijkstra's algorithm is a classical algorithm for determining the shortest path between two points in a weighted graph. Unlike A*, it does not utilize heuristics, which makes it optimal for general pathfinding but less so for environments, such as games, where an estimated cost to the goal can improve efficiency. It is applied more frequently in AI navigation and resource management of strategy games where exact short paths are needed.

**Implementation:**

- Applied Dijkstra's algorithm for
- Implemented a priority queue to enhance efficiency.
- Compared performance with A*.

**Example Code in Python:**

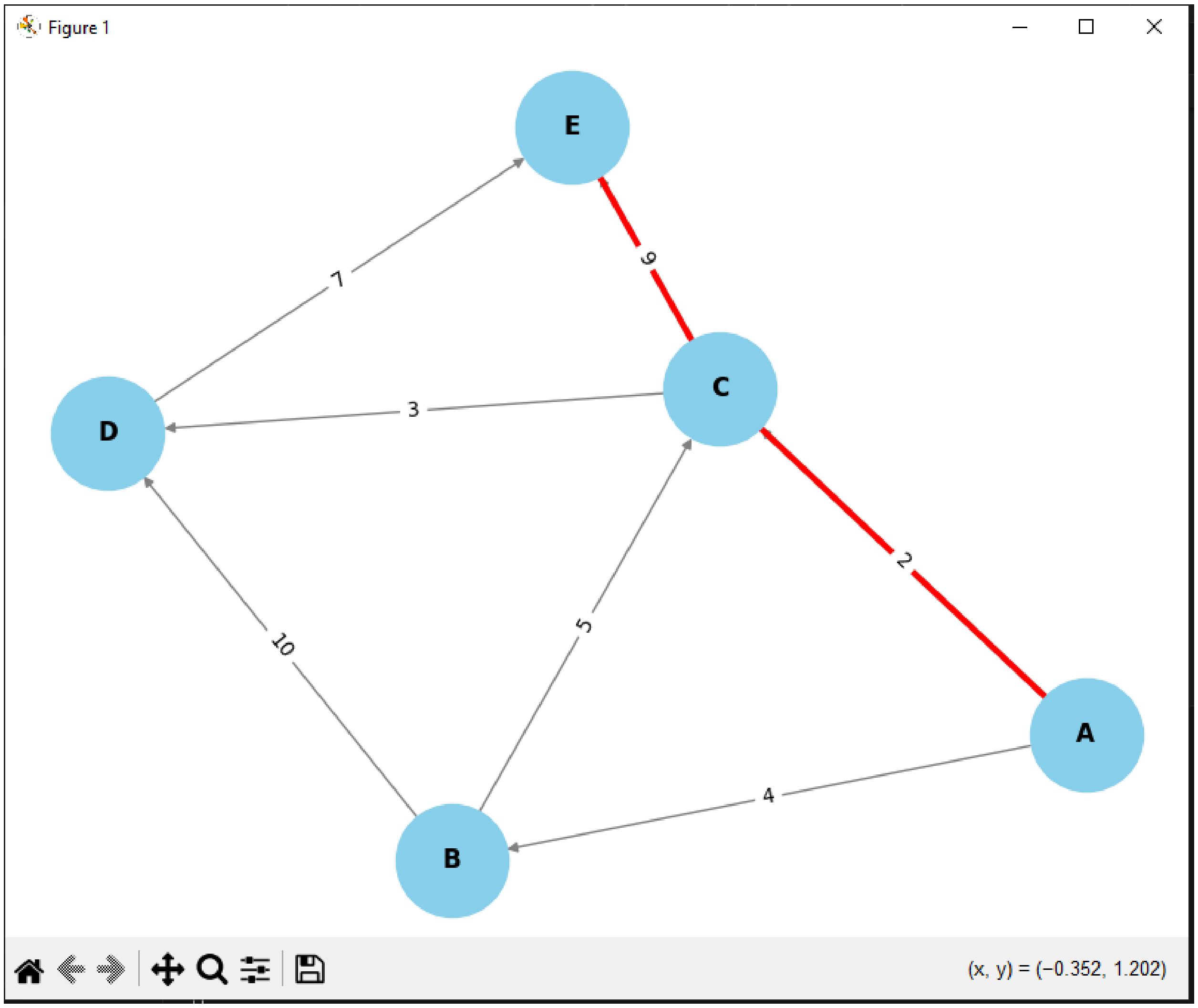
```
import networkx as nx
import matplotlib.pyplot as plt

# Create a directed graph
G = nx.DiGraph()
```

```python
# Add edges along with weights
edges = [
    ('A', 'B', 4),
    ('A', 'C', 2),
    ('B', 'C', 5),
    ('B', 'D', 10),
    ('C', 'D', 3),
    ('C', 'E', 9),
    ('D', 'E', 7)
]

G.add_weighted_edges_from(edges)

# Apply Dijkstra's algorithm to find the shortest path from A to E
shortest_path = nx.dijkstra_path(G, source='A', target='E', weight='weight')
path_length = nx.dijkstra_path_length(G, source='A', target='E', weight='weight')

# Visualize the graph
pos = nx.spring_layout(G)
plt.figure(figsize=(8, 6))

# Draw the graph
nx.draw(G, pos, with_labels=True, node_size=3000, node_color='skyblue',
font_size=12, font_weight='bold', edge_color='gray')

# Highlight the shortest path
edges_in_path = list(zip(shortest_path, shortest_path[1:]))
nx.draw_networkx_edges(G, pos, edgelist=edges_in_path, edge_color='red', width=3)

# Show the weights on edges
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=10)

plt.title(f"Shortest Path from A to E: {shortest_path} (Length: {path_length})")
plt.show()

# Print the shortest path and its length
print(f"Shortest path from A to E: {shortest_path}")
print(f"Path length: {path_length}")
```

**Reflection:**

- Create a directed graph using networkx.DiGraph().
- Define edges with weights for the graph.

- Add the edges to the graph using G.add_weighted_edges_from().
- Use Dijkstra's algorithm to find the shortest path and its length with nx.dijkstra_path() and nx.dijkstra_path_length().
- Set up the graph visualization using matplotlib.pyplot.
- Calculate the positions for nodes using nx.spring_layout().
- Draw the graph with nx.draw(), specifying node size, color, and edge color.
- Highlight the shortest path by identifying the edges in the shortest path and drawing them with red color using nx.draw_networkx_edges().
- Display the edge weights on the graph with nx.draw_networkx_edge_labels().
- Display the shortest path and its length using plt.title()



**Strength of A***

- Guarantees the shortest path in graphs with non-negative edge weights.
- Works efficiently for graphs with a relatively low number of nodes and edges.
- Can handle large graphs if combined with appropriate data structures (like heaps or priority queues).
- Simple to implement and understand.

**Weakness of A***

- Weakness: Inefficient for graphs with negative weight edges, as it may not return correct results.
- Weakness: Requires recalculation of paths for every new query (no dynamic updates of paths once calculated).
- Weakness: The algorithm is not parallelizable, making it slower for very large graphs in distributed systems.

## 3. Conclusion

The application and assessment of Brute Force Search, A* pathfinding, and Dijkstra's Algorithm within the context of real-time games was carried out for this portfolio. Brute Force yields the best results, but is very costly in terms of calculations. A* is good for pathfinding, but takes a lot of resources, and Dijkstra's Algorithm yields the best shortest paths but takes more time than A*. Knowledge of these methods improves one's ability to construct complex game AI systems.