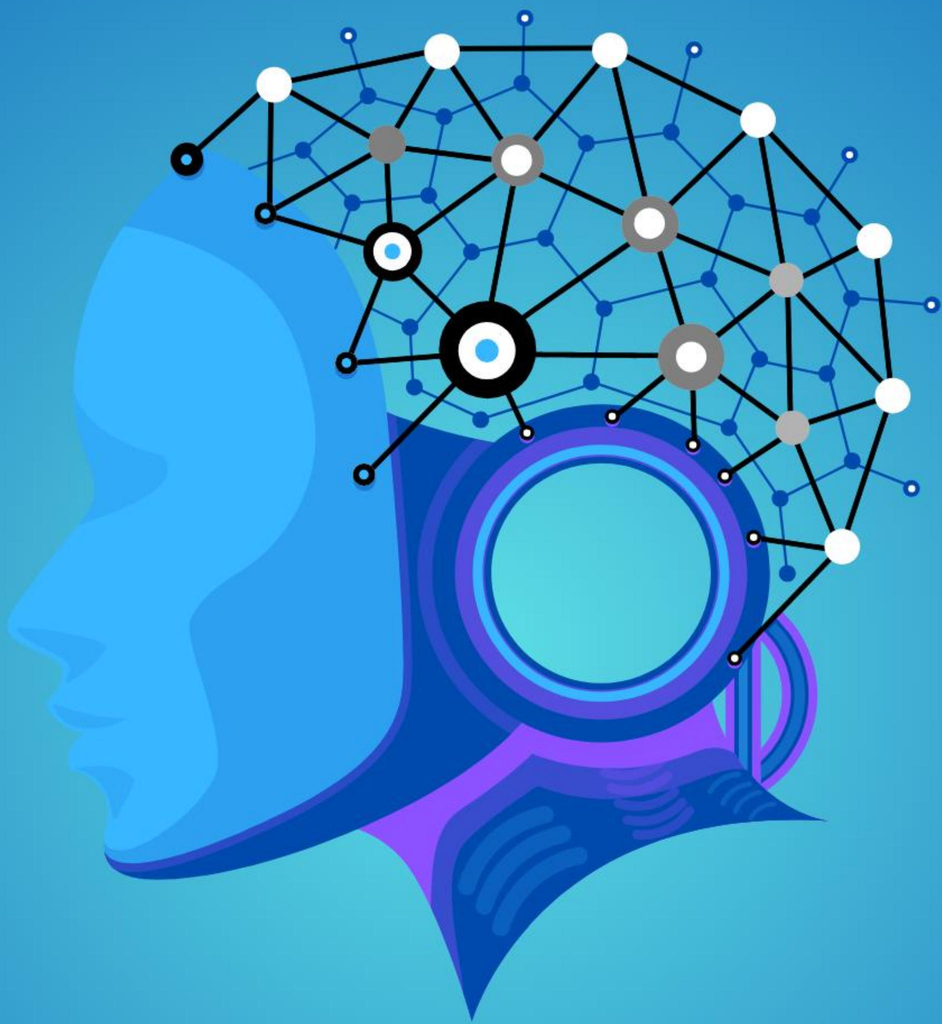


Course: Top-Up Bachelor Of Informatics

Module Code: KOL304CR

Module: Games and AI

Module Leader: Lena Erbs

Word Count: 1958

Name: Rabindra Kumar Sah

Contents

1. Brute Force (Algorithm) 3

Description of the Technique: 3

Implementation: 3

Visualisation: 6

Reflection: 6

2. Dijkstra's Algorithm (Algorithm) 8

Description of the Technique: 8

Implementation: 8

Visualisation: 11

Reflection: 11

3. **A Algorithm (Heuristic)*** 13

Description of the Technique: 13

Implementation: 13

Visualisation: 16

Reflection: 17

Conclusion: 17

GIT Hub: https://github.com/DRabindra/Portfolio-Blog-of-Weekly-Tasks---CW1.git

1. Brute Force (Algorithm)

Description of the Technique:

Brute force is a straightforward approach to problem-solving that involves checking all possible solutions to find the best one. In the context of game AI, brute force can be used for tasks like pathfinding, decision-making, or solving puzzles. While it guarantees finding the optimal solution, it is highly inefficient for large problems due to its exponential time complexity.

- **Role in Game AI:** Brute force is rarely used in real-time games because of its inefficiency, but it can be useful for small-scale problems or as a baseline for comparing more advanced algorithms.
- **Strengths:** Simple to implement, guarantees the optimal solution.
- **Weaknesses:** Extremely slow for large datasets, impractical for real-time applications.

Implementation:

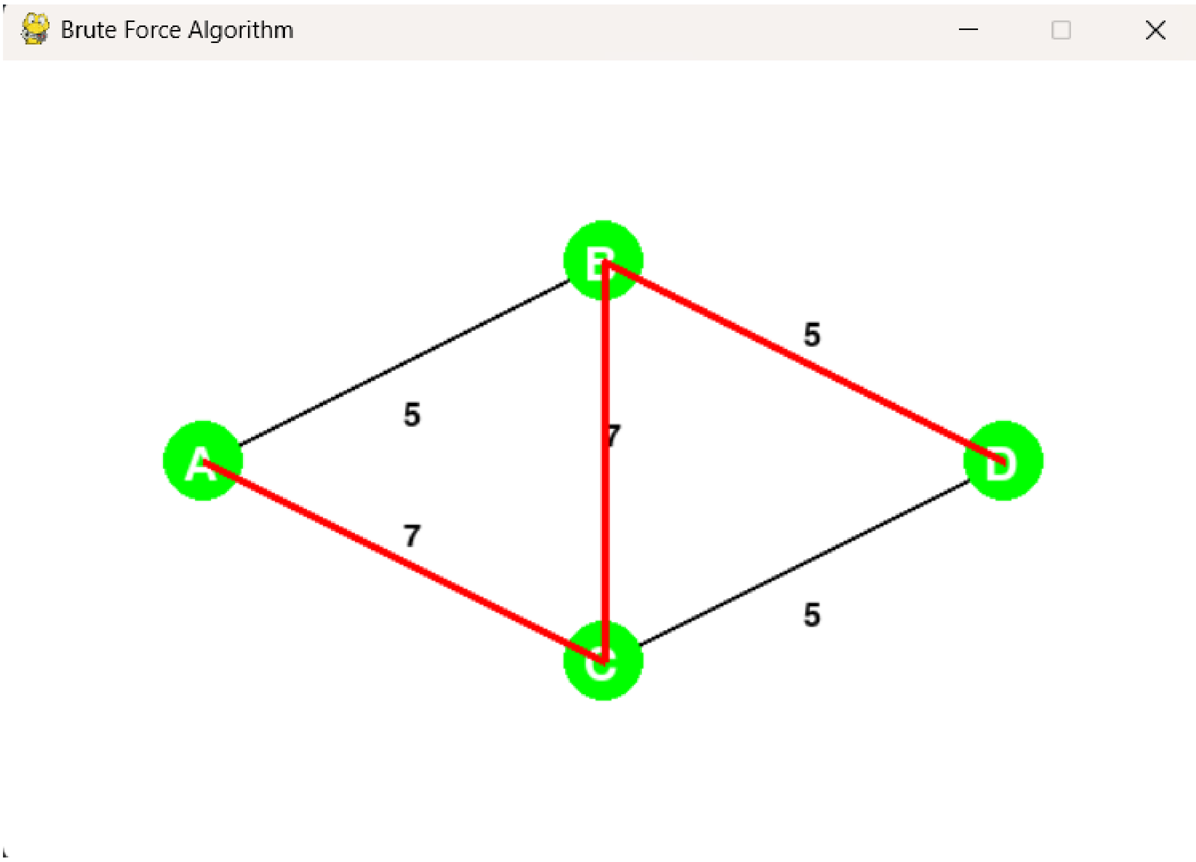
```
import pygame
import sys
from itertools import permutations

# Initialize Pygame
pygame.init()

# Screen dimensions
WIDTH, HEIGHT = 600, 400
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Brute Force Algorithm")

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

# Graph representation
graph = {
    'A': {'B': 2, 'C': 7},
    'B': {'A': 4, 'C': 7, 'D': 5},
```

```
    'C': {'A': 4, 'B': 2, 'D': 5},
    'D': {'B': 2, 'C': 7}
}

# Update edge weights (example: update A-B to 5)
graph['A']['B'] = 5
graph['B']['A'] = 5

# Node positions for visualization
node_positions = {
    'A': (100, 200),
    'B': (300, 100),
    'C': (300, 300),
    'D': (500, 200)
}

# Brute Force Algorithm
def brute_force_pathfinding(graph, start, end):
    nodes = list(graph.keys())
    nodes.remove(start)
    nodes.remove(end)
    all_paths = permutations(nodes)

    shortest_path = None
    min_cost = float('inf')

    for path in all_paths:
        current_path = [start] + list(path) + [end]
        current_cost = 0
        for i in range(len(current_path) - 1):
            current_cost += graph[current_path[i]][current_path[i + 1]]

        if current_cost < min_cost:
            min_cost = current_cost
            shortest_path = current_path

    return shortest_path, min_cost

# Find the shortest path
start, end = 'A', 'D'
shortest_path, cost = brute_force_pathfinding(graph, start, end)
print(f"Shortest Path: {shortest_path}, Cost: {cost}")

# Pygame visualization
```

```
def draw_graph():
    screen.fill(WHITE)
    # Track labeled edges to avoid duplicates
    labeled_edges = set()
    # Draw all edges and add numbers
    for node, edges in graph.items():
        for neighbor, weight in edges.items():
            # Skip if the edge has already been labeled
            if (node, neighbor) in labeled_edges or (neighbor, node) in labeled_edges:
                continue
            # Draw the edge
            pygame.draw.line(screen, BLACK, node_positions[node], node_positions[neighbor], 2)
            # Calculate midpoint for the number
            mid_point = (
                (node_positions[node][0] + node_positions[neighbor][0]) // 2,
                (node_positions[node][1] + node_positions[neighbor][1]) // 2
            )
            # Offset the number to avoid overlap
            if node_positions[node][1] < node_positions[neighbor][1]:
                # Edge goes downward, place number above
                offset_x, offset_y = 0, -20
            else:
                # Edge goes upward, place number below
                offset_x, offset_y = 0, 20
            number_pos = (mid_point[0] + offset_x, mid_point[1] + offset_y)
            # Draw the weight number
            font = pygame.font.Font(None, 24)
            text = font.render(str(weight), True, BLACK)
            screen.blit(text, number_pos)
            # Mark the edge as labeled
            labeled_edges.add((node, neighbor))
    # Draw all nodes
    for node, pos in node_positions.items():
        pygame.draw.circle(screen, GREEN, pos, 20)
        font = pygame.font.Font(None, 36)
        text = font.render(node, True, WHITE)
        screen.blit(text, (pos[0] - 10, pos[1] - 10))
    # Highlight shortest path
    for i in range(len(shortest_path) - 1):
        pygame.draw.line(screen, RED, node_positions[shortest_path[i]],
node_positions[shortest_path[i + 1]], 4)
    pygame.display.flip()

# Main loop
```

```
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    draw_graph()
pygame.quit()
sys.exit()
```

Visualisation:



Reflection:

- **Challenges:** The main challenge with brute force is its inefficiency. For a graph with n*n* nodes, the algorithm checks (n−2)!(*n*−2)! paths, which becomes impractical for large n*n*.
- **Strengths:** Brute force is easy to implement and guarantees the optimal solution, making it useful for small problems or as a benchmark.
- **Weaknesses:** It is not suitable for real-time games due to its poor scalability.
- **Comparison:** Compared to Dijkstra and A*, brute force is much slower and impractical for large-scale pathfinding.

2. Dijkstra's Algorithm (Algorithm)

Description of the Technique:

Dijkstra's algorithm is a widely used algorithm for finding the shortest path between two nodes in a graph. It works by iteratively exploring the closest nodes from the starting point and updating the shortest known distances to each node. It guarantees the shortest path in graphs with non-negative edge weights.

- **Role in Game AI:** Dijkstra's algorithm is used for pathfinding in games, especially when the shortest path is required and the graph is relatively small.
- **Strengths:** Guarantees the shortest path, works well for small to medium-sized graphs.
- **Weaknesses:** Inefficient for large graphs or real-time applications due to its O(n2)$O(n2)$ time complexity.

Implementation:

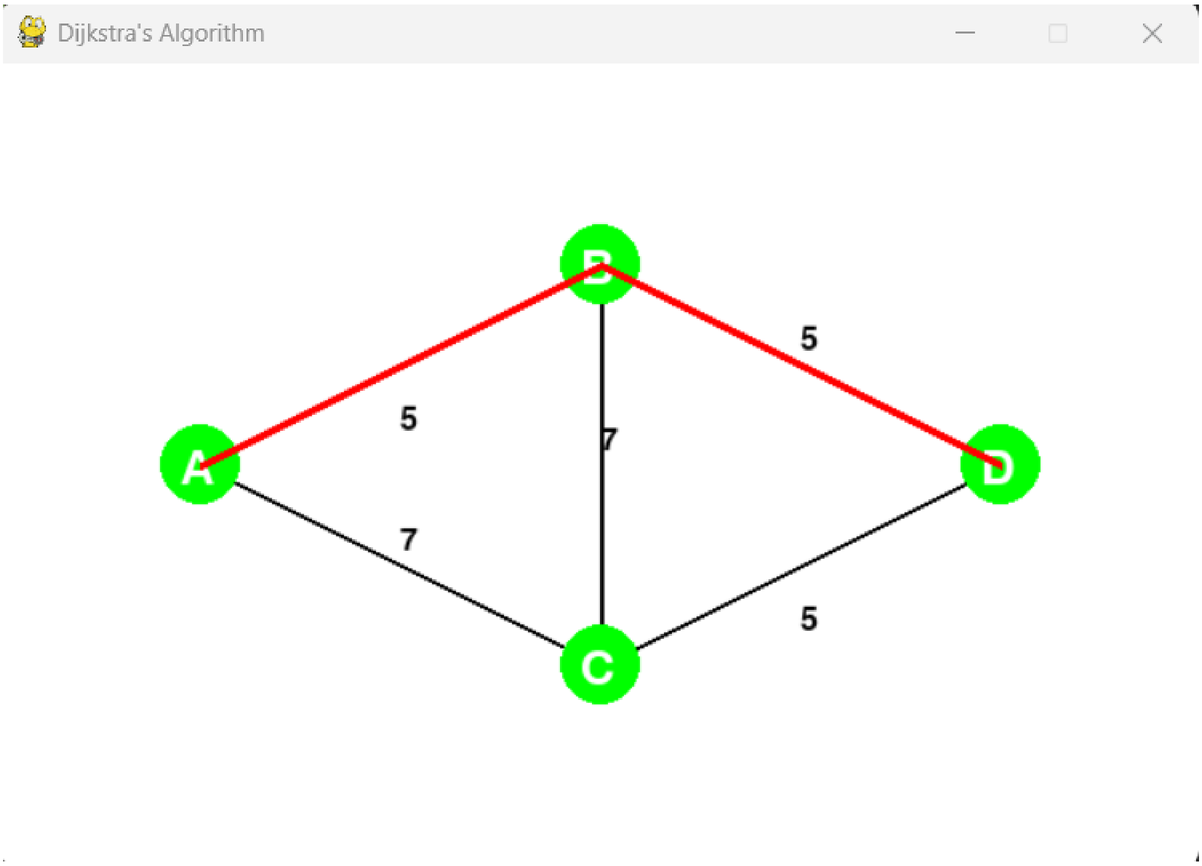
```
import pygame
import sys
import heapq

# Initialize Pygame
pygame.init()

# Screen dimensions
WIDTH, HEIGHT = 600, 400
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Dijkstra's Algorithm")

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

# Graph representation
graph = {
  'A': {'B': 2, 'C': 7},
  'B': {'A': 4, 'C': 7, 'D': 5},
  'C': {'A': 4, 'B': 2, 'D': 5},
  'D': {'B': 2, 'C': 7}
```

```
}

# Update edge weights (example: update A-B to 5)
graph['A']['B'] = 5
graph['B']['A'] = 5

# Node positions for visualization
node_positions = {
    'A': (100, 200),
    'B': (300, 100),
    'C': (300, 300),
    'D': (500, 200)
}

# Dijkstra's Algorithm
def dijkstra(graph, start, end):
    queue = [(0, start)]
    costs = {node: float('inf') for node in graph}
    costs[start] = 0
    path = {}

    while queue:
        current_cost, current_node = heapq.heappop(queue)

        if current_node == end:
            break

        for neighbor, weight in graph[current_node].items():
            new_cost = current_cost + weight
            if new_cost < costs[neighbor]:
                costs[neighbor] = new_cost
                heapq.heappush(queue, (new_cost, neighbor))
                path[neighbor] = current_node

    # Reconstruct the path
    shortest_path = []
    node = end
    while node != start:
        shortest_path.append(node)
        node = path[node]
    shortest_path.append(start)
    shortest_path.reverse()

    return shortest_path, costs[end]
```

```
# Find the shortest path
start, end = 'A', 'D'
shortest_path, cost = dijkstra(graph, start, end)
print(f"Shortest Path: {shortest_path}, Cost: {cost}")

# Pygame visualization
def draw_graph():
    screen.fill(WHITE)
    # Track labeled edges to avoid duplicates
    labeled_edges = set()
    # Draw all edges and add numbers
    for node, edges in graph.items():
        for neighbor, weight in edges.items():
            # Skip if the edge has already been labeled
            if (node, neighbor) in labeled_edges or (neighbor, node) in labeled_edges:
                continue
            # Draw the edge
            pygame.draw.line(screen, BLACK, node_positions[node], node_positions[neighbor], 2)
            # Calculate midpoint for the number
            mid_point = (
                (node_positions[node][0] + node_positions[neighbor][0]) // 2,
                (node_positions[node][1] + node_positions[neighbor][1]) // 2
            )
            # Offset the number to avoid overlap
            if node_positions[node][1] < node_positions[neighbor][1]:
                # Edge goes downward, place number above
                offset_x, offset_y = 0, -20
            else:
                # Edge goes upward, place number below
                offset_x, offset_y = 0, 20
            number_pos = (mid_point[0] + offset_x, mid_point[1] + offset_y)
            # Draw the weight number
            font = pygame.font.Font(None, 24)
            text = font.render(str(weight), True, BLACK)
            screen.blit(text, number_pos)
            # Mark the edge as labeled
            labeled_edges.add((node, neighbor))
    # Draw all nodes
    for node, pos in node_positions.items():
        pygame.draw.circle(screen, GREEN, pos, 20)
        font = pygame.font.Font(None, 36)
        text = font.render(node, True, WHITE)
        screen.blit(text, (pos[0] - 10, pos[1] - 10))
```

```
    # Highlight shortest path
    for i in range(len(shortest_path) - 1):
        pygame.draw.line(screen, RED, node_positions[shortest_path[i]],
node_positions[shortest_path[i + 1]], 4)
    pygame.display.flip()

# Main loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    draw_graph()
pygame.quit()
sys.exit()
```

Visualisation:



Reflection:

- **Challenges:** Dijkstra's algorithm can be slow for large graphs because it explores all nodes. Optimizing it for real-time games requires careful implementation.
- **Strengths:** It guarantees the shortest path and is relatively easy to implement.

- **Weaknesses:** It is inefficient for large graphs or real-time applications.
- **Comparison:** Compared to A*, Dijkstra explores more nodes because it does not use a heuristic to guide the search.

3. A* Algorithm (Heuristic)

Description of the Technique:

A* is an extension of Dijkstra's algorithm that uses a heuristic to guide the search. It combines the cost to reach a node (like Dijkstra) with an estimate of the cost to reach the goal (heuristic). This makes A* more efficient than Dijkstra for large graphs.

- **Role in Game AI:** A* is the standard algorithm for pathfinding in games because it is efficient and guarantees the shortest path when the heuristic is admissible.
- **Strengths:** Efficient, guarantees the shortest path with the right heuristic.
- **Weaknesses:** Performance depends on the quality of the heuristic.

Implementation:

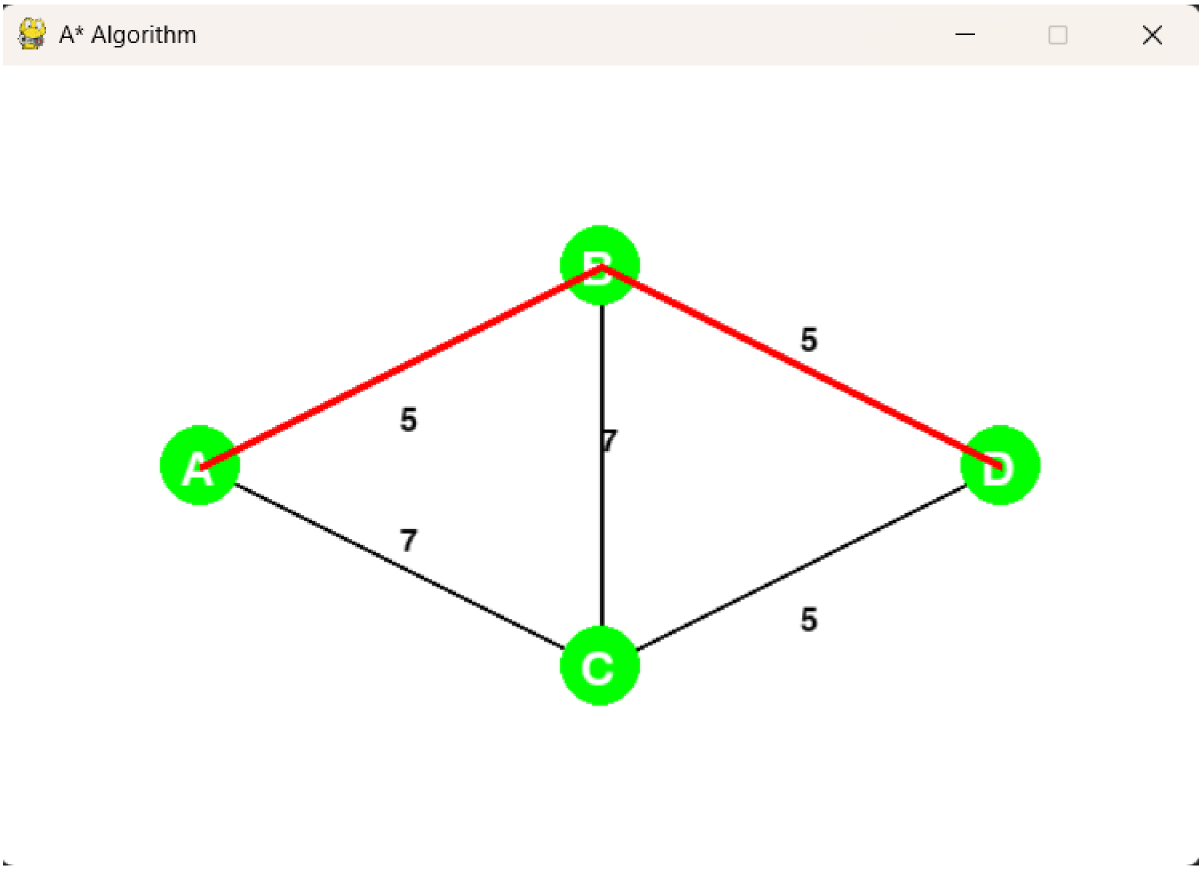
```
import pygame
import sys
import heapq

# Initialize Pygame
pygame.init()

# Screen dimensions
WIDTH, HEIGHT = 600, 400
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("A* Algorithm")

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

# Graph representation
graph = {
    'A': {'B': 2, 'C': 7},
    'B': {'A': 4, 'C': 7, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 5},
    'D': {'B': 2, 'C': 7}
}

# Update edge weights (example: update A-B to 5)
```

```
graph['A']['B'] = 5
graph['B']['A'] = 5

# Node positions for visualization
node_positions = {
    'A': (100, 200),
    'B': (300, 100),
    'C': (300, 300),
    'D': (500, 200)
}

# A* Algorithm
def heuristic(node, end):
    # Example heuristic: Manhattan distance
    return abs(ord(node) - ord(end))

def a_star(graph, start, end):
    queue = [(0, start)]
    costs = {node: float('inf') for node in graph}
    costs[start] = 0
    path = {}

    while queue:
        current_cost, current_node = heapq.heappop(queue)

        if current_node == end:
            break

        for neighbor, weight in graph[current_node].items():
            new_cost = current_cost + weight
            if new_cost < costs[neighbor]:
                costs[neighbor] = new_cost
                priority = new_cost + heuristic(neighbor, end)
                heapq.heappush(queue, (priority, neighbor))
                path[neighbor] = current_node

    # Reconstruct the path
    shortest_path = []
    node = end
    while node != start:
        shortest_path.append(node)
        node = path[node]
    shortest_path.append(start)
    shortest_path.reverse()
```

```
    return shortest_path, costs[end]

# Find the shortest path
start, end = 'A', 'D'
shortest_path, cost = a_star(graph, start, end)
print(f"Shortest Path: {shortest_path}, Cost: {cost}")

# Pygame visualization
def draw_graph():
    screen.fill(WHITE)
    # Track labeled edges to avoid duplicates
    labeled_edges = set()
    # Draw all edges and add numbers
    for node, edges in graph.items():
        for neighbor, weight in edges.items():
            # Skip if the edge has already been labeled
            if (node, neighbor) in labeled_edges or (neighbor, node) in labeled_edges:
                continue
            # Draw the edge
            pygame.draw.line(screen, BLACK, node_positions[node], node_positions[neighbor], 2)
            # Calculate midpoint for the number
            mid_point = (
                (node_positions[node][0] + node_positions[neighbor][0]) // 2,
                (node_positions[node][1] + node_positions[neighbor][1]) // 2
            )
            # Offset the number to avoid overlap
            if node_positions[node][1] < node_positions[neighbor][1]:
                # Edge goes downward, place number above
                offset_x, offset_y = 0, -20
            else:
                # Edge goes upward, place number below
                offset_x, offset_y = 0, 20
            number_pos = (mid_point[0] + offset_x, mid_point[1] + offset_y)
            # Draw the weight number
            font = pygame.font.Font(None, 24)
            text = font.render(str(weight), True, BLACK)
            screen.blit(text, number_pos)
            # Mark the edge as labeled
            labeled_edges.add((node, neighbor))
    # Draw all nodes
    for node, pos in node_positions.items():
        pygame.draw.circle(screen, GREEN, pos, 20)
        font = pygame.font.Font(None, 36)
```

```
        text = font.render(node, True, WHITE)
        screen.blit(text, (pos[0] - 10, pos[1] - 10))
    # Highlight shortest path
    for i in range(len(shortest_path) - 1):
        pygame.draw.line(screen, RED, node_positions[shortest_path[i]],
node_positions[shortest_path[i + 1]], 4)
    pygame.display.flip()

# Main loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    draw_graph()
pygame.quit()
sys.exit()
```

Visualisation:



Reflection:

- **Challenges:** Choosing the right heuristic is crucial. An overestimating heuristic can lead to suboptimal paths.
- **Strengths:** A* is highly efficient and guarantees the shortest path with an admissible heuristic.
- **Weaknesses:** Performance depends on the heuristic, and it can be slower than Dijkstra if the heuristic is poorly chosen.
- **Comparison:** A* is faster than Dijkstra because it uses a heuristic to guide the search, but it requires more memory to store the heuristic values.

Conclusion:

- **Brute Force:** Simple but impractical for large problems.
- **Dijkstra:** Guarantees the shortest path but is inefficient for large graphs.
- **A*:** Efficient and guarantees the shortest path with the right heuristic, making it the best choice for real-time games.