INTERNATIONAL BUSINESS ACADEMY

TOP-UP BACHELOR OF SCIENCE (HONOURS) INFORMATICS

COURSE NAME : TOP-UP BACHELOR OF SCIENCE (HONOURS) INFORMATICS

KOL304CR- GAMES AND AI

MODULE LEADER: LENA ERBS

PREPARED BY: BIPLAB TWATI

WORD COUNT: 1879

SUBMISSION DATE: 25/03/2025

GITHUB LINK : *https://github.com/BindashCharlie/CW1-GAMES-AND-AI*

Contents

1. Brute Force Algorithm

Description of the Technique:

Brute force is a very literal approach to problem solving, which tries every possibility until the best one is found. In the case of game AI, brute force can be used in pathfinding, decision-making and solving of puzzles. It is simple but very ineffective for large scale problems because of the exponential time complexity.

- **Application in the field of Game AI**: Not applied in real time games because of its inefficiency but it is useful in small projects or as a control measure.
- **Advantages**: Easy to incorporate, finds the best output.
- **Disadvantages**: Very slow for big data sets, unfit for real time use.

Implementation:

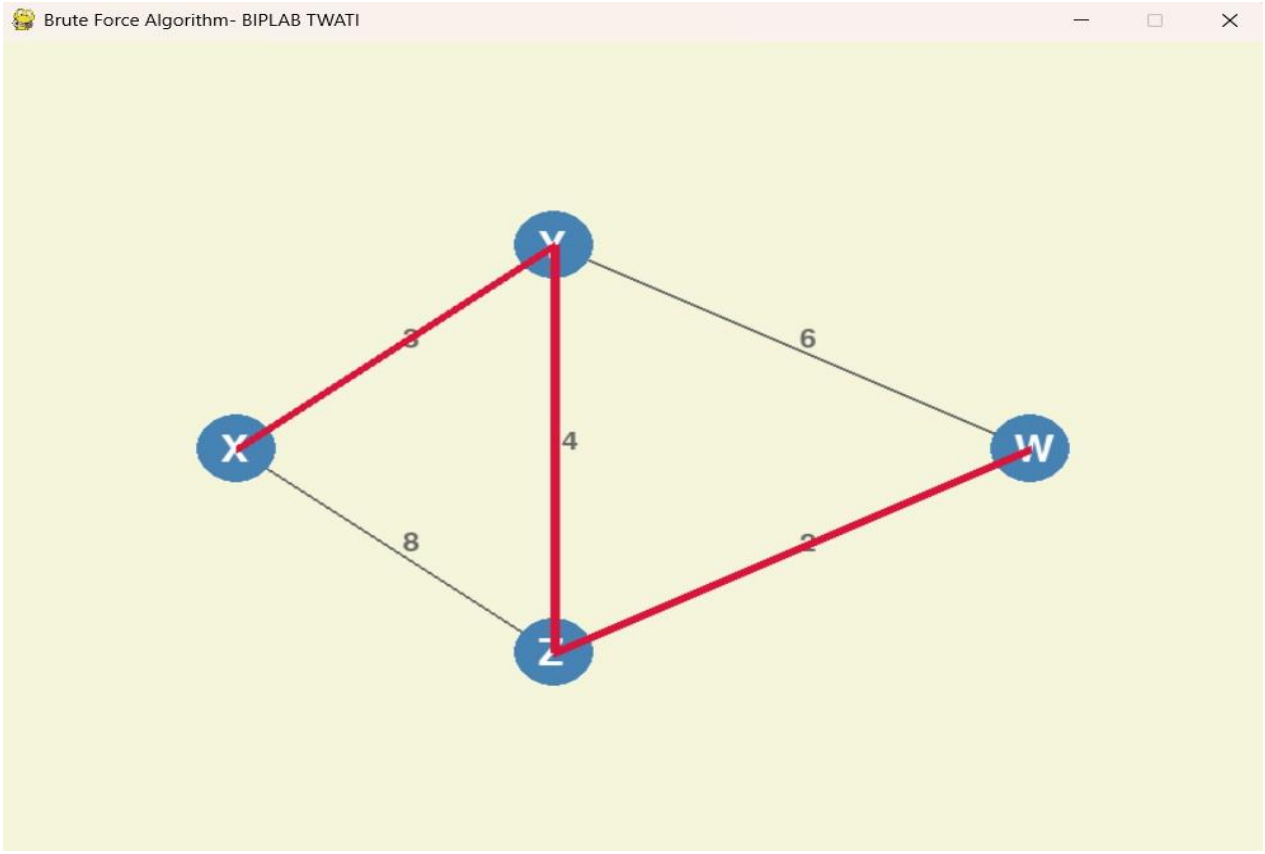
```
import pygame
import sys
from itertools import permutations

# Initialize Pygame
pygame.init()

# Screen dimensions
WIDTH, HEIGHT = 800, 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Brute Force Algorithm- BIPLAB TWATI")

# Color scheme
BACKGROUND = (245, 245, 220)
NODE_COLOR = (70, 130, 180)
EDGE_COLOR = (105, 105, 105)
PATH_COLOR = (220, 20, 60)
```

```
# Graph configuration
graph = {
    'X': {'Y': 3, 'Z': 8},
    'Y': {'X': 2, 'Z': 4, 'W': 6},
    'Z': {'X': 5, 'Y': 3, 'W': 2},
    'W': {'Y': 4, 'Z': 5}
}

# Node coordinates for visualization
node_coords = {
    'X': (150, 300),
    'Y': (350, 150),
    'Z': (350, 450),
    'W': (650, 300)
}

def brute_force_search(graph, start, target):
    nodes = list(graph.keys())
    nodes.remove(start)
    nodes.remove(target)

    shortest_path = None
    min_cost = float('inf')

    for permutation in permutations(nodes):
        current_path = [start] + list(permutation) + [target]
        total_cost = 0

        try:
            for i in range(len(current_path)-1):
                total_cost += graph[current_path[i]][current_path[i+1]]
        except KeyError:
            continue
```

```
        if total_cost < min_cost:
            min_cost = total_cost
            shortest_path = current_path

    return shortest_path, min_cost

def draw_interface():
    screen.fill(BACKGROUND)

    # Draw edges with weights
    drawn_edges = set()
    for node, connections in graph.items():
        for neighbor, weight in connections.items():
            if (node, neighbor) not in drawn_edges and (neighbor, node) not in drawn_edges:
                pygame.draw.line(screen, EDGE_COLOR,
                                 node_coords[node], node_coords[neighbor], 2)

                # Calculate weight label position
                mid_x = (node_coords[node][0] + node_coords[neighbor][0]) // 2
                mid_y = (node_coords[node][1] + node_coords[neighbor][1]) // 2
                font = pygame.font.Font(None, 28)
                text = font.render(str(weight), True, EDGE_COLOR)
                screen.blit(text, (mid_x + 5, mid_y - 15))

                drawn_edges.add((node, neighbor))

    # Draw nodes
    for node, pos in node_coords.items():
        pygame.draw.circle(screen, NODE_COLOR, pos, 25)
        label = pygame.font.Font(None, 40).render(node, True, (255, 255, 255))
        screen.blit(label, (pos[0]-10, pos[1]-12))
```

```
    # Highlight optimal path
    if shortest_path:
        for i in range(len(shortest_path)-1):
            start = node_coords[shortest_path[i]]
            end = node_coords[shortest_path[i+1]]
            pygame.draw.line(screen, PATH_COLOR, start, end, 6)

    pygame.display.flip()

# Calculate path
start_node = 'X'
end_node = 'W'
shortest_path, total_cost = brute_force_search(graph, start_node, end_node)
print(f"Optimal Path: {' → '.join(shortest_path)}")
print(f"Total Cost: {total_cost}")

# Visualization loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    draw_interface()

pygame.quit()
sys.exit()
```

Visualization:



Reflection:

- **Challenges**: Takes long for large datasets due to time complexity being factorial or exponential and requires much CPU and memory resources.
- **Strengths**: Easy to understand and implement and does not require complex data structures or heuristics.
- **Weaknesses**: Extremely slow for large inputs and performs poorly as input size increases.
- **Comparison**: Compared to Dijkstra and A*, brute force is highly inefficient and unsuitable for complex pathfinding due to its exhaustive search approach.

2. Dijkstra's Algorithm

Description of the Technique:

Dijkstra's Algorithm is a graph-based search algorithm to find the shortest path from an initial node to all other nodes in a weighted graph. It search recursively in the least-cost paths first and ensures optimal solutions. It is inefficient though, for big graphs without optimization techniques like a priority queue.

- **Application in the field of Game AI**: Used in pathfinding for strategy and simulation games where precise shortest paths are required, such as NPC navigation and route planning.
- **Advantages**: Guarantees the shortest path, works well with weighted graphs, and is more efficient than brute force.
- **Disadvantages**: Can be slow for large maps, does not perform well with dynamic environments without modifications.

Implementation:

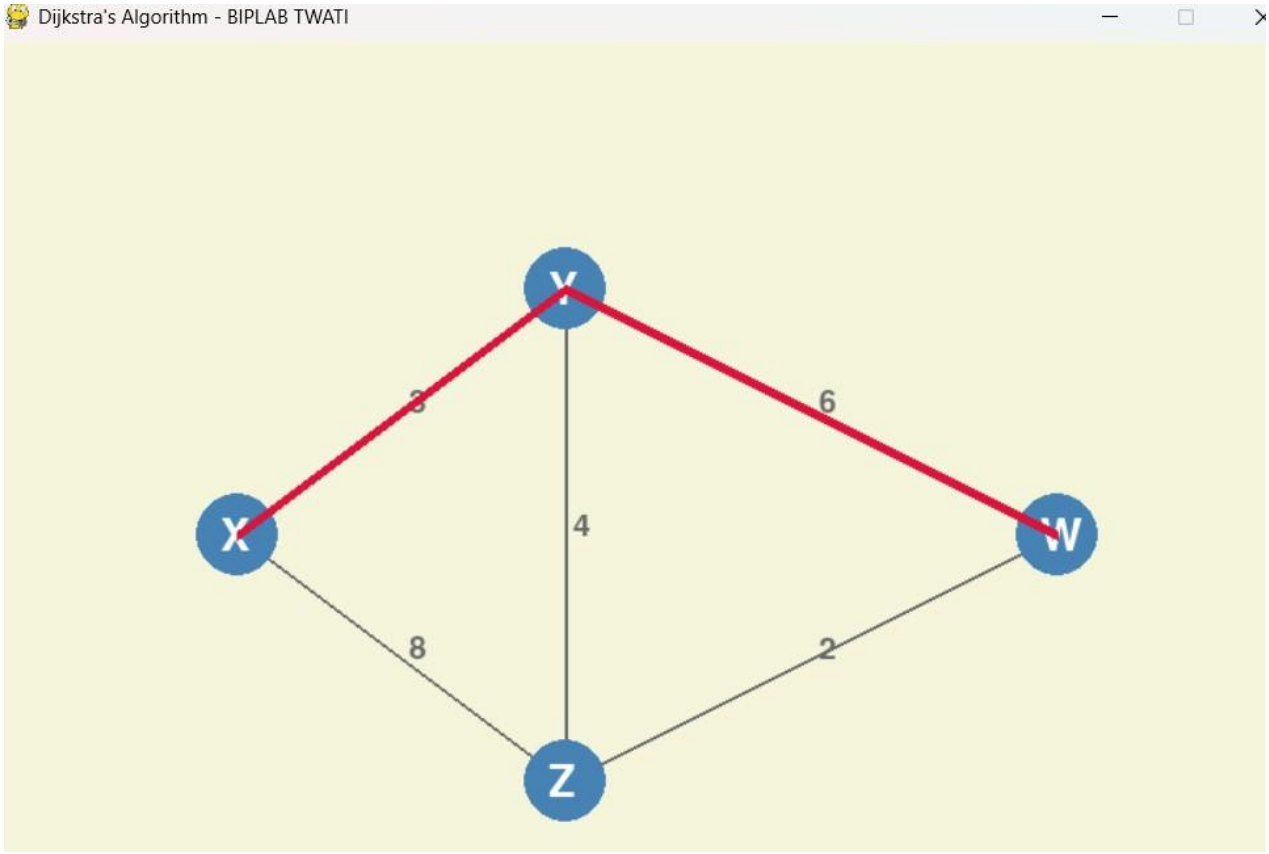
```
import pygame
import sys
import heapq

# Initialize Pygame
pygame.init()

# Screen dimensions
WIDTH, HEIGHT = 800, 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Dijkstra's Algorithm - BIPLAB TWATI")

# Color scheme
BACKGROUND = (245, 245, 220)
NODE_COLOR = (70, 130, 180)
```

```
EDGE_COLOR = (105, 105, 105)
PATH_COLOR = (220, 20, 60)

# Graph configuration
graph = {
    'X': {'Y': 3, 'Z': 8},
    'Y': {'X': 2, 'Z': 4, 'W': 6},
    'Z': {'X': 5, 'Y': 3, 'W': 2},
    'W': {'Y': 4, 'Z': 5}
}
# Node coordinates for visualization
node_coords = {
    'X': (150, 300),
    'Y': (350, 150),
    'Z': (350, 450),
    'W': (650, 300)
}

# Dijkstra's Algorithm
def dijkstra(graph, start, target):
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))  # (cost, node)
    came_from = {}
    shortest_distances = {node: float('inf') for node in graph}
    shortest_distances[start] = 0

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_node == target:
            path = []
            while current_node in came_from:
                path.append(current_node)
```

```
            current_node = came_from[current_node]
        path.append(start)
        path.reverse()
        return path, shortest_distances[target]

    for neighbor, weight in graph[current_node].items():
        distance = current_distance + weight
        if distance < shortest_distances[neighbor]:
            shortest_distances[neighbor] = distance
            came_from[neighbor] = current_node
            heapq.heappush(priority_queue, (distance, neighbor))

  return None, float('inf')

def draw_interface():
  screen.fill(BACKGROUND)
  # Draw edges with weights
  drawn_edges = set()
  for node, connections in graph.items():
    for neighbor, weight in connections.items():
      if (node, neighbor) not in drawn_edges and (neighbor, node) not in drawn_edges:
        pygame.draw.line(screen, EDGE_COLOR,
                  node_coords[node], node_coords[neighbor], 2)

        # Calculate weight label position
        mid_x = (node_coords[node][0] + node_coords[neighbor][0]) // 2
        mid_y = (node_coords[node][1] + node_coords[neighbor][1]) // 2
        font = pygame.font.Font(None, 28)
        text = font.render(str(weight), True, EDGE_COLOR)
        screen.blit(text, (mid_x + 5, mid_y - 15))

        drawn_edges.add((node, neighbor))
```

```
    # Draw nodes
    for node, pos in node_coords.items():
        pygame.draw.circle(screen, NODE_COLOR, pos, 25)
        label = pygame.font.Font(None, 40).render(node, True, (255, 255, 255))
        screen.blit(label, (pos[0]-10, pos[1]-12))

    # Highlight optimal path
    if shortest_path:
        for i in range(len(shortest_path)-1):
            start = node_coords[shortest_path[i]]
            end = node_coords[shortest_path[i+1]]
            pygame.draw.line(screen, PATH_COLOR, start, end, 6)

    pygame.display.flip()

# Calculate path
start_node = 'X'
end_node = 'W'
shortest_path, total_cost = dijkstra(graph, start_node, end_node)
print(f"Optimal Path: {' → '.join(shortest_path)}")
print(f"Total Cost: {total_cost}")

# Visualization loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    draw_interface()

pygame.quit()
sys.exit()
```

Visualization:



Reflection:

- **Challenges**: Slows down for very large graphs, especially without optimizations like priority queues, and not efficient at handling dynamic path changes .
- **Strengths**: Guarantees shortest path, handles weighted graphs well, and more efficient than brute force.
- **Weaknesses**: Runs out on large maps, and less efficient than A* because there is no heuristic guidance.
- **Comparison**: Dijkstra is far more efficient than brute force, but A* is even quicker with the application of heuristics to choose promising paths.

3. A* Algorithm (Heuristic)

Description of the Technique:

A* is an informed search algorithm used for pathfinding and graph traversal. It combines the cost-so-far (like Dijkstra) with a heuristic function to estimate the remaining cost, allowing it to prioritize the most promising paths. This makes it more efficient than Dijkstra's Algorithm, though performance depends on the heuristic used.

- **Application in the field of Game AI**: Widely used in real-time strategy and simulation games for NPC navigation, enemy movement, and dynamic pathfinding in open-world environments.
- **Advantages**: Faster than Dijkstra due to heuristic optimization, finds the shortest path efficiently, and balances exploration with cost estimation.
- **Disadvantages**: Requires a good heuristic for optimal performance, may use more memory than Dijkstra, and can be inefficient if the heuristic is poorly chosen.

Implementation:

```
import pygame
import sys
import heapq

# Initialize Pygame
pygame.init()

# Screen dimensions
WIDTH, HEIGHT = 800, 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("A* Algorithm - BIPLAB TWATI")
```

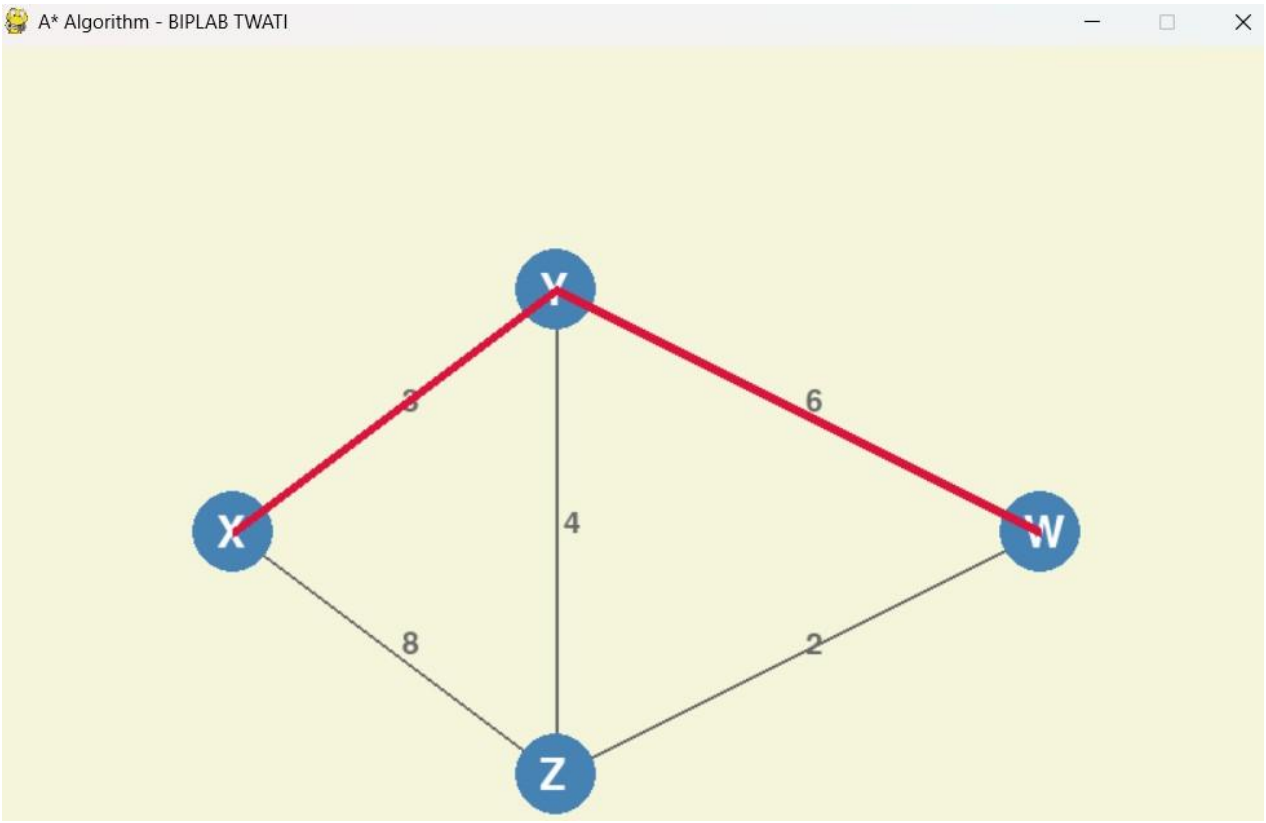
```
# Color scheme
BACKGROUND = (245, 245, 220)
NODE_COLOR = (70, 130, 180)
EDGE_COLOR = (105, 105, 105)
PATH_COLOR = (220, 20, 60)

# Graph configuration
graph = {
    'X': {'Y': 3, 'Z': 8},
    'Y': {'X': 2, 'Z': 4, 'W': 6},
    'Z': {'X': 5, 'Y': 3, 'W': 2},
    'W': {'Y': 4, 'Z': 5}
}

# Node coordinates for visualization
node_coords = {
    'X': (150, 300),
    'Y': (350, 150),
    'Z': (350, 450),
    'W': (650, 300)
}

# Heuristic function: Euclidean distance
def heuristic(node1, node2):
    x1, y1 = node_coords[node1]
    x2, y2 = node_coords[node2]
    return ((x2 - x1)**2 + (y2 - y1)**2) ** 0.5  # Euclidean distance

# A* algorithm
def a_star_search(graph, start, goal):
    open_set = []
    heapq.heappush(open_set, (0, start))  # (priority, node)
```

```
    came_from = {}
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0
    f_score = {node: float('inf') for node in graph}
    f_score[start] = heuristic(start, goal)

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            path.reverse()
            return path, g_score[goal]

        for neighbor, weight in graph[current].items():
            tentative_g_score = g_score[current] + weight

            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)
                heapq.heappush(open_set, (f_score[neighbor], neighbor))

    return None, float('inf')

def draw_interface():
    screen.fill(BACKGROUND)

    # Draw edges with weights
```

```
    drawn_edges = set()
    for node, connections in graph.items():
        for neighbor, weight in connections.items():
            if (node, neighbor) not in drawn_edges and (neighbor, node) not in drawn_edges:
                pygame.draw.line(screen, EDGE_COLOR,
                                 node_coords[node], node_coords[neighbor], 2)

                # Calculate weight label position
                mid_x = (node_coords[node][0] + node_coords[neighbor][0]) // 2
                mid_y = (node_coords[node][1] + node_coords[neighbor][1]) // 2
                font = pygame.font.Font(None, 28)
                text = font.render(str(weight), True, EDGE_COLOR)
                screen.blit(text, (mid_x + 5, mid_y - 15))

                drawn_edges.add((node, neighbor))

    # Draw nodes
    for node, pos in node_coords.items():
        pygame.draw.circle(screen, NODE_COLOR, pos, 25)
        label = pygame.font.Font(None, 40).render(node, True, (255, 255, 255))
        screen.blit(label, (pos[0]-10, pos[1]-12))

    # Highlight optimal path
    if shortest_path:
        for i in range(len(shortest_path)-1):
            start = node_coords[shortest_path[i]]
            end = node_coords[shortest_path[i+1]]
            pygame.draw.line(screen, PATH_COLOR, start, end, 6)

    pygame.display.flip()

# Calculate path
start_node = 'X'
```

```
end_node = 'W'
shortest_path, total_cost = a_star_search(graph, start_node, end_node)
print(f"Optimal Path: {' → '.join(shortest_path)}")
print(f"Total Cost: {total_cost}")

# Visualization loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    draw_interface()

pygame.quit()
sys.exit()
```

Visualization:



Reflection:

- **Challenges**: Requires a well-designed heuristic for optimal performance, can be computationally expensive in large or complex maps, and may consume more memory than Dijkstra's Algorithm.
- **Strengths**: Faster than Dijkstra due to heuristic-based optimization, guarantees the shortest path if an admissible heuristic is used, and efficiently balances exploration and path cost.
- **Weaknesses**: Performance depends on the quality of the heuristic, may become inefficient with a poorly chosen heuristic, and higher memory usage compared to Dijkstra.
- **Comparison**: Compared to brute force, A* is vastly more efficient, and compared to Dijkstra, it is faster by intelligently prioritizing paths using heuristics.

Conclusion:

- **Brute Force**: An easy but extremely inefficient technique that guarantees the best solution by checking all options, making it unusable for large-scale problems.
- **Dijkstra:** A decent algorithm for shortest path that works on weighted graphs but can be slow on big maps since its exhaustive nature. Best applied in projects where speed for accuracy is more important.
- **A* Algorithm**: A highly efficient pathfinding algorithm that is an enhancement of Dijkstra leveraging heuristics to guide the search. When paired with a good heuristic, it reduces computation time significantly, thus being appropriate for real-time usage such as game AI.