Algorithms Project : Shortest Paths

Members names	Students IDs
Remas abo khalil	444003748
Ruba Al-Saedi	444001785
Rimas Al-qurashi	444014714
Hoor Almarbaie	444002163
Jumanah Alruways	444000736
Remas Al-Rhaili	444005290
Refal ALnami	444000628
Lama Salawati	443002598
Retaj Mujahed	443007147
Murooj Ibrahim	444007269
Jana yahia	444010343
Reval Khafaji	444006814

UMM AL-QURA UNIVERSITY جامعة أم القرى

Task table:

Name	Tasks
رتاج اسامه مجاهد لمى خالد صلواتي	Randomized algorithm
ريفال خفاجي جنى سعيد	Dynamic programming Algorithms
ريماس الرحيلي ريماس القرشي	Divide and conquer Algorithms
ريماس ابوخليل ريفال النامي	Heuristic Algorithms
مروج إبراهيم ربى الصاعدي	Parallel Algorithms
جمانه الرويس حور المربعي	Greedy Algorithms

Shortest path

Introduction:
Shortest path algorithms are crucial tools in computer science, designed to determine the shortest path between specific points within a complex network. These algorithms are indispensable for numerous applications that need to identify efficient and optimal routes between different nodes.

Challenges:

One of the main challenges for Shortest path algorithms is execution time, particularly in large and intricate network environments. Some algorithms struggle when faced with frequent network changes or high memory demands.

Importance:

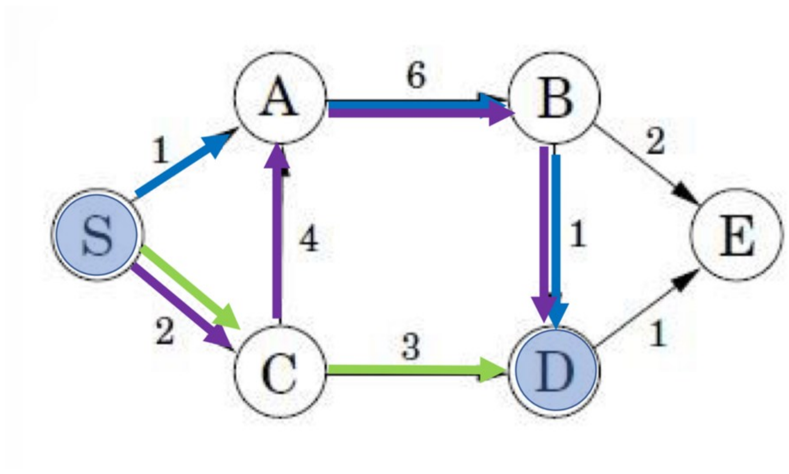
-Shortest path algorithms enable effective data and resource routing within computer networks.
- They play a key role in the development of navigation and road guidance apps, enhancing user experiences.
- Used across various domains, these algorithms improve logistics, manage traffic flow, enhance performance in electronic games, and more.

Applications:

- They guide aircraft, ships, and other transportation modes.
- Employed in planning and controlling the movement paths for robots and smart vehicles.
- Enhance user experience by offering guidance and navigation in smart applications.
- In electronic games, they refine player orientation and provide a more interactive gaming environment.

In summary, Shortest path algorithms are essential tools applied across a broad range of fields to compute effective and optimal paths within complex networks.

“The graph used for implementation and comparison between the algorithms."



Dynamic Programming

Dynamic programming is a powerful technique used to solve complex problems by breaking them down into simpler subproblems. In graph theory, dynamic programming can be applied to various scenarios such as finding the shortest path between two nodes or determining the minimum spanning tree.
In graphs, dynamic programming involves solving a problem for a smaller subgraph and building the solution iteratively until we reach the original problem. The key idea is to break down a complex problem into simpler subproblems, solve each subproblem only once, and store the results to avoid recomputation.

and we have 6 algorithms of dynamic programming: all-pairs shortest paths (Floyd-Warshall Algorithm), Dijkstra's Algorithm, Bellman-Ford Algorithm, Johnson's Algorithm, A A-Star Algorithm and (Breadth-First Search Algorithm).

We going to compare two of them:

1- Floyd-Warshall Algorithm.
2- Breadth-first search (BFS) Algorithm.

Floyd-Warshall Algorithm is one of the most commonly used algorithms for finding all-pairs shortest paths. Data Structure: It uses a 2D array.

Time Complexity:
The Floyd Warshall Algorithm has a time complexity of O(V3) and a space complexity of O(V2), where V represents the number of vertices in the graph.

Pros of Floyd-Warshall Algorithm:
Versatility: Calculates shortest paths between all pairs of nodes in directed or undirected graphs.
Simplicity: Easy to understand and implement, relying on dynamic programming.
Efficiency in Dense Graphs: Works efficiently in dense graphs with many edges, as it has a fixed time complexity per node pair.

Cons of Floyd-Warshall Algorithm:
Time Complexity: Operates with
$O(n$^3) time complexity, making it slow for large graphs.
Memory Usage: Requires an n×n matrix to store distances, increasing memory consumption.
Limited Suitability: Less effective for sparse graphs compared to other algorithms, like Dijkstra’s.

Code:
```
# Define a large number to represent infinity (no direct path)
INF = float('inf')
```

```
# Define the adjacency matrix for the graph
# S = 0, A = 1, B = 2, C = 3, D = 4, E = 5
graph = [
    [0, 1, INF, 2, INF, INF],  # S
    [INF, 0, 6, INF, INF, INF], # A
    [INF, INF, 0, INF, 1, 2],   # B
    [INF, 4, 3, 0, 3, INF],     # C
    [INF, INF, INF, INF, 0, INF], # D
    [INF, INF, INF, INF, INF, 0] # E    ]
# Number of vertices
n = len(graph)
# Implementing the Floyd-Warshall algorithm
def floyd_warshall(graph):
    # Initialize the distance matrix with graph values
    distance = [[graph[i][j] for j in range(n)] for i in range(n)]
    # Iterate over each possible intermediate vertex
    for k in range(n):
        for i in range(n):
            for j in range(n):
                # Update the distance if a shorter path is found
                if distance[i][j] > distance[i][k] + distance[k][j]:
                    distance[i][j] = distance[i][k] + distance[k][j]
    # Print the resulting shortest path matrix
    print("Shortest distances between every pair of vertices:")
    for i in range(n):
        for j in range(n):
            if distance[i][j] == INF:
                print("INF", end=" ")
            else:
                print(distance[i][j], end=" ")
        print()
floyd_warshall(graph)
```

Breadth-first search (BFS) algorithm is a great shortest path algorithm for all graphs, the path found by breadth first search to any node is the shortest path to that node, i.e the path that contains the smallest number of edges in unweighted graphs.
Date Structure: Use a queue to store a pair of values {node, distance}

Time Complexity:
Auxiliary Space: O(N) for storing distance for each node.

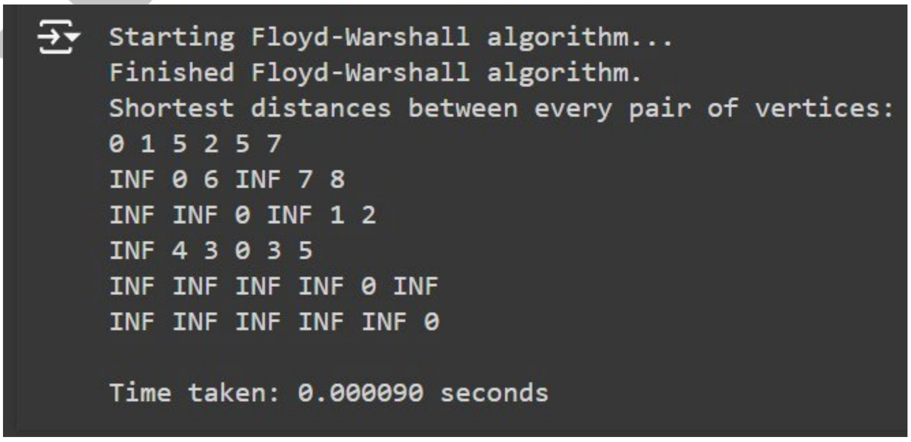
Pros of BFS (Breadth-First Search):
Simplicity: Easy to understand and implement using a queue data structure.
Guarantees Shortest Path: Ensures finding the shortest path in unweighted graphs.
Efficiency: Operates with a time complexity of O(V+E).

Cons of BFS (Breadth-First Search):
High Memory Consumption: Can require significant memory, especially in dense or deep graphs.
Not Suitable for Weighted Edges: Cannot find the shortest path in graphs with weighted edges, as it does not consider edge weights.
Only Works for Fixed Distances: Suitable only for cases where all edges are equal in weight.

Code:
```
# Define the adjacency list for the graph
# S = 0, A = 1, B = 2, C = 3, D = 4, E = 5
```

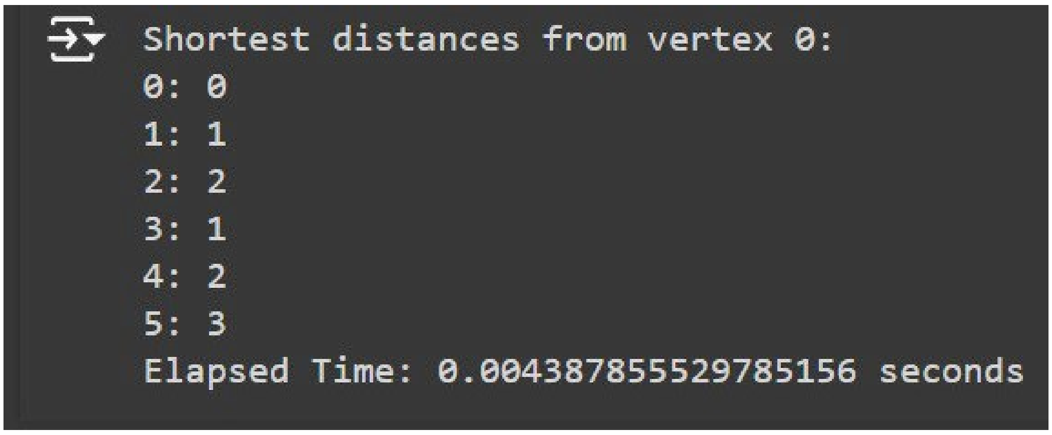
```
graph = {
   0: [1, 3],    # S -> A, S -> C
   1: [2],       # A -> B
   2: [4, 5],    # B -> D, B -> E
   3: [1, 2, 4], # C -> A, C -> B, C -> D
   4: [],        # D has no outgoing edges
   5: []         # E has no outgoing edges          }
# BFS function to find shortest paths from the starting node
def bfs_shortest_paths(graph, start):
   # Initialize distances with infinity and set the distance to the
start node to 0
   distances = {node: float('inf') for node in graph}
   distances[start] = 0
   # Queue to manage the BFS traversal
   queue = deque([start])
   while queue:
      current = queue.popleft()
      # Visit each neighbor
      for neighbor in graph[current]:
# If we haven't visited this neighbor, update the distance and
add to queue
         if distances[neighbor] == float('inf'):
            distances[neighbor] = distances[current] + 1
            queue.append(neighbor)
   # Print the shortest distances from start to each node
   print(f"Shortest distances from vertex {start}:")
   for vertex, distance in distances.items():
      if distance == float('inf'):
         print(f"{vertex}: INF")
      else:
         print(f"{vertex}: {distance}")
bfs_shortest_paths(graph, 0)
```

the output and time of **Floyd-Warshall Algorithm**:



```
Starting Floyd-Warshall algorithm...
Finished Floyd-Warshall algorithm.
Shortest distances between every pair of vertices:
0 1 5 2 5 7
INF 0 6 INF 7 8
INF INF 0 INF 1 2
INF 4 3 0 3 5
INF INF INF INF 0 INF
INF INF INF INF INF 0

Time taken: 0.000090 seconds
```

the output and time of **BFS (Breadth-First Search)**:



```
Shortest distances from vertex 0:
0:  0
1:  1
2:  2
3:  1
4:  2
5:  3
Elapsed Time:  0.004387855529785156  seconds
```

This indicates that the Floyd-Warshall Algorithm is better than Breadth-first search Algorithm (BFS) in terms of time complexity.

Operation	Floyd-Warshall	Dijkstra	Bellman-Ford	Johnson’s	Breadth-first search (BFS)	A (A-Star)
Time complexity	O(V^3)	O((V+E)logV) With a priority queue	O(V*E)	O(V^2logV+V*E)	O(V+E)	O(E)

جامعة أم القرى
UMM AL-QURA UNIVERSITY

Parallel Algorithms

An algorithm is a set of instructions that receives input from users, performs computations, and generates an output. A parallel algorithm is a type of algorithm that can carry out multiple instructions concurrently across various processing units and then merge the individual results to derive the final outcome.

The Selected parallel Algorithms:

1. Dijkstra's Algorithm (Parallelized)
2. Bellman-Ford Algorithm (Parallelized)
3. Floyd-Warshall Algorithm
4. Genetic Algorithms

1 - Parallel Dijkstra's algorithm
Parallel Dijkstra's algorithm speeds up finding the shortest path in a graph from one source to all other vertices by distributing tasks across multiple processing units or threads for simultaneous computation and updating of distances, enhancing efficiency through parallel execution.

Positive effects:

- Faster Pathfinding: Divides the task among multiple processors for quicker solutions, especially for large graphs.

- Efficient Resource Use: Optimizes resource usage for better performance and lower energy consumption

Negative effects:

Complex Implementation: Requires advanced programming skills and techniques.

Higher Resource Demands: Needs more computational resources, increasing costs.

2 - Bellman-Ford Algorithm (Parallelized)
Parallel Bellman-Ford algorithm computes shortest paths from one source to all other vertices, handling graphs with negative edge weights by allowing multiple processors to work on different parts of the graph simultaneously. Each processor works on different parts of the graph, with synchronization points required after each iteration to ensure consistency.

Strengths: Can handle graphs with negative edge weights. Parall elization can improve performance for large graphs.

Weaknesses: More iterations compared to Dijkstra's algorithm, l eading to higher time complexity even when parallelized.

3 - Floyd-Warshall Algorithm

The Floyd-Warshall algorithm computes shortest paths between all pairs of vertices in a graph. Parallel execution allows each processor to compute a subset of distances, combining results to improve performance for dense graphs. This can be achieved by dividing the matrix among multiple processors, where each processor computes a subset of the distances, then combines results.

Strengths: Computes shortest paths between all pairs of vertices. Ideal for dense graphs and fully connected networks.

- Weaknesses: High time complexity (O(V^3)) and memory usage.

Parallelization helps but doesn't reduce the overall complexity significantly.

4 - Genetic Algorithms

Genetic algorithms solve the shortest path problem using evolutionary techniques. Populations evolve over generations, with parallel execution enabling different populations to evolve simultaneously. In parallel strategy, different populations can be evolved in parallel, with occasional migration of solutions between populations to enhance diversity and convergence.

- Strengths: Useful for complex optimization problems and can provide good solutions in reasonable time. High potential for parallelization due to population-based approach.

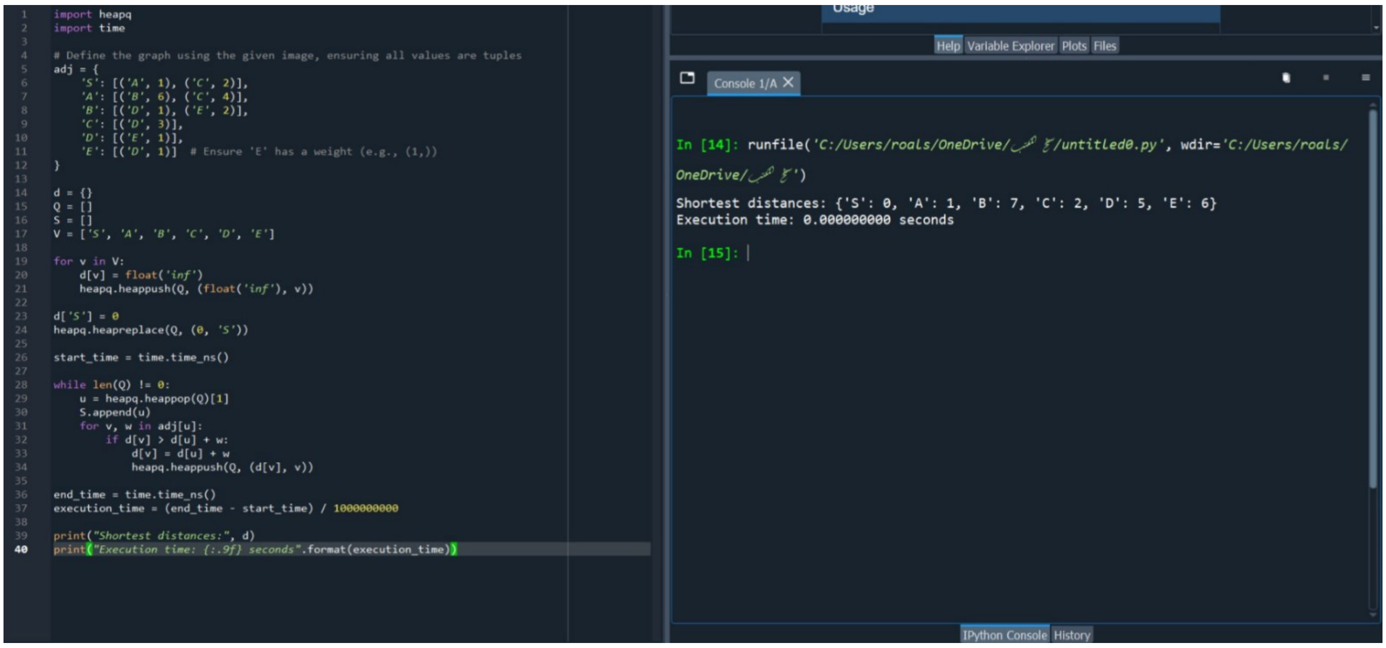
- Weaknesses: May not always find the optimal solution and requires tuning of parameters. Performance depends on the quality of the genetic operators.

Algorithm name	Time complexity
Parallel Dijkstra's algorithm	$O\left(\frac{V^2}{P} + V \cdot \log(P)\right)$
Bellman-Ford Algorithm	$O(V^2)$
Floyd-Warshall Algorithm	$O(V^3)$
Genetic Algorithms	O(G \cdot (nm + nm + n))

4. Best Algorithm

Among these, Dijkstra's Algorithm (Parallelized) is often considered the best for most applications due to its efficiency and Ford Algorithm (Parallelized) may be more appropriate.

5. Code for the Best Algorithm Here's a sample code snippet for parallelizing Dijkstra's Algorithm using Python’s



```python
import heapq
import time

# Define the graph using the given image, ensuring all values are tuples
adj = {
    'S': [('A', 1), ('C', 2)],
    'A': [('B', 6), ('C', 4)],
    'B': [('D', 1), ('E', 2)],
    'C': [('D', 3)],
    'D': [('E', 1)],
    'E': [('D', 1)]  # Ensure 'E' has a weight (e.g., (1,))
}

d = {}
Q = []
S = []
V = ['S', 'A', 'B', 'C', 'D', 'E']

for v in V:
    d[v] = float('inf')
    heapq.heappush(Q, (float('inf'), v))

d['S'] = 0
heapq.heapreplace(Q, (0, 'S'))

start_time = time.time_ns()

while len(Q) != 0:
    u = heapq.heappop(Q)[1]
    S.append(u)
    for v, w in adj[u]:
        if d[v] > d[u] + w:
            d[v] = d[u] + w
            heapq.heappush(Q, (d[v], v))

end_time = time.time_ns()
execution_time = (end_time - start_time) / 1000000000

print("Shortest distances:", d)
print("Execution time: {:.9f} seconds".format(execution_time))
```

```
In [14]: runfile('C:/Users/roals/OneDrive/سطح المكتب/untitled0.py', wdir='C:/Users/roals/OneDrive/سطح المكتب')

Shortest distances: {'S': 0, 'A': 1, 'B': 7, 'C': 2, 'D': 5, 'E': 6}
Execution time: 0.000000000 seconds

In [15]:
```

Heuristic Algorithms

The Heuristics strategy is a powerful tools that have been used to solve complex problems, it relay on experimentation and intuition rather than searching for optimal solutions, and these algorithms aim to arrive at suitable solutions in a reasonable time. These algorithms are widely used in various fields such as AI, CS, and data science

The Selected Heuristic Algorithms:

1- **Bidirectional Dijkstra :** This algorithm is used to determine the shortest path between two points in a directed or undirected graph, where the two searches meet at a common point, which will reduces the execution time.

Positive effects:

- it Speeds up pathfinding by searching only half of the graph.
- in large graphs, it reduce execution time significantly which will Boost Performance .

Negative effects:

- requires more memory because it maintains two search trees, which can be problematic in large graphs
- It is not effective for graphs that change frequently, as it needs to be restarted if edge weights change.

2- **A* algorithm**

This algorithm is used to find the shortest path from a starting point to a target within a network of nodes it provide efficiency in reaching the goal and avoiding unnecessary calculations, which will make it faster and more effective.

It work by exploring nodes based on an estimated cost to reach the target. It uses an evaluation function to determine which node is the best to explore next. This function relies on the cost of reaching the current node from the start, plus an estimate of the remaining distance to the goal.

Positive effects:

- Fast and efficient at finding the shortest path.
- Guarantees the optimal solution if the heuristics are accura te.
- Flexible and customizable for various problems.

Negative effects:

- High memory used.
- may be less effective if heuristics are inaccurate.
- Can be slow in complex graphs.

3- Bidirectional Search

It is a search algorithm that explores paths from the starting point and the destination at the same time. The idea is to meet in the middle, which effectively reduces the search space and search time compared to traditional one-way search methods (such as Dijkstra or A*).

How It Works:

1. Two searches are initiated simultaneously: one from the start node and one from the goal node.
2. Each search continues until the searches converge on a common node.
3. Once they meet, the algorithm can construct the shortest path by combining the two search paths.

Positive effects:

- Efficiency: By searching from both ends, it can significantly reduce the number of nodes explored, making it faster for large graphs.
- Optimal Path: Use appropriate heuristics to ensure that the shortest path is found.

Negative effects:

- Complexity: Implementing bidirectional search can be more complicated than traditional single-direction searches.

- Memory Usage: It might need additional memory because two search trees are managed at the same time.

- Less effective on weighted graphs: Its performance may be poor when there are varying weights on the edges.

4- Greedy Best-First Search

The Greedy Best First Search algorithm is a search algorithm for finding the shortest path in a graph. It works by estimating the remaining distance to the goal using a heuristic function, often called h(n). The algorithm selects the nodes with the lowest h value to check first. While this makes the algorithm faster, it may not always find the best path

Positive effects:

- **Speed**: It quickly explores nodes that are closer to the goal, making it efficient in many scenarios.

- **Simplicity**: The reliance on a single heuristic function simplifies understanding and implementation.

Negative effects:

- **No Optimal Solution Guarantee**: It may overlook important nodes in its quest for a fast solution.

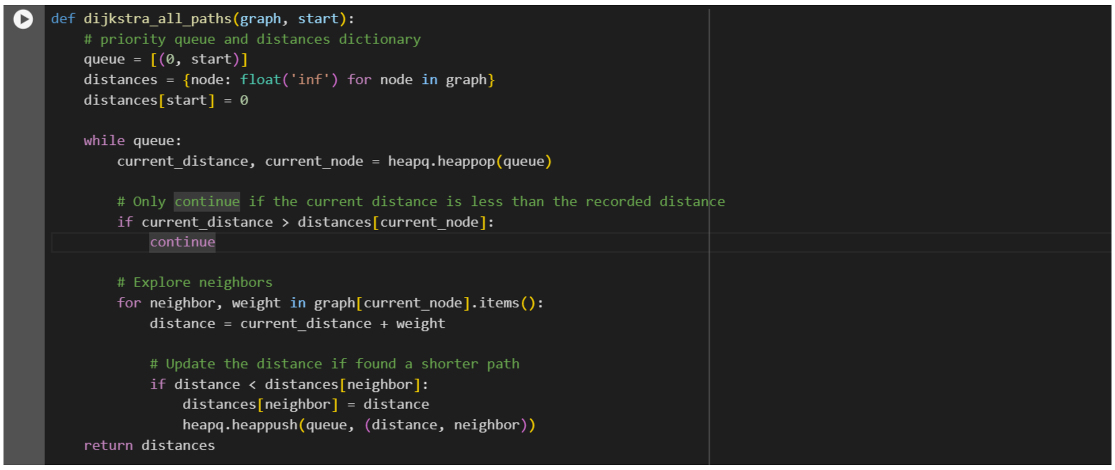
- **Heuristic Sensitivity**: The results heavily depend on the accuracy of the heuristic function used. If it's inaccurate, it can lead to suboptimal paths.

Algorithm	Time complexity
Bidirectional Dijkstra	O(b^d/2)
A* algorithm	O(b^d)
Bidirectional Search	**O(b^d/2**
Greedy Best-First Search	**O(b^d)**

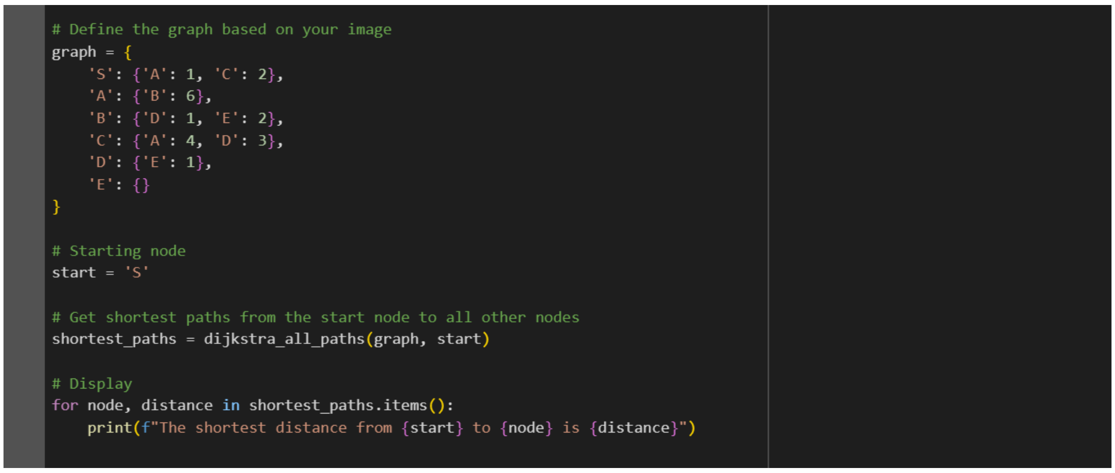
The best algorithm for shortest path problem I a weighted graph:
Bidirectional Dijkstra

It is specifically designed to handle graphs with non-negative weights, ensuring that the shortest path is always found. In contrast, Bidirectional Search may not be as effective in graphs with varying weights.

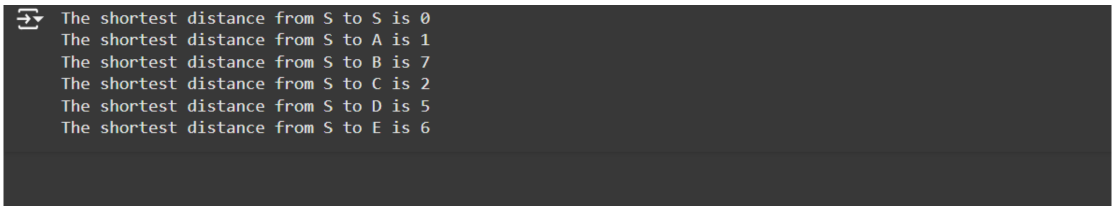
Bidirectional Dijkstra Algorithms Implementation part:



```python
def dijkstra_all_paths(graph, start):
    # priority queue and distances dictionary
    queue = [(0, start)]
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

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

        # Only continue if the current distance is less than the recorded distance
        if current_distance > distances[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            # Update the distance if found a shorter path
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))
    return distances
```



```python
# Define the graph based on your image
graph = {
    'S': {'A': 1, 'C': 2},
    'A': {'B': 6},
    'B': {'D': 1, 'E': 2},
    'C': {'A': 4, 'D': 3},
    'D': {'E': 1},
    'E': {}
}

# Starting node
start = 'S'

# Get shortest paths from the start node to all other nodes
shortest_paths = dijkstra_all_paths(graph, start)

# Display
for node, distance in shortest_paths.items():
    print(f"The shortest distance from {start} to {node} is {distance}")
```



```
The shortest distance from S to S is 0
The shortest distance from S to A is 1
The shortest distance from S to B is 7
The shortest distance from S to C is 2
The shortest distance from S to D is 5
The shortest distance from S to E is 6
```

Greedy Algorithms

Greedy Strategies in Shortest Path Algorithms:
Greedy algorithms make a sequence of decisions, each one appearing to be the most favorable at that time. In shortest path algorithms, these approaches aim to extend the most promising route according to specific criteria, such as the shortest distance at the current step.

Dijkstra's Algorithm Advantages:
Efficient for finding the shortest path from a single source to multiple destinations in graphs with non-negative weights. - When using efficient data structures like a min-heap, it performs well, especially with sparse graphs.

Disadvantages:
- Limited to Non-Negative Weights: Dijkstra's algorithm does not work with graphs containing negative edge weights; algorithms like Bellman-Ford are better suited for such cases.
- Space Consumption: Extra space is needed to store the priority queue and distance array, impacting its memory usage.

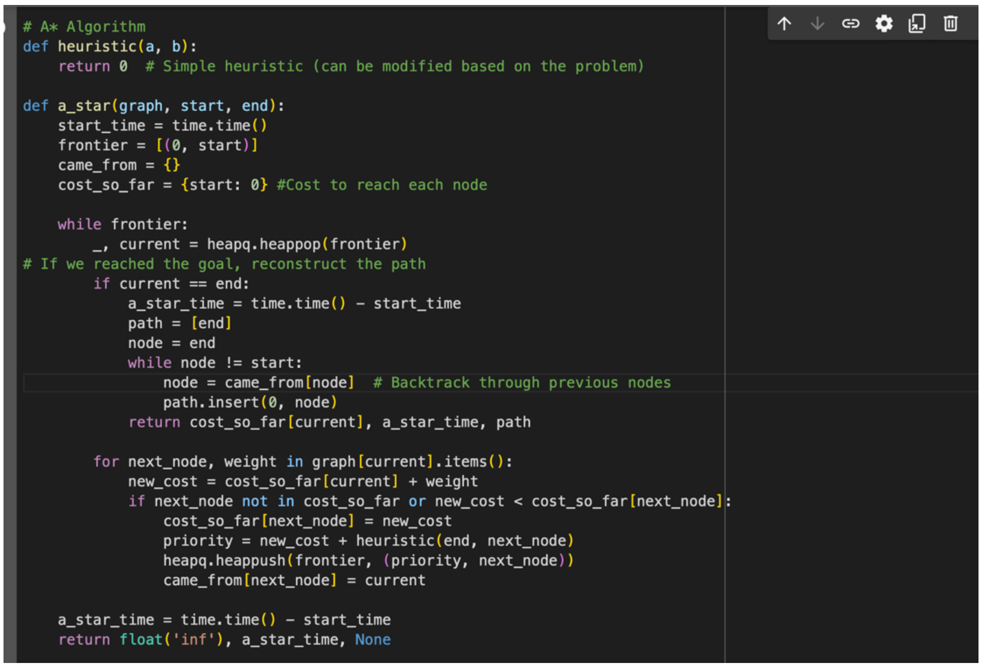
Forward Search Algorithm Advantages:
- Simple to Understand: The algorithm's logic is intuitive, making it easier to grasp and implement effectively.
- Handles Non-Negative Weights: Like Dijkstra's algorithm, it works well with non-negative weights, delivering optimal paths.

Disadvantages:
- Potential Inefficiency: It may end up exploring more nodes than necessary, especially when compared to heuristic-driven algorithms like A*, which can make it slower in large or complex graphs.
- No Heuristic Guidance: Lacking a heuristic function means it doesn't prioritize the most promising paths, often resulting in longer search times.

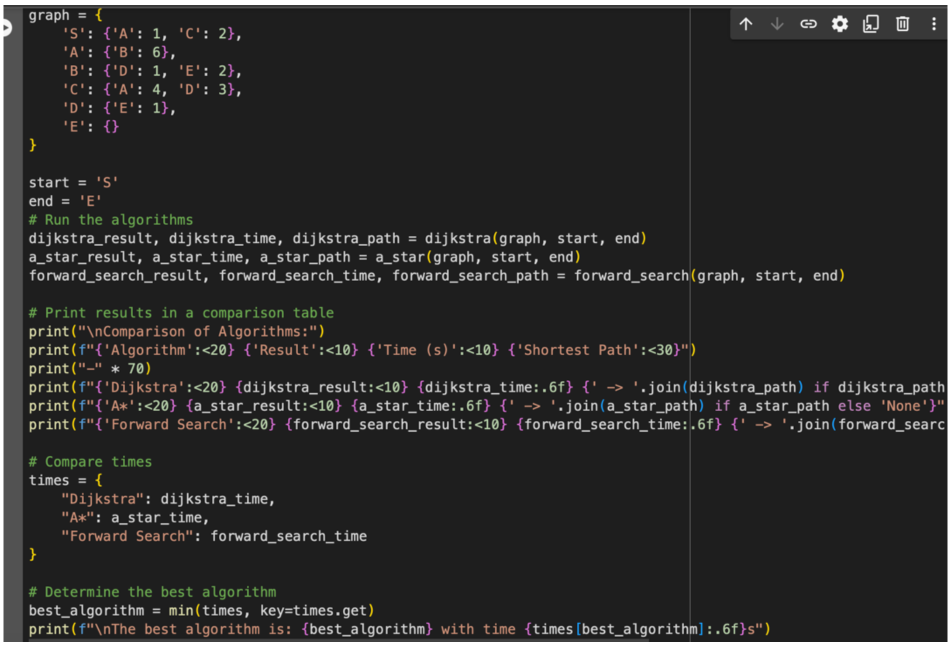
A* Algorithm Advantages:
- Often Faster than Dijkstra's: A* can be much quicker in large graphs when using an effective heuristic, as it prioritizes the most promising paths.

Disadvantages:
- Heavily Dependent on Heuristic Quality: The efficiency and accuracy of A* rely on the heuristic; a poor heuristic may result in inefficient performance.

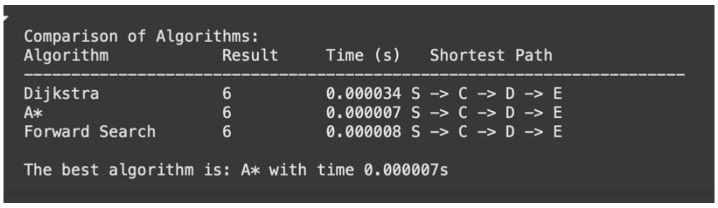
Implementation the best time complexity :



```python
# A* Algorithm
def heuristic(a, b):
    return 0  # Simple heuristic (can be modified based on the problem)

def a_star(graph, start, end):
    start_time = time.time()
    frontier = [(0, start)]
    came_from = {}
    cost_so_far = {start: 0} #Cost to reach each node

    while frontier:
        _, current = heapq.heappop(frontier)
# If we reached the goal, reconstruct the path
        if current == end:
            a_star_time = time.time() - start_time
            path = [end]
            node = end
            while node != start:
                node = came_from[node]  # Backtrack through previous nodes
                path.insert(0, node)
            return cost_so_far[current], a_star_time, path

        for next_node, weight in graph[current].items():
            new_cost = cost_so_far[current] + weight
            if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:
                cost_so_far[next_node] = new_cost
                priority = new_cost + heuristic(end, next_node)
                heapq.heappush(frontier, (priority, next_node))
                came_from[next_node] = current

    a_star_time = time.time() - start_time
    return float('inf'), a_star_time, None
```



```python
graph = {
    'S': {'A': 1, 'C': 2},
    'A': {'B': 6},
    'B': {'D': 1, 'E': 2},
    'C': {'A': 4, 'D': 3},
    'D': {'E': 1},
    'E': {}
}

start = 'S'
end = 'E'
# Run the algorithms
dijkstra_result, dijkstra_time, dijkstra_path = dijkstra(graph, start, end)
a_star_result, a_star_time, a_star_path = a_star(graph, start, end)
forward_search_result, forward_search_time, forward_search_path = forward_search(graph, start, end)

# Print results in a comparison table
print("\nComparison of Algorithms:")
print(f"{'Algorithm':<20} {'Result':<10} {'Time (s)':<10} {'Shortest Path':<30}")
print("-" * 70)
print(f"{'Dijkstra':<20} {dijkstra_result:<10} {dijkstra_time:.6f} {' -> '.join(dijkstra_path) if dijkstra_path
print(f"{'A*':<20} {a_star_result:<10} {a_star_time:.6f} {' -> '.join(a_star_path) if a_star_path else 'None'}"
print(f"{'Forward Search':<20} {forward_search_result:<10} {forward_search_time:.6f} {' -> '.join(forward_searc

# Compare times
times = {
    "Dijkstra": dijkstra_time,
    "A*": a_star_time,
    "Forward Search": forward_search_time
}

# Determine the best algorithm
best_algorithm = min(times, key=times.get)
print(f"\nThe best algorithm is: {best_algorithm} with time {times[best_algorithm]:.6f}s")
```



```
Comparison of Algorithms:
Algorithm           Result     Time (s)   Shortest Path
------------------------------------------------------------------------
Dijkstra            6          0.000034 S -> C -> D -> E
A*                  6          0.000007 S -> C -> D -> E
Forward Search      6          0.000008 S -> C -> D -> E

The best algorithm is: A* with time 0.000007s
```

Algorithm	Worst-Case Time Complexity
Dijkstra's	O((V+E) log V)
A*	O((V+E) log V)
Forward search	O((V+E) log V)

Analysis :

All algorithms have the same theoretical time complexity of (O((V + E) log V)) , the actual execution time can differ based on implementation, graph characteristics.

Despite having the same theoretical time complexity:

Forward search: This algorithm performed well in many scenarios, using a systematic approach to exploring paths. However, it did not outperform the A* algorithm in this case due to its lack of heuristic guidance.

Dijkstra's algorithm: is well suited for finding the shortest path in graphs with non-negative weights.
- However, in this case, it was the slowest of the three algorithms.

The A* :This algorithm emerged as the best performer for the shortest path problem, leveraging its heuristic function to navigate the graph more efficiently. This focused approach allowed it to explore relevant paths more effectively than the other algorithms.

In conclusion, while all three algorithms had the same theoretical time complexity, the **A* algorithm performed the best** for the graph and problem.

Randomized algorithm

Randomized algorithm: is class of algorithms that incorporate randomness into their execution process, allowing them to make decisions based on random choices during the execution. Unlike traditional deterministic algorithms which produced the same output for a given input every time randomized algorithms can give different results for each run with the same input. This can lead to so many effective solutions for problems that may be efficiently addressed by the deterministic methods such as faster execution times.

Selected examples of randomized algorithms:

Dijkstra's Algorithm: is a method for finding the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights.

- **Author**: Edsger W. Dijkstra
- **Published**: The algorithm was first published in 1959.

Positive effects:

1. It is less than consuming the time complexity is(O(ELogV).

2. Greedy approachable is taking implement the algorithm.

3. Has less overhead than Belmont Ford's algorithm.

Negative effects:

1. It can not be implemented easily in a distracted way

2. May or may not work when there is a negative weight edge.

Bellman-ford's Algorithm: is an algorithm which used to find the shortest paths from a single source to all of vertices in a weighted graph.

- **Authors:** Richard Bellman and Lester R. Ford, Jr.
- **Published:** The algorithm was developed in the early 1960s, with significant contributions from both Bellman and Ford.

Positive effects:

1. It can easily be implemented in a disrupted way.

2. Can handle graphs with negative weight edges.

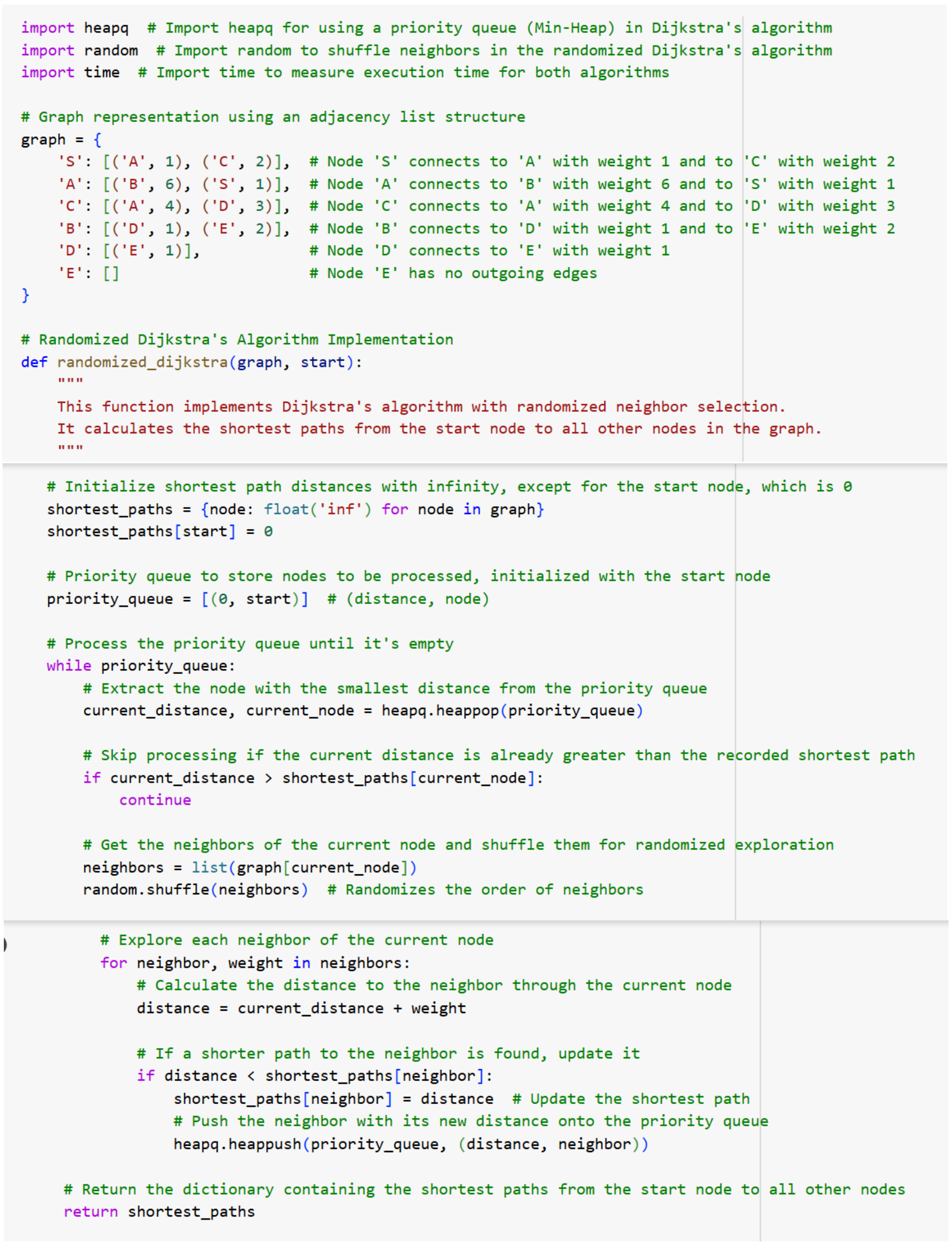
3. Can detect the presence of negative weight cycles.

Negative effects:

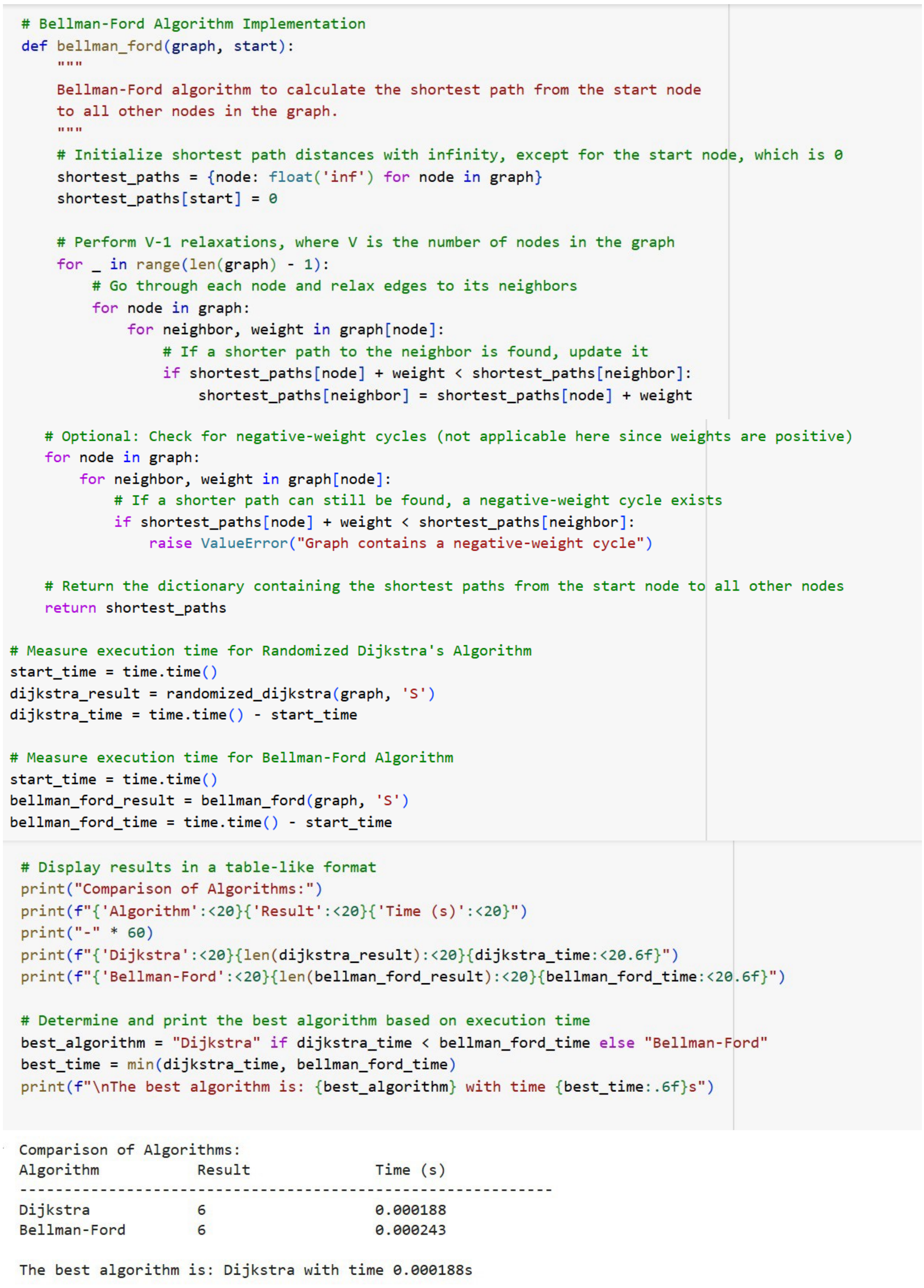
1. More time-consuming than Dijkstra algorithm, time complexity (O(VE).

2. Has more overhead than the Dijkstra or greater.

Randomized Dijkstra's Algorithm implementation



```python
import heapq  # Import heapq for using a priority queue (Min-Heap) in Dijkstra's algorithm
import random  # Import random to shuffle neighbors in the randomized Dijkstra's algorithm
import time  # Import time to measure execution time for both algorithms

# Graph representation using an adjacency list structure
graph = {
    'S': [('A', 1), ('C', 2)],  # Node 'S' connects to 'A' with weight 1 and to 'C' with weight 2
    'A': [('B', 6), ('S', 1)],  # Node 'A' connects to 'B' with weight 6 and to 'S' with weight 1
    'C': [('A', 4), ('D', 3)],  # Node 'C' connects to 'A' with weight 4 and to 'D' with weight 3
    'B': [('D', 1), ('E', 2)],  # Node 'B' connects to 'D' with weight 1 and to 'E' with weight 2
    'D': [('E', 1)],            # Node 'D' connects to 'E' with weight 1
    'E': []                     # Node 'E' has no outgoing edges
}

# Randomized Dijkstra's Algorithm Implementation
def randomized_dijkstra(graph, start):
    """
    This function implements Dijkstra's algorithm with randomized neighbor selection.
    It calculates the shortest paths from the start node to all other nodes in the graph.
    """
    # Initialize shortest path distances with infinity, except for the start node, which is 0
    shortest_paths = {node: float('inf') for node in graph}
    shortest_paths[start] = 0

    # Priority queue to store nodes to be processed, initialized with the start node
    priority_queue = [(0, start)]  # (distance, node)

    # Process the priority queue until it's empty
    while priority_queue:
        # Extract the node with the smallest distance from the priority queue
        current_distance, current_node = heapq.heappop(priority_queue)

        # Skip processing if the current distance is already greater than the recorded shortest path
        if current_distance > shortest_paths[current_node]:
            continue

        # Get the neighbors of the current node and shuffle them for randomized exploration
        neighbors = list(graph[current_node])
        random.shuffle(neighbors)  # Randomizes the order of neighbors

        # Explore each neighbor of the current node
        for neighbor, weight in neighbors:
            # Calculate the distance to the neighbor through the current node
            distance = current_distance + weight

            # If a shorter path to the neighbor is found, update it
            if distance < shortest_paths[neighbor]:
                shortest_paths[neighbor] = distance  # Update the shortest path
                # Push the neighbor with its new distance onto the priority queue
                heapq.heappush(priority_queue, (distance, neighbor))

    # Return the dictionary containing the shortest paths from the start node to all other nodes
    return shortest_paths
```



```python
# Bellman-Ford Algorithm Implementation
def bellman_ford(graph, start):
    """
    Bellman-Ford algorithm to calculate the shortest path from the start node
    to all other nodes in the graph.
    """
    # Initialize shortest path distances with infinity, except for the start node, which is 0
    shortest_paths = {node: float('inf') for node in graph}
    shortest_paths[start] = 0

    # Perform V-1 relaxations, where V is the number of nodes in the graph
    for _ in range(len(graph) - 1):
        # Go through each node and relax edges to its neighbors
        for node in graph:
            for neighbor, weight in graph[node]:
                # If a shorter path to the neighbor is found, update it
                if shortest_paths[node] + weight < shortest_paths[neighbor]:
                    shortest_paths[neighbor] = shortest_paths[node] + weight

    # Optional: Check for negative-weight cycles (not applicable here since weights are positive)
    for node in graph:
        for neighbor, weight in graph[node]:
            # If a shorter path can still be found, a negative-weight cycle exists
            if shortest_paths[node] + weight < shortest_paths[neighbor]:
                raise ValueError("Graph contains a negative-weight cycle")

    # Return the dictionary containing the shortest paths from the start node to all other nodes
    return shortest_paths

# Measure execution time for Randomized Dijkstra's Algorithm
start_time = time.time()
dijkstra_result = randomized_dijkstra(graph, 'S')
dijkstra_time = time.time() - start_time

# Measure execution time for Bellman-Ford Algorithm
start_time = time.time()
bellman_ford_result = bellman_ford(graph, 'S')
bellman_ford_time = time.time() - start_time
```

```python
# Display results in a table-like format
print("Comparison of Algorithms:")
print(f"{'Algorithm':<20}{'Result':<20}{'Time (s)':<20}")
print("-" * 60)
print(f"{'Dijkstra':<20}{len(dijkstra_result):<20}{dijkstra_time:<20.6f}")
print(f"{'Bellman-Ford':<20}{len(bellman_ford_result):<20}{bellman_ford_time:<20.6f}")

# Determine and print the best algorithm based on execution time
best_algorithm = "Dijkstra" if dijkstra_time < bellman_ford_time else "Bellman-Ford"
best_time = min(dijkstra_time, bellman_ford_time)
print(f"\nThe best algorithm is: {best_algorithm} with time {best_time:.6f}s")
```

```
Comparison of Algorithms:
Algorithm           Result              Time (s)
------------------------------------------------------------
Dijkstra            6                   0.000188
Bellman-Ford        6                   0.000243

The best algorithm is: Dijkstra with time 0.000188s
```

Algorithms Names	Time complexity
Dijkstra’s Algorithm	O((V+E) ·logV)
Bellman Ford’s Algorithm	O(V·E)

Result: Which Algorithm is Better?

Best Algorithm in Terms of Time Complexity:

Dijkstra’s Algorithm is the best choice when all weights are **positive**, due to its efficient **time complexity**.

Dijkstra’s Algorithm (Time Complexity):

O((V+E)·logV)

- **V** is the number of **vertices**.
- **E** is the number of **edges**.
  Data structure: In Dijkstra's algorithm, a Priority Queue is used to quickly extract the node with the smallest distance.

Why is Dijkstra’s Faster than Bellman-Ford?

- **Bellman-Ford** has a **higher time complexity: O(V·E)**
- It needs to **relax all edges (V-1 times)**, which makes it slower
- In the Bellman-Ford algorithm, **no priority queue** or heap is used. Instead, the relaxation process is repeated for all edges directly V-1 times, where V is the number of vertices.

In conclusion:

- **Dijkstra's Algorithm** will be **faster and more efficient** due to its **lower time complexity**.
- **Bellman-Ford** offers additional features (handling **negative weights** and detecting **negative cycles**), but it is **slower** for graphs with only positive weights.

Divide and conquer strategy:

It is a strategy for solving some problems. It divides the big problem into smaller problems and solves them independently. Then, in the end, it integrates the small solutions together to become the final solution to the problem.

Algorithmic data:

The divide and conquer algorithm was not written by one parson, but is a general strategy in computer science and mathematics that has evolved over time and is used by many famous algorithms. This method was widely adopted by John von Neumann in 1945 when he developed the merge sort algorithm, which is one of the first algorithms to use the divide and conquer concept systematically. Since then, this algorithm has become the basis for many algorithms used in many fields, such as quick sorting, data matrix partitioning algorithms, and others.

Importance:

The Divide and Conquer algorithm is important because it allows you to deal with complex problems by breaking them down into simpler parts that can be solved more quickly and in a less complex way. Its use is particularly effective in improving software speed and reducing computational complexity, and is also used to develop quick solutions to big data problems.

Algorithms that use divide and conquer strategy to solve the shortest path problem:

1) Shortest Path Faster Algorithm
2) Johnson’s Algorithm
3) Divide and Conquer Shortest Path (DCSP)
4) MapReduce Shortest Path

Shortest Path Faster Algorithm:

This algorithm is used in graphs to find the shortest path. It is derived from Bellman's algorithm with improvements and relies on the divide and conquer strategy to reduce computational operations.

- **Positive effects:**
    1) Its efficiency is improved and it is usually faster than the bellman in some cases.
    2) It can handle negative weights.
    3) Easy to understand and implement.

- **Negative effects:**
    1) worst-case time complexity is O(VE) which means it is less efficient than some other algorithms such as Dijkstra.
    2) It may have poor performance if the edges are arranged in some way due to its sensitivity to structure.
    3) May be ineffective with very large graphics.
    4) It can't handle negative cycles.

Johnson’s Algorithm:

It is used in directed graphs and can handle negative weights but cannot handle negative cycles. Johnson relies on the Bellman and Dijkstra algorithm to calculate the shortest path.

Positive effects:

1) It is effective in rare graph.
2) It can handle negative weights.
3) It uses the advantages of other algorithms such as Bellman to handle negative weights and dextra to speed up the process of finding the shortest path with non-negative weights.
4) It can be used in different applications and is not limited to one type of application.

Negative effects:

1) Its time complexity is considered high in dense graphs.
2) It needs to be re-weighted which adds to the complexity.
3) It requires a deep understanding of the algorithms involved and is not easy to implement.

Shortest Path Divide and Conquer (DCSP):

DCSP is an algorithm used to find the shortest path between two points in a graph. The algorithm is based on the principle of "divide and conquer", where a large problem is solved (to find the shortest path) by dividing it into smaller parts and combining the results to obtain the final solution. The goal of using this strategy is to reduce the calculation time by reducing the number of operations required to reach the solution.

Positive effects:

1) Increase efficiency: The DCSP algorithm is often more efficient than other algorithms, especially in complex graphs.
2) Ability to handle negative weights: The algorithm can handle negative weights, which is useful in applications that need to take negative costs into account.
3) Ease of partitioning: The algorithm is based on a partitioning strategy, which allows smaller parts to be analyzed and solved independently, making them easier to understand and apply.

Negative effects:

1) Time complexity: In the worst case, the time complexity can be O(V^2) or higher, making it slower than algorithms like DJestra in some applications.
2) Structure sensitive: Inconsistent or arranged graph structures are negatively affected, affecting performance.
3) Not suitable for very large charts: It may not work with large charts where the amount of operations required increases exponentially.
4) Negative cycles cannot be handled: Like the Bellman-Ford algorithm, the DCSP algorithm cannot handle cycles with negative weights.

MapReduce Shortest Path:

The MapReduce Shortest Path algorithm is an algorithm used to calculate the shortest path for large graphs using the MapReduce framework. This common framework features breaking big data into smaller chunks that are processed in parallel by multiple servers, and combining results from each server to form the end result. This algorithm is suitable for large charts such as social networks and maps, which require parallel processing to process data quickly and efficiently.

Positive effects:

1) Scalability: The algorithm distributes processing across multiple servers, making it applicable to large datasets and suitable for large graphs.
2) Parallelism: The MapReduce framework reduces execution time by performing calculations in parallel, enabling faster productivity.
3) Reliability: MapReduce improves processing reliability as work is distributed and tasks can be resumed if the server fails.
4) Suitable for processing big data: The algorithm is suitable for graphs containing millions to billions of nodes, which are difficult to process with traditional algorithms.

Negative effects:

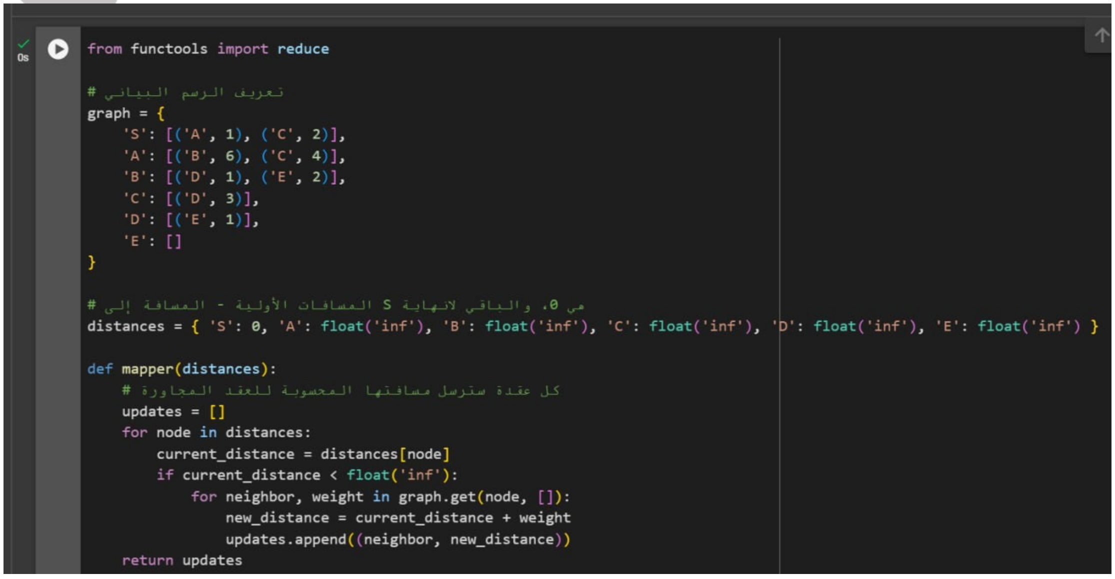
1) Complex implementation: Setting up algorithms using the MapReduce framework requires in-depth knowledge of parallelism and distribution, making implementation relatively complex and difficult.
2) Time delay: Although MapReduce helps save time, it distributes work and eventually collects results, which can cause delays and inappropriate for tasks that require quick answers.
3) High resource utilization: To implement MapReduce effectively, a large amount of resources (memory, processors, servers) is required, which can increase operational costs.
4) Inefficient for small graphs: The algorithm may be inefficient or useless when the data is small, as distributions are unnecessary and may cause unnecessary complications.

Algorithm name	Time complexity
Shortest Path Faster Algorithm	O(E + V log V)
Johnson's Algorithm	(OV^2 log V + VE)
Shortest Path Divide and Conquer (DCSP)	O(V2logV)
MapReduce Shortest Path	O(V+E)

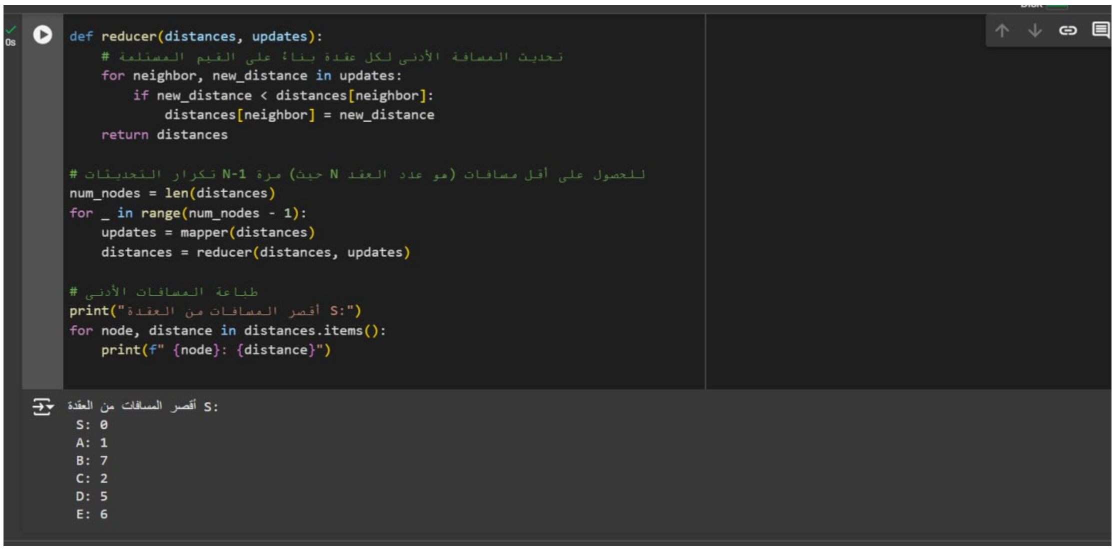
The best algorithm to solve the shortest path problem using the divide and conquer strategy:

Building the time complexity comparison table above we find that the algorithm " **MapReduce Shortest Path** " with **O(V+E)** time complexity is less than others in terms of time complexity because it is more efficient and faster in dealing with large graph and uses less calculations, so it is the best for solving the shortest path problem.

Code for the better algorithm :



```python
from functools import reduce

# تعريف الرسم البياني
graph = {
    'S': [('A', 1), ('C', 2)],
    'A': [('B', 6), ('C', 4)],
    'B': [('D', 1), ('E', 2)],
    'C': [('D', 3)],
    'D': [('E', 1)],
    'E': []
}

# المسافات الأولية - المسافة إلى S هي 0، والباقي لانهاية
distances = { 'S': 0, 'A': float('inf'), 'B': float('inf'), 'C': float('inf'), 'D': float('inf'), 'E': float('inf') }

def mapper(distances):
    # كل عقدة سترسل مسافتها المحسوبة للعقد المجاورة
    updates = []
    for node in distances:
        current_distance = distances[node]
        if current_distance < float('inf'):
            for neighbor, weight in graph.get(node, []):
                new_distance = current_distance + weight
                updates.append((neighbor, new_distance))
    return updates
```



```python
def reducer(distances, updates):
    # تحديث المسافة الأدنى لكل عقدة بناءً على القيم المستلمة
    for neighbor, new_distance in updates:
        if new_distance < distances[neighbor]:
            distances[neighbor] = new_distance
    return distances

# تكرار التحديثات N-1 مرة (حيث N هو عدد العقد) للحصول على أقل مسافات
num_nodes = len(distances)
for _ in range(num_nodes - 1):
    updates = mapper(distances)
    distances = reducer(distances, updates)

# طباعة المسافات الأدنى
print("أقصر المسافات من العقدة S:")
for node, distance in distances.items():
    print(f" {node}: {distance}")
```

```
أقصر المسافات من العقدة S:
 S: 0
 A: 1
 B: 7
 C: 2
 D: 5
 E: 6
```

Algorithm name	Authors	Publication date	Time complexity	Data structures used	Summary
Floyd-Warshay	Folyd and Warshall	1962	O(V^3)	2D Array	find the shortest distance between all pairs of nodes in a graph. It works by gradually building the shortest paths using a matrix representation called the "distance ".matrix
Bellman-Ford	Richard Bellman, Lester Ford	1958	O(VE)	Array	Computes shortest paths in graphs with negative weights.
A-star	Peter Hart, Nils Nilsson, Bertram Raphael	1968	O((v+E)logv)	Priority Queue	Combines cost and heuristic for efficient pathfinding.
Breadth-First Search	Konrad Zuse, Edward F. Moore , C. Y. Lee	1959	O(V+E)	Queue	It is effective in the problem of finding the shortest path. It starts from a point and does not move until it has finished visiting all the neighbors of the point from which it started without repeating the visit.
Dijkstra's algorithm	Dijkstra, Leyzorek et al.	1959	O((V+E)log V)	priority queue	The problem is solved by finding the shortest path from the starting point, called the "source", to the rest of the other points, where the lower weight has priority in Priority Q, but it cannot see negative weights.
Parallel Dijkstra’s algorithm	The "Parallel Dijkstra’s Algorithm" was a collaborative effort in computer science with no single author credited.	1959	$O\left(\frac{V^2}{P} + V \cdot \log(P)\right)$	Priority Queue	Parallel Dijkstra's algorithm speeds up finding the shortest path in a graph by splitting the graph into smaller parts processed simultaneously on multiple processors.

Binary search	Nils Nilsson, Peter Hart	1957	O(Log n)	Arrays	This algorithm is used to find an element in a sorted array by dividing the array into two halves and determining which half contains the desired element.
Linear Discriminant Anaiysis (LDA)	Ronald Fisher	1936	O(n*d^2)	Arrays, Matrices	Linear discriminant analysis (LDA) is a supervised machine learning algorithm used for classification and dimensionality reduction. It aims to find the best linear combination of features that separates classes, assuming that the data within each class follows a Gaussian distribution.
Johnson's Algorithm:	Donald B.Johnson	1977	O(V^2 log V+VE)	Priority Queue	It solves the shortest path for all pairs and can handle negative weights but cannot .handle negative loops
Forward Search	Various	N/A	O((v+E)logv)	Queue	Expands the least-cost node first, focusing on reaching the target efficiently.
Shortest Path Divide and Conquer (DCSP)	Many researchers in the field of computer science have studied and analyzed the algorithms of the shortest path.	1981	O(V2logV)	Graph Priority Queue Array Hash Table	The DCSP algorithm aims to find the shortest path between two points in a graph by dividing the problem into subproblems. The algorithm divides the graph into two or more parts, calculates the shortest paths separately in each part, and then combines the results to get the shortest path between the two points. This approach enhances performance compared to traditional algorithms, making it effective in

					dealing with large and complex graphs.
MapReduce Shortest Path	Edsgar Dijkstra , Peter Hart , Nils Nilsson.	It started in the early seventies	O(V+E)	Graph Priority Queue Array Hash Table	The Redius Map Shortest Path algorithm is used to find the shortest path in a network of nodes (vertices) and links (edges) that represent a map or transport system. The algorithm is based on determining the shortest distance between two points using techniques such as sequential search, depth search, or display search, with a focus on reducing the time spent calculating routes. This algorithm is effective in applications such as navigation system and route planning.
Bidirectional Dijkstra	Peter E.Hart Nils Nilsson	1968	O(b^d/2)	Priority Queue	An improvement to the traditional Dijkstra algorithm is to perform the search from both the start and end points simultaneously, which reduces the number of nodes explored and speeds up the process of finding the shortest path, especially in large graphs.
A* algorithm	Peter E.Hart Nils Nilsson	1968	O(b^d)	Priority Queue Graph	A is a search algorithm used to find the shortest path from a starting point to an ending point in a graph based on the use of a heuristic function that combines the actual cost of reaching a node with the estimated cost of reaching the goal. A is effective in many

					applications such as video games and navigation systems, where it offers a balance between efficiency and accuracy.
Bidirectional Search	J. B. Kruskal and R. C. D. Peled	1972	O(b^d/2)	Queue	It'san algorithm that explores paths from both the starting point and the goal simultaneously. It uses queues to manage nodes being explored in each direction
Greedy Best-First Search	Peter Hart and Nils Nilsson and Bertrand Raphael	1970	O(b^d)	Priority Queue	, **Greedy Best-First Search** prioritizes nodes based on a heuristic function, using a priority queue to always explore the most promising paths first.
Shortest Path Faster Algorithm (SPFA)	FanDingDuan	1994	O(VE)	queue	SPFA optimizes pathfinding in weighted graphs over Bellman-Ford. It iteratively refines distances until no more improvements are possible. Faster than Bellman-Ford generally, SPFA can have a time in (VE)complexity of O .some instances

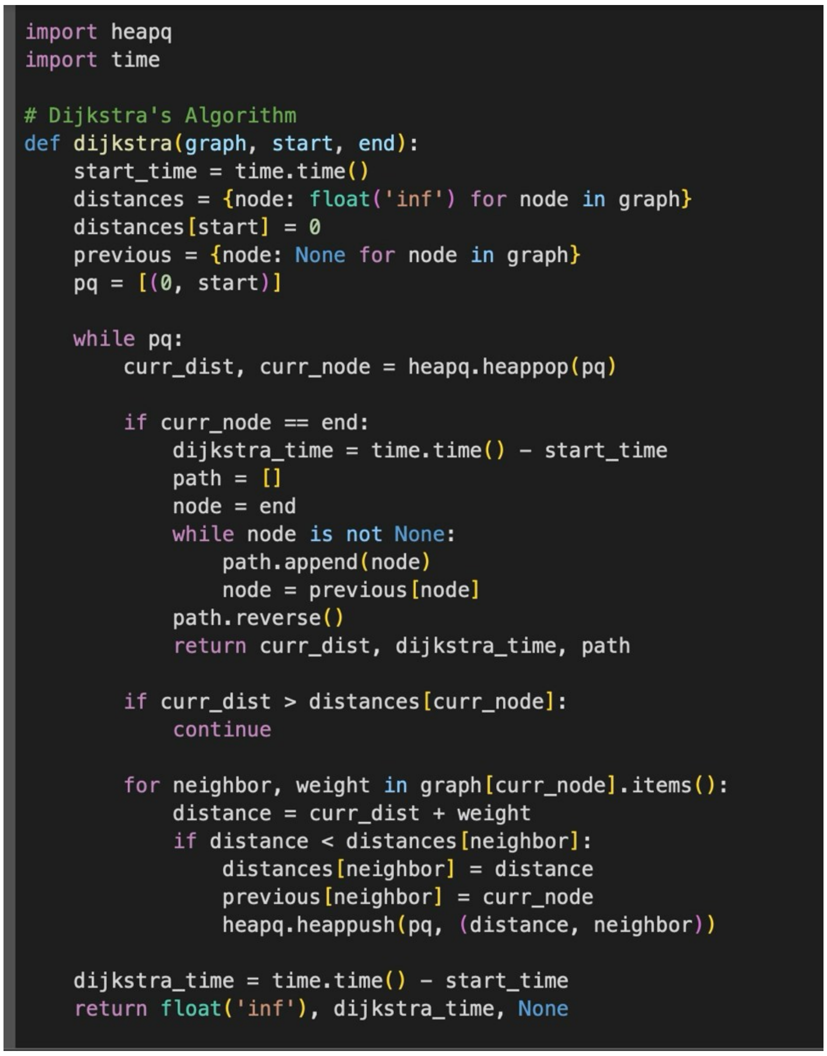
The best algorithm in each strategy is:

Based on our results, research and comparison to find the best algorithm for each of the six strategies, we found that these three strategies are the best:

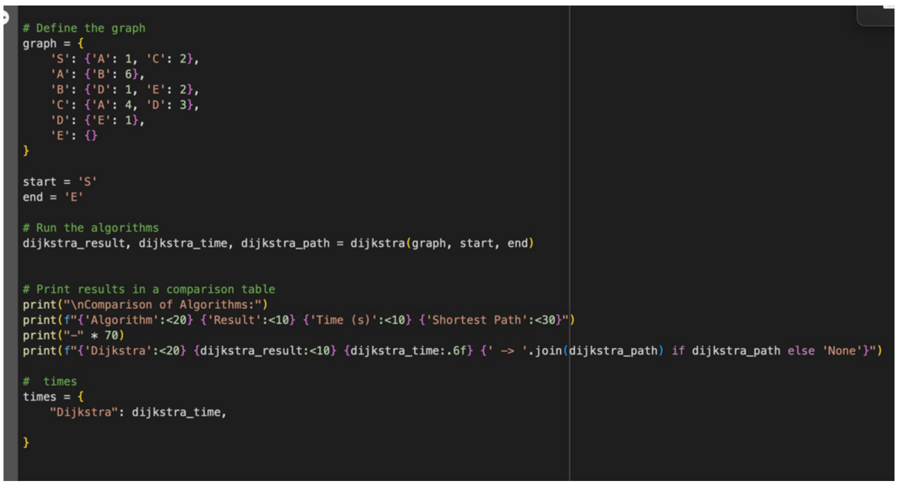
- Greedy strategy
  Includes **Dijkstra's algorithm**
- Heuristic strategy
  A* algorithm includes
- Dynamic strategy
  Includes **Floyd-Warshall algorithm**

The best algorithm among them to solve the shortest path problem is Dijkstra's algorithm .

This is the implementation :



```python
import heapq
import time

# Dijkstra's Algorithm
def dijkstra(graph, start, end):
    start_time = time.time()
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    previous = {node: None for node in graph}
    pq = [(0, start)]

    while pq:
        curr_dist, curr_node = heapq.heappop(pq)

        if curr_node == end:
            dijkstra_time = time.time() - start_time
            path = []
            node = end
            while node is not None:
                path.append(node)
                node = previous[node]
            path.reverse()
            return curr_dist, dijkstra_time, path

        if curr_dist > distances[curr_node]:
            continue

        for neighbor, weight in graph[curr_node].items():
            distance = curr_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous[neighbor] = curr_node
                heapq.heappush(pq, (distance, neighbor))

    dijkstra_time = time.time() - start_time
    return float('inf'), dijkstra_time, None
```



```python
# Define the graph
graph = {
    'S': {'A': 1, 'C': 2},
    'A': {'B': 6},
    'B': {'D': 1, 'E': 2},
    'C': {'A': 4, 'D': 3},
    'D': {'E': 1},
    'E': {}
}

start = 'S'
end = 'E'

# Run the algorithms
dijkstra_result, dijkstra_time, dijkstra_path = dijkstra(graph, start, end)


# Print results in a comparison table
print("\nComparison of Algorithms:")
print(f"{'Algorithm':<20} {'Result':<10} {'Time (s)':<10} {'Shortest Path':<30}")
print("-" * 70)
print(f"{'Dijkstra':<20} {dijkstra_result:<10} {dijkstra_time:.6f} {' -> '.join(dijkstra_path) if dijkstra_path else 'None'}")

#  times
times = {
    "Dijkstra": dijkstra_time,

}
```

References:

Dynamic programming

1. https://www.codingdrills.com/tutorial/introduction-to-dynamic-algorithms/all-pairs-shortest-paths
2. https://www.geeksforgeeks.org/time-and-space-complexity-of-floyd-warshall-algorithm/
3. https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/

Divide and conquer:

1. https://www.bing.com/ck/a?!&&p=097262b51da7058cJmltdHM9MTcyOTkwMDgwMCZpZ3VpZD0xNWMzOWE3MC02MzVhLTZhNzItMWE4Ny04ODc4NjJkMTZiMDcmaW5zaWQ9NTQ3Ng&ptn=3&ver=2&hsh=3&fclid=15c39a70-635a-6a72-1a87-887862d16b07&psq=Divide+and+Conquer+algorithm+history+John+von+Neumann+Merge+Sort+1945&u=a1aHR0cHM6Ly9zaW1wbGUud2lraXBlZGlhLm9yZy93aWtpL01lcmdlX3NvcnQjOn46dGV4dD1NZXJnZSUyMHNvcnQlMjAlMjhvciUyMG1lcmdlc29ydCUyOSUyMGlzJTIwYW4lMjBkaXZpZGUlMjBhbmQsbGlzdHMlMjBpcyUyMHRyaXZpYWwlMkMlMjBhbmQlMjBtZXJnaW5nJTIwdGhlbSUyMHRvZ2V0aGVyJTIwaXNuJTI3dC4&ntb=1
2. Merge Sort Demystified: A Beginner's Guide to Divide and Conquer Sorting - DEV Community
3. Merge Sort Explained: A Data Scientist’s Algorithm Guide | NVIDIA Technical Blog
4. https://en.wikipedia.org/wiki/Shortest_path_problem
5. https://stackoverflow.com/questions/73189578/faster-algorithm-than-dijikstra-for-finding-shortest-path-to-all-nodes-starting
6. https://brilliant.org/wiki/johnsons-algorithm/
7. https://ypsilon.dev/ar/blog/%d9%85%d8%a7%d8%b0%d8%a7-%d9%8a%d8%b9%d9%86%d9%8a-johnsons-algorithm-

%d9%81%d9%8a-%d9%85%d8%ac%d8%a7%d9%84-%d8%a7%d9%84%d8%ae%d9%88%d8%a7%d8%b1%d8%b2%d9%85%d9%8a%d8%a7%d8%aa-%d9%88%d9%87%d9%8a%d8%a7/

Greedy :

1. https://youtu.be/ySN5Wnu88nE?si=_Rq0fH03s8WhRfrQ
2. https://www.geeksforgeeks.org/difference-between-/
3. https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/#:~:text=Time%20Complexity%20of%20Dijkstra's%20Algorithm,E%20l%20o%20g%20V%20)%20
4. https://youtu.be/pVfj6mxhdMw?si=9kPysKhpfEQWNgjI

Randomized:

1. https://almuhammadi.com/sultan/books_2020/Alsuwaiyel_2016.pdf
2. https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/
3. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/?ref=header_outind
4. https://medium.com/@brianpatrao1996/dijkstras-vs-bellman-ford-algorithm-383e4771c2cb
5. https://stackoverflow.com/questions/19482317/bellman-ford-vs-dijkstra-under-what-circumstances-is-bellman-ford-better
6. https://brilliant.org/wiki/randomized-algorithms-overview/
7. https://arxiv.org/pdf/2307.04139

Parallel Algorithms:

1. https://developer.nvidia.com/discover/shortest-path-problem
2. https://geniusjournals.org/index.php/ejet/article/download/2755/2360
3. https://www.researchgate.net/publication/241684993_Parallel_Algorithms
4. https://www.cs.cmu.edu/~scandal/nesl/algorithms.html
5. https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_structure.htm
6. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ye-Fall-2012-CSE633.pdf

Heuristic Algorithms

1. Bidirectional Search - GeeksforGeeks

2. Greedy Best first search algorithm - GeeksforGeeks

3. A* Search Algorithm - Wikipedia

4. Bidirectional Search - Wikipedia

5. CH Warmups | Contraction Hierarchies Guide

6. bidirectional_dijkstra — NetworkX 3.4.2 documentation