

Báo Cáo Cuối kỳ Môn Tổ Hợp Và Lý Thuyết Đồ Thị
Project 5: Shortest Path Problems on Graphs

Trần Mạnh Đức

Ngày 23 tháng 7 năm 2025

Mục lục

1	Bài toán 14: Thuật toán Dijkstra trên Đơn đồ thị Hữu hạn	3
1.1	Phân tích và Phát biểu Bài toán	3
1.1.1	Phân tích	3
1.2	Nền tảng Toán học và Chứng minh	3
1.2.1	Chứng minh tính đúng đắn (Proof by Contradiction)	3
1.3	Mô phỏng Thuật toán	4
1.4	Cài đặt và Phân tích Mã nguồn	4
1.4.1	Cài đặt bằng Python	4
1.4.2	Cài đặt bằng C++	6
1.5	Phân tích Nâng cao	10
1.6	Kết luận cho Bài toán 14	10
2	Bài toán 15: Thuật toán Dijkstra trên Đa đồ thị Hữu hạn	11
2.1	Phân tích và Đặc điểm của Đa đồ thị	11
2.2	Tính Áp dụng của Thuật toán Dijkstra	11
2.3	Mô phỏng từng bước trên Đa đồ thị	11
2.4	Cài đặt và Biểu diễn Đa đồ thị	12
2.4.1	Cài đặt bằng Python	13
2.4.2	Cài đặt bằng C++	16
2.5	Kết luận cho Bài toán 15	19
3	Bài toán 16: Thuật toán Dijkstra trên Đồ thị Tổng quát	20
3.1	Phân tích và Định nghĩa Đồ thị Tổng quát	20
3.2	Ảnh hưởng đến Thuật toán Dijkstra – Một Phân tích Phê bình	20
3.2.1	Xử lý các Đặc điểm của Đồ thị Tổng quát	20
3.3	Cài đặt (Với Giả định Bắt buộc)	21
3.3.1	Cài đặt bằng Python (Giả định trọng số không âm)	21
3.3.2	Cài đặt bằng C++ (Giả định trọng số không âm)	23
3.4	Phân tích và Mâu thuẫn Lý thuyết	27
3.5	Thuật toán Thay thế cho Đồ thị Tổng quát Thực sự	27
3.5.1	Thuật toán Bellman-Ford	27
3.5.2	Thuật toán SPFA (Shortest Path Faster Algorithm)	29
3.6	Kết luận cho Bài toán 16	31
3.7	Kết luận cho Bài toán 16	32

1 Bài toán 14: Thuật toán Dijkstra trên Đơn đồ thị Hữu hạn

1.1 Phân tích và Phát biểu Bài toán

Đề bài 14

Let $G = (V, E)$ be a finite simple graph. Implement the Dijkstra's algorithm to find the shortest path problem on G .

(Cho $G = (V, E)$ là một đơn đồ thị hữu hạn. Hãy cài đặt thuật toán Dijkstra để giải bài toán tìm đường đi ngắn nhất trên G .)

1.1.1 Phân tích

- Đơn đồ thị (Simple Graph): Là đồ thị vô hướng, không có khuyên và cạnh song song.
- Giả định hợp lý: Bài toán yêu cầu tìm đường đi ngắn nhất và sử dụng thuật toán Dijkstra, điều này ngụ ý sự tồn tại của trọng số trên các cạnh. Do đó, ta sẽ làm việc với một đơn đồ thị hữu hạn có trọng số không âm, nơi mỗi cạnh $e \in E$ được gán một trọng số $w(e) \geq 0$.

1.2 Nền tảng Toán học và Chứng minh

Thuật toán Dijkstra dựa trên chiến lược tham lam (Greedy Strategy) và nguyên lý cấu trúc con tối ưu (Optimal Substructure).

1.2.1 Chứng minh tính đúng đắn (Proof by Contradiction)

Ta chứng minh rằng khi thuật toán chọn một đỉnh u và thêm vào tập S (tập các đỉnh đã có đường đi ngắn nhất cuối cùng), thì khoảng cách $d[u]$ lúc đó chính là khoảng cách ngắn nhất thực sự $\delta(s, u)$.

1. Giả sử phản chứng: Tồn tại một đỉnh u được thêm vào S đầu tiên mà $d[u] > \delta(s, u)$.
2. Phân tích: Nếu có một đường đi ngắn hơn tới u , đường đi đó (gọi là P) phải xuất phát từ nguồn s , đi qua một số đỉnh trong S , sau đó đi qua một cạnh (x, y) để vào vùng $V - S$ (với $x \in S, y \in V - S$), và cuối cùng đến u .
3. Lập luận:
 - Vì $x \in S$ và được chọn trước u , theo giả định quy nạp, ta có $d[x] = \delta(s, x)$.
 - Khi thuật toán xử lý x , nó đã relax cạnh (x, y) , do đó $d[y] \leq d[x] + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$.
 - Vì y nằm trên đường đi ngắn nhất P tới u và trọng số các cạnh không âm, ta có $\delta(s, y) \leq \delta(s, u)$.
 - Từ giả định phản chứng, $\delta(s, u) < d[u]$.
 - Kết hợp lại: $d[y] \leq \delta(s, y) \leq \delta(s, u) < d[u]$.
4. Mâu thuẫn: Ta có $d[y] < d[u]$. Điều này mâu thuẫn với việc thuật toán đã chọn u từ tập $V - S$ vì nó có giá trị d nhỏ nhất. Do đó, giả định phản chứng là sai.

Kết luận: Thuật toán Dijkstra luôn đúng với đồ thị có trọng số không âm.

1.3 Mô phỏng Thuật toán

Xét đồ thị ví dụ và đỉnh nguồn là A.

Bước	Chọn (u)	Tập S	Cập nhật distances và PQ
0	-	\emptyset	$d = \{A : 0, \dots : \infty\}$, $PQ = \{(0, A)\}$
1	A	$\{A\}$	$d[B] = 10, d[C] = 3$. $PQ = \{(3, C), (10, B)\}$
2	C	$\{A, C\}$	$d[B] = 4, d[D] = 11, d[E] = 5$. $PQ = \{(4, B), (5, E), \dots\}$
3	B	$\{A, C, B\}$	$d[D] = 6$. $PQ = \{(5, E), (6, D), \dots\}$
4	E	$\{A, C, B, E\}$	Không cập nhật. $PQ = \{(6, D), \dots\}$
5	D	$\{A, C, B, E, D\}$	Không cập nhật. PQ rỗng.

Kết quả cuối cùng: distances = {A: 0, B: 4, C: 3, D: 6, E: 5}.

1.4 Cài đặt và Phân tích Mã nguồn

1.4.1 Cài đặt bằng Python

```
1 import heapq
2
3 def dijkstra_simple_graph(graph: dict, start_node: str):
4     """
5     Implements Dijkstra's algorithm for a simple graph.
6
7     This function calculates the shortest paths from a single source node
8     to all other
9     nodes in a simple graph with non-negative edge weights.
10
11     :param graph: A dictionary of dictionaries representing the graph.
12                   Example: {'A': {'B': 10, 'C': 3}, 'B': {'A': 10}}
13     :param start_node: The starting node for the algorithm.
14     :return: A tuple containing two dictionaries: (distances,
15             predecessors).
16             'distances' maps each node to its shortest distance from the
17             source.
18             'predecessors' maps each node to its preceding node in the
19             shortest path.
20     """
21     # --- Initialization ---
22
23     # Initialize distances for all nodes to infinity, and the start node
24     # to 0.
25     distances = {node: float('inf') for node in graph}
26     distances[start_node] = 0
27
28     # This dictionary will store the path for reconstruction.
29     predecessors = {node: None for node in graph}
30
31     # The priority queue stores tuples of (distance, node).
32     pq = [(0, start_node)]
33
34     # --- Main Loop ---
35
36     while pq:
```

```

32     # Get the node with the smallest distance from the priority queue
33     current_distance, current_node = heapq.heappop(pq)
34
35     # Optimization: If we've already found a better path, skip.
36     if current_distance > distances[current_node]:
37         continue
38
39     # --- Relaxation Step ---
40
41     # Iterate through all neighbors of the current node.
42     # .items() is used as the value of each graph key is another
    dictionary.
43     for neighbor, weight in graph[current_node].items():
44         distance_through_u = current_distance + weight
45
46         # If we found a new, shorter path to the neighbor...
47         if distance_through_u < distances[neighbor]:
48             # ...update the new shortest distance.
49             distances[neighbor] = distance_through_u
50             # ...record the predecessor for path reconstruction.
51             predecessors[neighbor] = current_node
52             # ...and push the new path information to the priority
    queue.
53             heapq.heappush(pq, (distance_through_u, neighbor))
54
55     return distances, predecessors
56
57 def reconstruct_path(predecessors: dict, start_node: str, end_node: str):
58     """
59     Reconstructs the shortest path from the predecessors dictionary.
60     """
61     path = []
62     current_node = end_node
63     while current_node is not None:
64         path.insert(0, current_node)
65         current_node = predecessors.get(current_node)
66
67     if path and path[0] == start_node:
68         return path
69     else:
70         return None
71
72
73 # =====
74 # MAIN EXECUTION BLOCK - DEMONSTRATION FOR PROBLEM 14 (SIMPLE GRAPH)
75 # =====
76 if __name__ == "__main__":
77     # --- 1. Graph Creation for a Simple Graph ---
78     # A dictionary of dictionaries is a natural way to represent a simple
    graph.
79     # The keys of the inner dicts are unique, preventing parallel edges.
80     simple_graph = {
81         'A': {'B': 10, 'C': 3},
82         'B': {'A': 10, 'C': 1, 'D': 2},
83         'C': {'A': 3, 'B': 1, 'D': 8, 'E': 2},
84         'D': {'B': 2, 'C': 8, 'E': 5},
85         'E': {'C': 2, 'D': 5}

```

```

86     }
87
88     start_node = 'A'
89
90     # --- 2. Execution ---
91     distances, predecessors = dijkstra_simple_graph(simple_graph,
92 start_node)
93
94     # --- 3. Reporting Results ---
95     print("=" * 60)
96     print("Dijkstra's Algorithm Report for Simple Graph (Problem 14)")
97     print(f"Source Node: '{start_node}'")
98     print("=" * 60)
99
100    # Sort nodes for consistent output order.
101    sorted_nodes = sorted(simple_graph.keys())
102
103    for node in sorted_nodes:
104        print(f"\n--- Destination: '{node}' ---")
105        dist = distances.get(node)
106
107        print(f"    Shortest Distance: ", end="")
108        if dist == float('inf'):
109            print("UNREACHABLE")
110        else:
111            print(dist)
112
113        path = reconstruct_path(predecessors, start_node, node)
114        print("    Path: ", end="")
115        if path:
116            print(" -> ".join(path))
117        else:
118            print(f"'{node}' (Source Node)")
119
120    print("=" * 60)

```

Listing 1: Cài đặt Dijkstra bằng Python, có truy vết đường đi.

1.4.2 Cài đặt bằng C++

```

1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <limits>
5  #include <string>
6  #include <map>
7  #include <algorithm>
8  #include <functional> // For std::greater
9
10 // Use long long to avoid overflow with long paths.
11 using ll = long long;
12 // Use type aliases for better readability.
13 using Edge = std::pair<int, int>; // {weight, neighbor_node}
14 using Graph = std::vector<std::vector<Edge>>;
15
16 /**
17  * @class DijkstraSolver

```

```

18  * @brief A class to compute shortest paths from a single source using
    * Dijkstra's algorithm.
19  *
20  * This class is designed for simple graphs and encapsulates the data and
    * logic for the algorithm.
21  */
22  class DijkstraSolver {
23  public:
24      std::vector<ll> distances;
25      std::vector<int> predecessors;
26      int num_nodes;
27
28      /**
29       * @brief Constructs a DijkstraSolver for a graph with a given number
        * of nodes.
        * @param n The number of nodes in the graph.
        */
31      DijkstraSolver(int n) : num_nodes(n) {
32          // Initialize distances to infinity.
33          distances.resize(n, std::numeric_limits<ll>::max());
34          // Initialize predecessors to -1 (indicating no predecessor).
35          predecessors.resize(n, -1);
36      }
37
38      /**
39       * @brief Executes Dijkstra's algorithm on the given graph from a
        * start node.
        * @param adj The graph, represented as an adjacency list.
        * @param start_node The index of the source node.
        */
43      void solve(const Graph& adj, int start_node) {
44          // The distance from the start node to itself is 0.
45          distances[start_node] = 0;
46
47          // Min-heap: stores {distance, node}.
48          // std::greater is used to make the priority_queue a min-heap.
49          std::priority_queue<std::pair<ll, int>,
50                          std::vector<std::pair<ll, int>>,
51                          std::greater<std::pair<ll, int>>> pq;
52          pq.push({0, start_node});
53
54          while (!pq.empty()) {
55              ll d = pq.top().first;
56              int u = pq.top().second;
57              pq.pop();
58
59              // Optimization: if we've found a better path already, skip
        this one.
61              if (d > distances[u]) {
62                  continue;
63              }
64
65              // Iterate through all outgoing edges of node u.
66              for (const auto& edge : adj[u]) {
67                  int weight = edge.first;
68                  int v = edge.second;
69
70                  // Relaxation step: if we found a shorter path to v

```

```

71     through u...
72         if (distances[u] + weight < distances[v]) {
73             // ...update the distance and predecessor...
74             distances[v] = distances[u] + weight;
75             predecessors[v] = u;
76             // ...and push the new, better path to the priority
77             queue.
78             pq.push({ distances[v], v });
79         }
80     }
81 }
82 /**
83  * @brief Reconstructs the shortest path to a given end node.
84  * @param end_node The destination node.
85  * @return A vector of integers representing the path from the start
86  * node to the end node.
87  */
88 std::vector<int> reconstruct_path(int end_node) {
89     std::vector<int> path;
90     for (int at = end_node; at != -1; at = predecessors[at]) {
91         path.push_back(at);
92     }
93     std::reverse(path.begin(), path.end());
94     if (!path.empty() && path[0] == 0) { // Assuming start_node is
95         always 0
96         return path;
97     }
98     return {}; // Return an empty path if no path was found.
99 };
100
101 // =====
102 // MAIN FUNCTION - DEMONSTRATION FOR PROBLEM 14 (SIMPLE GRAPH)
103 // =====
104 int main() {
105     int num_nodes = 5;
106     int start_node = 0; // Starting from node 'A' (index 0)
107
108     std::map<int, std::string> node_names = {{0, "A"}, {1, "B"}, {2, "C"},
109     {3, "D"}, {4, "E"}};
110
111     // --- Graph Creation for a Simple Graph ---
112     // Since it's a simple, undirected graph, for each edge (u, v), we
113     // add both
114     // u -> v and v -> u to the adjacency list.
115     Graph adj(num_nodes);
116
117     // Node A (0)
118     adj[0].push_back({10, 1}); // A -> B (weight 10)
119     adj[0].push_back({3, 2}); // A -> C (weight 3)
120     // Node B (1)
121     adj[1].push_back({10, 0}); // B -> A
122     adj[1].push_back({1, 2}); // B -> C
123     adj[1].push_back({2, 3}); // B -> D
124     // Node C (2)

```



```

123 adj[2].push_back({3, 0}); // C -> A
124 adj[2].push_back({1, 1}); // C -> B
125 adj[2].push_back({8, 3}); // C -> D
126 adj[2].push_back({2, 4}); // C -> E
127 // Node D (3)
128 adj[3].push_back({2, 1}); // D -> B
129 adj[3].push_back({8, 2}); // D -> C
130 adj[3].push_back({5, 4}); // D -> E
131 // Node E (4)
132 adj[4].push_back({2, 2}); // E -> C
133 adj[4].push_back({5, 3}); // E -> D
134
135 // --- Execution and Output ---
136
137 DijkstraSolver solver(num_nodes);
138 solver.solve(adj, start_node);
139
140 std::cout << "=====
141 << std::endl;
142 std::cout << "Dijkstra's Algorithm Report for Simple Graph (Problem
143 14)" << std::endl;
144 std::cout << "Source Node: '" << node_names[start_node] << "'" << std
145 ::endl;
146 std::cout << "=====
147 << std::endl;
148
149 for (int i = 0; i < num_nodes; ++i) {
150     std::cout << "\n--- Destination: '" << node_names[i] << "' ---"
151     << std::endl;
152     std::cout << "    Shortest Distance: ";
153     if (solver.distances[i] == std::numeric_limits<ll>::max()) {
154         std::cout << "UNREACHABLE" << std::endl;
155     } else {
156         std::cout << solver.distances[i] << std::endl;
157         std::vector<int> path = solver.reconstruct_path(i);
158         std::cout << "    Path: ";
159         if (path.empty()) {
160             std::cout << "N/A" << std::endl;
161         } else {
162             for (size_t j = 0; j < path.size(); ++j) {
163                 std::cout << node_names[path[j]] << (j == path.size()
164                 - 1 ? "" : " -> ");
165             }
166             std::cout << std::endl;
167         }
168     }
169 }
170
171 std::cout << "=====
172 << std::endl;
173
174 return 0;
175 }

```

Listing 2: Cài đặt Dijkstra bằng C++ sử dụng lớp và truy vết.

1.5 Phân tích Nâng cao

- Độ phức tạp: Với hàng đợi ưu tiên (binary heap), độ phức tạp của thuật toán là $O(|E| \log |V|)$.
- Trường hợp trọng số âm: Thuật toán Dijkstra sẽ cho kết quả sai. Cần dùng thuật toán Bellman-Ford ($O(|V| \cdot |E|)$) hoặc SPFA.

1.6 Kết luận cho Bài toán 14

Báo cáo đã phân tích, chứng minh, mô phỏng và cài đặt thành công thuật toán Dijkstra cho bài toán tìm đường đi ngắn nhất trên đơn đồ thị hữu hạn có trọng số không âm. Mã nguồn cung cấp có khả năng truy vết đường đi, một yêu cầu quan trọng trong các ứng dụng thực tế.

2 Bài toán 15: Thuật toán Dijkstra trên Đa đồ thị Hữu hạn

Dề bài 15

Let $G = (V, E)$ be a finite multigraph. Implement the Dijkstra's algorithm to find the shortest path problem on G .

(Cho $G = (V, E)$ là một đa đồ thị hữu hạn. Hãy cài đặt thuật toán Dijkstra để giải bài toán tìm đường đi ngắn nhất trên G .)

2.1 Phân tích và Đặc điểm của Đa đồ thị

- Định nghĩa Đa đồ thị (Multigraph): Là một loại đồ thị trong đó cho phép tồn tại nhiều hơn một cạnh giữa cùng một cặp đỉnh. Các cạnh này được gọi là cạnh song song (parallel edges). Mỗi cạnh song song có thể có trọng số riêng biệt.
- Tác động đến bài toán tìm đường đi ngắn nhất: Sự tồn tại của các cạnh song song đặt ra một lựa chọn hiển nhiên: nếu có nhiều con đường trực tiếp từ đỉnh u đến v , một thuật toán tìm đường đi tối ưu sẽ luôn ưu tiên cạnh có trọng số nhỏ nhất. Bất kỳ cạnh song song nào khác có trọng số lớn hơn đều trở nên không quan trọng (redundant) trong bối cảnh này.

Ví dụ, nếu có hai cạnh từ A đến B, một cạnh trọng số 5 và một cạnh trọng số 10, đường đi ngắn nhất sẽ luôn xem xét cạnh có trọng số 5.

2.2 Tính Áp dụng của Thuật toán Dijkstra

Nền tảng lý thuyết của thuật toán Dijkstra vẫn hoàn toàn đúng đắn và hiệu quả khi áp dụng cho đa đồ thị có trọng số không âm.

- Nguyên lý Cốt lõi Không đổi: Các nguyên lý về cấu trúc con tối ưu (một phần của đường đi ngắn nhất cũng là đường đi ngắn nhất) và chiến lược tham lam (luôn chọn đỉnh chưa xét có khoảng cách ước lượng nhỏ nhất) không bị ảnh hưởng bởi sự tồn tại của cạnh song song.
- Bước Nới lỏng (Relaxation) là Chìa khóa: Tính đúng đắn được bảo toàn nhờ bản chất của bước nới lỏng. Công thức nới lỏng cho một cạnh (u, v) là:

$$d[v] = \min(d[v], d[u] + w(u, v))$$

Khi thuật toán duyệt qua các cạnh kề của đỉnh u , nếu có nhiều cạnh nối đến cùng một đỉnh kề v , vòng lặp sẽ xử lý từng cạnh này một cách độc lập. Hàm \min sẽ tự động đảm bảo rằng chỉ có đường đi thông qua cạnh có trọng số nhỏ nhất mới có cơ hội cập nhật giá trị $d[v]$. Thuật toán xử lý đa đồ thị một cách tự nhiên mà không cần sửa đổi logic.

2.3 Mô phỏng từng bước trên Đa đồ thị

Để minh họa, ta xét một đa đồ thị bằng cách thêm các cạnh song song vào ví dụ trước.

- Cạnh song song B-D: Cạnh gốc $B \rightarrow D$ (trọng số 2), thêm cạnh mới $B \rightarrow D$ (trọng số 1).
- Cạnh song song A-C: Cạnh gốc $A \rightarrow C$ (trọng số 3), thêm cạnh mới $A \rightarrow C$ (trọng số 8).

Ta chạy thuật toán với đỉnh nguồn là A.

Bước	Chọn (u)	Tập S	Hành động và Cập nhật
0	-	\emptyset	Khởi tạo: $d = \{A : 0, \dots : \infty\}$, $PQ = \{(0, A)\}$
1	A	$\{A\}$	Lấy $(0, A)$. $\text{Relax } (A, B, 10) \rightarrow d[B] = 10$. $\text{Relax } (A, C, 3) \rightarrow d[C] = 3$. $\text{Relax } (A, C, 8) \rightarrow 0 + 8 \not< 3$. Không cập nhật. $PQ = \{(3, C), (10, B)\}$
2	C	$\{A, C\}$	Lấy $(3, C)$. $\text{Relax } (C, B, 1) \rightarrow 3 + 1 < d[B] = 10 \rightarrow d[B] = 4$. $\text{Relax } (C, D, 8) \rightarrow 3 + 8 < \infty \rightarrow d[D] = 11$. $\text{Relax } (C, E, 2) \rightarrow 3 + 2 < \infty \rightarrow d[E] = 5$. $PQ = \{(4, B), (5, E), (10, B), (11, D)\}$
3	B	$\{A, C, B\}$	Lấy $(4, B)$. $\text{Relax } (B, D, 2) \rightarrow 4 + 2 < d[D] = 11 \rightarrow d[D] = 6$. $\text{Relax } (B, D, 1) \rightarrow 4 + 1 < d[D] = 6 \rightarrow d[D] = 5$. $PQ = \{(5, D), (5, E), (6, D), (10, B), \dots\}$
4	D	$\{A, C, B, D\}$	Lấy $(5, D)$. Các đỉnh kề đã ở trong S hoặc không có đường đi tốt hơn. $PQ = \{(5, E), (6, D), \dots\}$
5	E	$\{A, C, B, D, E\}$	Lấy $(5, E)$. $\text{Relax } (E, D, 5) \rightarrow 5 + 5 \not< d[D] = 5$. Không cập nhật. Thuật toán kết thúc.

Kết quả cuối cùng: distances = {A: 0, B: 4, C: 3, D: 5, E: 5}. Kết quả này khác so với trường hợp đơn đồ thị, trong đó $d[D] = 6$. Sự khác biệt này là do thuật toán đã tận dụng được cạnh song song $B \rightarrow D$ có trọng số tốt hơn là 1.

2.4 Cài đặt và Biểu diễn Đa đồ thị

Sự thay đổi chính trong việc cài đặt không nằm ở logic thuật toán, mà ở cấu trúc dữ liệu được dùng để biểu diễn đồ thị.

2.4.1 Cài đặt bằng Python

Để biểu diễn các cạnh song song, ta sử dụng một dictionary trong đó mỗi khóa (đỉnh nguồn) ánh xạ tới một danh sách các tuple '(đỉnh kề, trọng số)'.

```
1 import heapq
2
3 def dijkstra_multigraph_python(graph: dict, all_nodes: set, start_node:
4     str):
5     """
6     Implements Dijkstra's algorithm for a multigraph.
7
8     This function calculates the shortest paths from a single source node
9     to all other
10    nodes in a graph that may contain parallel edges.
11
12    :param graph: A dictionary representing the multigraph.
13                  Keys are source nodes, and values are lists of (
14    neighbor, weight) tuples.
15                  Example: {'A': [('B', 10), ('B', 12)]}
16    :param all_nodes: A set containing all unique nodes in the graph.
17    This is necessary
18                      to initialize distances for nodes that may only be
19    destinations.
20    :param start_node: The starting node for the algorithm.
21    :return: A tuple containing two dictionaries: (distances,
22    predecessors).
23            'distances' maps each node to its shortest distance from the
24            source.
25            'predecessors' maps each node to its preceding node in the
26            shortest path.
27    """
28    # --- Initialization ---
29
30    # Initialize distances for all nodes to infinity, and the start node
31    to 0.
32    distances = {node: float('inf') for node in all_nodes}
33    distances[start_node] = 0
34
35    # This dictionary will store the path.
36    predecessors = {node: None for node in all_nodes}
37
38    # The priority queue stores tuples of (distance, node).
39    # heapq implements a min-heap, so it will always give us the item
40    with the smallest distance.
41    pq = [(0, start_node)]
42
43    # --- Main Loop ---
44
45    while pq:
46        # Get the node with the smallest distance from the priority queue
47        .
48        current_distance, current_node = heapq.heappop(pq)
49
50        # Optimization: If we have already found a shorter path to this
51        node, skip.
52        if current_distance > distances[current_node]:
53            continue
```

```

43     # --- Relaxation Step ---
44
45     # This is the key part for multigraphs. We iterate through the
46     # list of edges.
47     # If there are parallel edges, this loop will process all of them
48     .
49     if current_node in graph:
50         for neighbor, weight in graph[current_node]:
51             distance_through_u = current_distance + weight
52
53             # If we found a new, shorter path to the neighbor...
54             if distance_through_u < distances[neighbor]:
55                 # ...update the new shortest distance.
56                 distances[neighbor] = distance_through_u
57                 # ...record that we reached this neighbor via the
58                 current node.
59                 predecessors[neighbor] = current_node
60                 # ...and push the new path information to the
61                 priority queue.
62                 heapq.heappush(pq, (distance_through_u, neighbor))
63
64     return distances, predecessors
65
66 def reconstruct_path(predecessors: dict, start_node: str, end_node: str):
67     """
68     Reconstructs the shortest path from the predecessors dictionary.
69
70     :param predecessors: The dictionary mapping each node to its
71     predecessor.
72     :param start_node: The starting node of the path.
73     :param end_node: The ending node of the path.
74     :return: A list of nodes representing the path, or None if no path
75     exists.
76     """
77     path = []
78     current_node = end_node
79     # Backtrack from the end node to the start node.
80     while current_node is not None:
81         path.insert(0, current_node) # Insert at the beginning to build
82         the path in correct order
83         current_node = predecessors.get(current_node) # Use .get for
84         safety
85
86     # If the path starts with the start_node, it's a valid path.
87     if path and path[0] == start_node:
88         return path
89     else:
90         return None
91
92 # =====
93 # MAIN EXECUTION BLOCK - DEMONSTRATION FOR PROBLEM 15 (MULTIGRAPH)
94 # =====
95 if __name__ == "__main__":
96     # --- 1. Graph Creation for a Multigraph ---
97     # The data structure is a dictionary where keys are nodes and values
98     # are
99     # lists of (neighbor, weight) tuples. This allows for parallel edges.

```

```

92     multigraph = {
93         'A': [( 'B', 10), ( 'C', 3), ( 'C', 8)], # Parallel edge A->C
94         'B': [( 'A', 10), ( 'C', 1), ( 'D', 2), ( 'D', 1)], # Parallel edge B
->D
95         'C': [( 'A', 3), ( 'A', 8), ( 'B', 1), ( 'D', 8), ( 'E', 2)],
96         'D': [( 'B', 2), ( 'B', 1), ( 'C', 8), ( 'E', 5)],
97         'E': [( 'C', 2), ( 'D', 5)]
98     }
99
100     start_node = 'A'
101
102     # --- 2. Pre-processing: Get all unique nodes ---
103     # This is important for multigraphs where a node might only be a
destination.
104     all_nodes = set(multigraph.keys())
105     for node in multigraph:
106         for neighbor, weight in multigraph[node]:
107             all_nodes.add(neighbor)
108
109     # --- 3. Execution ---
110     # Run the Dijkstra algorithm on the defined graph.
111     distances, predecessors = dijkstra_multigraph_python(multigraph,
all_nodes, start_node)
112
113     # --- 4. Reporting Results ---
114     print("=" * 60)
115     print("Dijkstra's Algorithm Report for Multigraph (Problem 15)")
116     print(f"Source Node: '{start_node}'")
117     print("=" * 60)
118
119     # Sort nodes for consistent output order.
120     sorted_nodes = sorted(list(all_nodes))
121
122     for node in sorted_nodes:
123         print(f"\n--- Destination: '{node}' ---")
124         dist = distances.get(node)
125
126         print(f"    Shortest Distance: ", end="")
127         if dist == float('inf'):
128             print("UNREACHABLE")
129         else:
130             print(dist)
131
132         # Reconstruct and print the path
133         path = reconstruct_path(predecessors, start_node, node)
134         print("    Path: ", end="")
135         if path:
136             print(" -> ".join(path))
137         else:
138             # This case handles the source node itself, which has no
path to reconstruct.
139             print(f"'{node}' (Source Node)")
140
141     print("=" * 60)

```

Listing 3: Cài đặt Dijkstra cho đa đồ thị trong Python.

2.4.2 Cài đặt bằng C++

Với C++, cấu trúc dữ liệu ‘std::vector<std::vector<Edge>’ (danh sách kề) vốn đã hỗ trợ đa đồ thị một cách tự nhiên. Ta không cần thay đổi bất kỳ dòng code nào trong lớp ‘DijkstraSolver’ đã viết ở Bài toán 14. Sự khác biệt duy nhất là cách chúng ta điền dữ liệu vào đồ thị.

```
1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 #include <limits>
5 #include <string>
6 #include <map>
7 #include <algorithm>
8 #include <functional> // For std::greater
9
10 // Use long long to avoid overflow with long paths.
11 using ll = long long;
12 // Use type aliases for better readability.
13 using Edge = std::pair<int, int>; // {weight, neighbor_node}
14 using Graph = std::vector<std::vector<Edge>>;
15
16 /**
17  * @class DijkstraSolver
18  * @brief A class to compute the shortest paths from a single source
19  *        using Dijkstra's algorithm.
20  *
21  * This class encapsulates the data structures and logic needed for the
22  * algorithm,
23  * including storing distances, predecessors for path reconstruction, and
24  * the main solving logic.
25  * It is designed to work with graphs represented by an adjacency list.
26  */
27 class DijkstraSolver {
28 public:
29     std::vector<ll> distances;
30     std::vector<int> predecessors;
31     int num_nodes;
32
33     /**
34      * @brief Constructs a DijkstraSolver for a graph with a given number
35      *        of nodes.
36      * @param n The number of nodes in the graph.
37      */
38     DijkstraSolver(int n) : num_nodes(n) {
39         // Initialize distances to infinity.
40         distances.resize(n, std::numeric_limits<ll>::max());
41         // Initialize predecessors to -1 (indicating no predecessor).
42         predecessors.resize(n, -1);
43     }
44
45     /**
46      * @brief Executes Dijkstra's algorithm on the given graph from a
47      *        start node.
48      * @param adj The graph, represented as an adjacency list.
49      * @param start_node The index of the source node.
50      */
51     void solve(const Graph& adj, int start_node) {
```



```

47 // The distance from the start node to itself is 0.
48 distances[start_node] = 0;
49
50 // Min-heap: stores {distance, node}.
51 // std::greater is used to make the priority_queue a min-heap.
52 std::priority_queue<std::pair<ll, int>,
53                     std::vector<std::pair<ll, int>>,
54                     std::greater<std::pair<ll, int>>> pq;
55 pq.push({0, start_node});
56
57 while (!pq.empty()) {
58     ll d = pq.top().first;
59     int u = pq.top().second;
60     pq.pop();
61
62     // Optimization: if we've found a better path already, skip
this one.
63     // This is crucial for correctness and performance.
64     if (d > distances[u]) {
65         continue;
66     }
67
68     // Iterate through all outgoing edges of node u.
69     // Use const auto& to avoid unnecessary copies.
70     for (const auto& edge : adj[u]) {
71         int weight = edge.first;
72         int v = edge.second;
73
74         // Relaxation step: if we found a shorter path to v
through u...
75         if (distances[u] + weight < distances[v]) {
76             // ...update the distance and predecessor...
77             distances[v] = distances[u] + weight;
78             predecessors[v] = u;
79             // ...and push the new, better path to the priority
queue.
80             pq.push({distances[v], v});
81         }
82     }
83 }
84
85 /**
86  * @brief Reconstructs the shortest path to a given end node.
87  * @param end_node The destination node.
88  * @return A vector of integers representing the path from the start
node to the end node.
89  *         Returns an empty vector if no path exists.
90  */
91
92 std::vector<int> reconstruct_path(int end_node) {
93     std::vector<int> path;
94     // Backtrack from the end node using the predecessors array.
95     for (int at = end_node; at != -1; at = predecessors[at]) {
96         path.push_back(at);
97     }
98     // The path is constructed in reverse, so we need to reverse it
back.
99     std::reverse(path.begin(), path.end());

```

```

100         // If the path does not start with the source node, it's not a
101         valid path.
102         if (!path.empty() && path[0] == 0) { // Assuming start_node is
103         always 0 in this context
104             return path;
105         }
106         return {}; // Return an empty path if no path was found.
107     };
108
109     // =====
110     // MAIN FUNCTION - DEMONSTRATION FOR PROBLEM 15 (MULTIGRAPH)
111     // =====
112     int main() {
113         int num_nodes = 5;
114         int start_node = 0; // Starting from node 'A' (index 0)
115
116         // Map integer indices to human-readable names for clear output.
117         std::map<int, std::string> node_names = {{0, "A"}, {1, "B"}, {2, "C"},
118         {3, "D"}, {4, "E"}};
119
120         // --- Graph Creation for a Multigraph ---
121         // The adjacency list naturally handles multigraphs by allowing
122         // multiple
123         // entries for the same neighbor in the list.
124         Graph adj(num_nodes);
125
126         // Add ALL edges, including parallel ones.
127         // Node A (0)
128         adj[0].push_back({10, 1}); // A -> B
129         adj[0].push_back({3, 2}); // A -> C (edge 1)
130         adj[0].push_back({8, 2}); // A -> C (edge 2, parallel)
131
132         // Node B (1)
133         adj[1].push_back({10, 0});
134         adj[1].push_back({1, 2}); // B -> C
135         adj[1].push_back({2, 3}); // B -> D (edge 1)
136         adj[1].push_back({1, 3}); // B -> D (edge 2, parallel, better weight)
137     )
138
139     // Node C (2)
140     adj[2].push_back({3, 0}); // C -> A (edge 1)
141     adj[2].push_back({8, 0}); // C -> A (edge 2, parallel)
142     adj[2].push_back({1, 1});
143     adj[2].push_back({8, 3});
144     adj[2].push_back({2, 4});
145
146     // Node D (3)
147     adj[3].push_back({2, 1}); // D -> B (edge 1)
148     adj[3].push_back({1, 1}); // D -> B (edge 2, parallel)
149     adj[3].push_back({8, 2});
150     adj[3].push_back({5, 4});
151
152     // Node E (4)
153     adj[4].push_back({2, 2});
154     adj[4].push_back({5, 3});

```

```

153 // --- Execution and Output ---
154
155 // Create an instance of the solver.
156 DijkstraSolver solver(num_nodes);
157
158 // Run the algorithm.
159 solver.solve(adj, start_node);
160
161 // Print a detailed report of the results.
162 std::cout << "=====
163 << std::endl;
164 std::cout << "Dijkstra's Algorithm Report for Multigraph (Problem 15)
165 " << std::endl;
166 std::cout << "Source Node: '" << node_names[start_node] << "' << std
167 ::endl;
168 std::cout << "=====
169 << std::endl;
170
171 for (int i = 0; i < num_nodes; ++i) {
172     std::cout << "\n--- Destination: '" << node_names[i] << "' ---"
173     << std::endl;
174     std::cout << "    Shortest Distance: ";
175     if (solver.distances[i] == std::numeric_limits<ll>::max()) {
176         std::cout << "UNREACHABLE" << std::endl;
177     } else {
178         std::cout << solver.distances[i] << std::endl;
179
180         // Reconstruct and print the path
181         std::vector<int> path = solver.reconstruct_path(i);
182         std::cout << "    Path: ";
183         if (path.empty()) {
184             std::cout << "N/A" << std::endl;
185         } else {
186             for (size_t j = 0; j < path.size(); ++j) {
187                 std::cout << node_names[path[j]] << (j == path.size()
188                 - 1 ? "" : " -> ");
189             }
190             std::cout << std::endl;
191         }
192     }
193 }
194
195 std::cout << "=====
196 << std::endl;
197
198 return 0;
199 }

```

Listing 4: Xây dựng đa đồ thị trong C++. Logic thuật toán không đổi.

2.5 Kết luận cho Bài toán 15

Thuật toán Dijkstra hoàn toàn có thể áp dụng để giải bài toán tìm đường đi ngắn nhất trên đa đồ thị mà không cần sửa đổi logic cốt lõi. Sự thay đổi chính nằm ở tầng cài đặt, cụ thể là cách biểu diễn đồ thị để có thể lưu trữ các cạnh song song. Một khi đã có cấu trúc dữ liệu phù hợp (như danh sách kề), thuật toán sẽ tự động xử lý và chọn ra cạnh tối ưu nhất giữa hai đỉnh bất kỳ nhờ vào cơ chế nới lỏng. Độ phức tạp tính

toán vẫn là $O(|E|\log|V|)$, trong đó $|E|$ là tổng số cạnh trong đa đồ thị, bao gồm cả các cạnh song song.

3 Bài toán 16: Thuật toán Dijkstra trên Đồ thị Tổng quát

Dề bài 16

Let $G = (V, E)$ be a general graph. Implement the Dijkstra's algorithm to find the shortest path problem on G .

(Cho $G = (V, E)$ là một đồ thị tổng quát. Hãy cài đặt thuật toán Dijkstra để giải bài toán tìm đường đi ngắn nhất trên G .)

3.1 Phân tích và Định nghĩa Đồ thị Tổng quát

Đồ thị tổng quát (General Graph) là loại đồ thị có ít ràng buộc nhất. Nó có thể bao gồm:

- Cạnh song song (Parallel Edges): Giống như đa đồ thị (Bài toán 15).
- Khuyên (Self-loops): Các cạnh nối một đỉnh với chính nó, ví dụ cạnh (u, u) .
- Trọng số âm (Negative Weights): Không có ràng buộc nào về trọng số của cạnh. Chúng có thể là số dương, số không, hoặc số âm.

Đây chính là điểm khác biệt quan trọng và phức tạp nhất. Việc đề bài yêu cầu "cài đặt thuật toán Dijkstra" trên một đồ thị tổng quát đã tạo ra một mâu thuẫn lý thuyết sâu sắc.

3.2 Ảnh hưởng đến Thuật toán Dijkstra – Một Phân tích Phê bình

Thuật toán Dijkstra được xây dựng dựa trên một giả định cốt lõi: một khi một đỉnh được chọn và đưa vào tập S (các đỉnh đã có khoảng cách cuối cùng), khoảng cách đó sẽ không bao giờ có thể được cải thiện. Giả định này chỉ đúng khi tất cả các trọng số cạnh là không âm.

3.2.1 Xử lý các Đặc điểm của Đồ thị Tổng quát

- Cạnh song song: Như đã phân tích ở Bài toán 15, thuật toán Dijkstra xử lý các cạnh này một cách tự nhiên.
- Khuyên (Self-loops):
 - Nếu khuyên (u, u) có trọng số $w \geq 0$, nó không ảnh hưởng đến thuật toán. Bước nối lỏng sẽ là $d[u] = \min(d[u], d[u] + w)$. Vì $d[u] + w \geq d[u]$, khoảng cách sẽ không bao giờ được cập nhật.

- Nếu khuyên (u, u) có trọng số $w < 0$, nó tạo ra một chu trình âm có độ dài 1. Không có đường đi ngắn nhất trong một đồ thị chứa chu trình âm có thể đến được từ đỉnh nguồn.
- Trọng số âm (Negative Weights): Đây là điểm mà Dijkstra thất bại. Chiến lược tham lam của Dijkstra sụp đổ.

Ví dụ kinh điển về sự thất bại của Dijkstra: Xét đồ thị: $S \rightarrow A$ (trọng số 5), $S \rightarrow B$ (trọng số 2), $A \rightarrow B$ (trọng số -4).

1. Dijkstra khởi tạo $d[S] = 0, d[A] = \infty, d[B] = \infty$. $PQ = \{(0, S)\}$.
2. Lấy $(0, S)$. Relax cạnh kề: $d[A] = 5, d[B] = 2$. $PQ = \{(2, B), (5, A)\}$.
3. Lựa chọn tham lam: Lấy $(2, B)$. Thuật toán "chốt" khoảng cách ngắn nhất đến B là 2 và thêm B vào tập S .
4. Lấy $(5, A)$. Relax cạnh kề: $A \rightarrow B$ (trọng số -4). Khoảng cách mới đến B là $d[A] + (-4) = 5 - 4 = 1$.
5. Quá muộn! Thuật toán đã "chốt" $d[B] = 2$. Nó sẽ không cập nhật lại nữa. Kết quả sai lầm là $d[B] = 2$, trong khi đường đi ngắn nhất thực sự là $S \rightarrow A \rightarrow B$ với độ dài là 1.

3.3 Cài đặt (Với Giả định Bất buộc)

Do những phân tích trên, không thể cài đặt một thuật toán Dijkstra đúng đắn cho một đồ thị tổng quát thực sự (có thể chứa trọng số âm).

Để hoàn thành yêu cầu của đề bài, chúng ta phải đưa ra một giả định bất buộc: "Đồ thị tổng quát được đề cập trong bài toán này được giới hạn chỉ có các trọng số không âm."

Với giả định này, bài toán trở nên tương tự Bài toán 15. Mã nguồn cho đa đồ thị cũng sẽ xử lý được các khuyên có trọng số không âm một cách tự nhiên.

3.3.1 Cài đặt bằng Python (Giả định trọng số không âm)

Mã nguồn giống hệt với Bài toán 15. Ta chỉ cần thêm khuyên vào dữ liệu đầu vào để minh họa.

```

1 import heapq
2
3 def dijkstra_general_graph(graph: dict, all_nodes: set, start_node: str):
4     """
5     Implements Dijkstra's algorithm.
6
7     WARNING: This implementation is ONLY correct if all edge weights in
8     the
9     'general graph' are non-negative. It can handle parallel edges and
10    non-negative self-loops, but it will fail with negative weights.
11
12    :param graph: A dict mapping a node to a list of (neighbor, weight)
13    tuples.
14    :param all_nodes: A set containing all unique nodes in the graph.
15    :param start_node: The starting node for the algorithm.

```

```

14     :return: A tuple containing two dictionaries: (distances ,
15     predecessors).
16     """
17     # --- Initialization ---
18     distances = {node: float('inf') for node in all_nodes}
19     distances[start_node] = 0
20     predecessors = {node: None for node in all_nodes}
21     pq = [(0, start_node)]
22
23     # --- Main Loop ---
24     while pq:
25         current_distance, current_node = heapq.heappop(pq)
26
27         if current_distance > distances[current_node]:
28             continue
29
30         # --- Relaxation Step ---
31         # The loop structure naturally handles parallel edges and self-
32         loops.
33         if current_node in graph:
34             for neighbor, weight in graph[current_node]:
35                 # Critical assumption: weight >= 0.
36                 # If weight were negative, this greedy logic would be
37                 flawed.
38                 distance_through_u = current_distance + weight
39
40                 if distance_through_u < distances[neighbor]:
41                     distances[neighbor] = distance_through_u
42                     predecessors[neighbor] = current_node
43                     heapq.heappush(pq, (distance_through_u, neighbor))
44
45     return distances, predecessors
46
47 def reconstruct_path(predecessors: dict, start_node: str, end_node: str):
48     """
49     Reconstructs the shortest path from the predecessors dictionary.
50     """
51     path = []
52     current_node = end_node
53     while current_node is not None:
54         path.insert(0, current_node)
55         current_node = predecessors.get(current_node)
56
57     if path and path[0] == start_node:
58         return path
59     else:
60         return None
61
62 # =====
63 # MAIN EXECUTION BLOCK - DEMONSTRATION FOR PROBLEM 16 (GENERAL GRAPH)
64 # =====
65 if __name__ == "__main__":
66     # --- 1. Graph Creation for a General Graph ---
67     # This graph includes parallel edges and a self-loop.
68     # CRITICAL ASSUMPTION: All weights are non-negative for Dijkstra to
69     be valid.
70     general_graph = {
71         'A': [('B', 10), ('C', 3)],

```

```

68         'B': [( 'D', 2), ( 'D', 1)], # Parallel edge B->D
69         'C': [( 'A', 3), ( 'D', 8), ( 'E', 2)],
70         'D': [( 'E', 5)],
71         'E': [( 'C', 1), ( 'E', 4)] # Self-loop E->E with non-negative
weight
72     }
73
74     start_node = 'A'
75
76     # --- 2. Pre-processing: Get all unique nodes ---
77     all_nodes = set(general_graph.keys())
78     for node in general_graph:
79         for neighbor, weight in general_graph[node]:
80             all_nodes.add(neighbor)
81
82     # --- 3. Execution ---
83     distances, predecessors = dijkstra_general_graph(general_graph,
all_nodes, start_node)
84
85     # --- 4. Reporting Results ---
86     print("=" * 70)
87     print("Dijkstra's Algorithm Report for General Graph (Problem 16)")
88     print(f"Source Node: '{start_node}'")
89     print("-" * 70)
90     print("WARNING: These results are correct ONLY under the assumption
that")
91     print("        all edge weights in the graph are non-negative.")
92     print("        Dijkstra's algorithm fails with negative edge weights
.")
93     print("=" * 70)
94
95     sorted_nodes = sorted(list(all_nodes))
96
97     for node in sorted_nodes:
98         print(f"\n--- Destination: '{node}' ---")
99         dist = distances.get(node)
100
101         print(f"    Shortest Distance: ", end="")
102         if dist == float('inf'):
103             print("UNREACHABLE")
104         else:
105             print(dist)
106
107         path = reconstruct_path(predecessors, start_node, node)
108         print("    Path: ", end="")
109         if path:
110             print(" -> ".join(path))
111         else:
112             print(f"'{node}' (Source Node)")
113
114     print("=" * 70)

```

Listing 5: Cài đặt Dijkstra cho đồ thị tổng quát (chỉ trọng số không âm).

3.3.2 Cài đặt bằng C++ (Giả định trọng số không âm)

Tương tự, lớp ‘DijkstraSolver’ từ các bài trước vẫn hoạt động tốt dưới giả định trọng số không âm. Ta chỉ cần thêm các loại cạnh của đồ thị tổng quát vào lúc khởi tạo.

```

1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 #include <limits>
5 #include <string>
6 #include <map>
7 #include <algorithm>
8 #include <functional> // For std::greater
9
10 // Use long long to avoid overflow with long paths.
11 using ll = long long;
12 // Use type aliases for better readability.
13 using Edge = std::pair<int, int>; // {weight, neighbor_node}
14 using Graph = std::vector<std::vector<Edge>>;
15
16 /**
17  * @class DijkstraSolver
18  * @brief A class to compute shortest paths from a single source using
19  *        Dijkstra's algorithm.
20  *
21  * This implementation is suitable for graphs with non-negative edge
22  * weights.
23  * It can handle parallel edges and self-loops as long as their weights
24  * are not negative.
25  */
26 class DijkstraSolver {
27 public:
28     std::vector<ll> distances;
29     std::vector<int> predecessors;
30     int num_nodes;
31
32     /**
33      * @brief Constructs a DijkstraSolver for a graph with a given number
34      *        of nodes.
35      * @param n The number of nodes in the graph.
36      */
37     DijkstraSolver(int n) : num_nodes(n) {
38         distances.resize(n, std::numeric_limits<ll>::max());
39         predecessors.resize(n, -1); // -1 indicates no predecessor
40     }
41
42     /**
43      * @brief Executes Dijkstra's algorithm.
44      * @warning This method is ONLY correct for graphs with non-negative
45      *        edge weights.
46      * @param adj The graph, represented as an adjacency list.
47      * @param start_node The index of the source node.
48      */
49     void solve(const Graph& adj, int start_node) {
50         distances[start_node] = 0;
51
52         std::priority_queue<std::pair<ll, int>,
53                             std::vector<std::pair<ll, int>>,
54                             std::greater<std::pair<ll, int>>> pq;
55         pq.push({0, start_node});
56
57         while (!pq.empty()) {
58             ll d = pq.top().first;

```



```

54         int u = pq.top().second;
55         pq.pop();
56
57         if (d > distances[u]) {
58             continue;
59         }
60
61         for (const auto& edge : adj[u]) {
62             int weight = edge.first;
63             int v = edge.second;
64
65             // The core logic of Dijkstra's algorithm.
66             // Assumes 'weight' is non-negative.
67             if (distances[u] + weight < distances[v]) {
68                 distances[v] = distances[u] + weight;
69                 predecessors[v] = u;
70                 pq.push({distances[v], v});
71             }
72         }
73     }
74 }
75
76 /**
77  * @brief Reconstructs the shortest path to a given end node.
78  * @param end_node The destination node.
79  * @return A vector of integers representing the path.
80  */
81 std::vector<int> reconstruct_path(int end_node) {
82     std::vector<int> path;
83     for (int at = end_node; at != -1; at = predecessors[at]) {
84         path.push_back(at);
85     }
86     std::reverse(path.begin(), path.end());
87
88     if (!path.empty() && path[0] == 0) { // Assuming start_node is
always 0
89         return path;
90     }
91     return {};
92 }
93 };
94
95 // =====
96 // MAIN FUNCTION - DEMONSTRATION FOR PROBLEM 16 (GENERAL GRAPH)
97 // =====
98 int main() {
99     int num_nodes = 5;
100     int start_node = 0;
101
102     std::map<int, std::string> node_names = {{0, "A"}, {1, "B"}, {2, "C"},
{3, "D"}, {4, "E"}};
103
104     // --- Graph Creation for a General Graph ---
105     // This graph includes parallel edges and a self-loop.
106     // CRITICAL ASSUMPTION: All weights are non-negative.
107     Graph adj(num_nodes);
108
109     // Node A (0)

```

```

110 adj[0].push_back({10, 1}); // A -> B
111 adj[0].push_back({3, 2}); // A -> C
112 // Node B (1)
113 adj[1].push_back({2, 3}); // B -> D (edge 1)
114 adj[1].push_back({1, 3}); // B -> D (parallel edge with better
weight)
115 // Node C (2)
116 adj[2].push_back({3, 0});
117 adj[2].push_back({8, 3});
118 adj[2].push_back({2, 4});
119 // Node D (3)
120 adj[3].push_back({5, 4});
121 // Node E (4)
122 adj[4].push_back({1, 2});
123 adj[4].push_back({4, 4}); // E -> E (self-loop with non-negative
weight)
124
125 // --- Execution and Output ---
126
127 DijkstraSolver solver(num_nodes);
128 solver.solve(adj, start_node);
129
130 std::cout << "
=====
131 std::endl;
132 std::cout << "Dijkstra's Algorithm Report for General Graph (Problem
16)" << std::endl;
133 std::cout << "Source Node: '" << node_names[start_node] << "'" << std
::endl;
134 std::cout << "WARNING: Results are correct ONLY under the assumption
that" << std::endl;
135 std::cout << "                all edge weights are non-negative." << std::
endl;
136 std::cout << "
=====
137 std::endl;
138
139 for (int i = 0; i < num_nodes; ++i) {
140     std::cout << "\n--- Destination: '" << node_names[i] << "'" << "
---"
141 << std::endl;
142     std::cout << "    Shortest Distance: ";
143     if (solver.distances[i] == std::numeric_limits<ll>::max()) {
144         std::cout << "UNREACHABLE" << std::endl;
145     } else {
146         std::cout << solver.distances[i] << std::endl;
147         std::vector<int> path = solver.reconstruct_path(i);
148         std::cout << "    Path: ";
149         if (path.empty()) {
150             std::cout << "N/A" << std::endl;
151         } else {
152             for (size_t j = 0; j < path.size(); ++j) {
153                 std::cout << node_names[path[j]] << (j == path.size()
- 1 ? "" : " -> ");
154             }
155             std::cout << std::endl;
156         }
157     }
158 }
159

```

```

156     std::cout << "
157     _____" <<
158     std::endl;
159     return 0;
160 }

```

Listing 6: Xây dựng đồ thị tổng quát trong C++ (chỉ trọng số không âm).

3.4 Phân tích và Mâu thuẫn Lý thuyết

Một đồ thị tổng quát (General Graph) có thể chứa cạnh song song, khuyên, và quan trọng nhất là trọng số âm. Yêu cầu "cài đặt thuật toán Dijkstra" trên một đồ thị như vậy đã tạo ra một mâu thuẫn lý thuyết, vì thuật toán Dijkstra không đảm bảo tính đúng đắn khi có sự tồn tại của trọng số âm.

- Thất bại của Dijkstra: Chiến lược tham lam của Dijkstra "chốt" khoảng cách của một đỉnh quá sớm, bỏ qua khả năng có một đường đi dài hơn về số cạnh nhưng ngắn hơn về tổng trọng số (thông qua một cạnh âm) được phát hiện sau đó.
- Giả định để tiếp tục: Để hoàn thành yêu cầu đề bài, ta phải giả định rằng "đồ thị tổng quát" được giới hạn chỉ có trọng số không âm. Dưới giả định này, mã nguồn từ Bài toán 15 (đa đồ thị) có thể được áp dụng.

Tuy nhiên, một báo cáo đầy đủ cần phải trình bày các giải pháp đúng đắn cho bài toán tổng quát thực sự.

3.5 Thuật toán Thay thế cho Đồ thị Tổng quát Thực sự

Khi một đồ thị có thể chứa trọng số âm, ta phải sử dụng các thuật toán mạnh mẽ hơn.

3.5.1 Thuật toán Bellman-Ford

Đây là thuật toán tiêu chuẩn cho bài toán tìm đường đi ngắn nhất từ một đỉnh nguồn trong đồ thị có thể có trọng số âm.

Nguyên lý hoạt động: Bellman-Ford thực hiện nới lỏng (relax) tất cả các cạnh của đồ thị, và lặp lại quá trình này $|V| - 1$ lần. Sau $|V| - 1$ lượt, nếu không có chu trình âm, thuật toán đảm bảo tìm được đường đi ngắn nhất. Một lượt lặp thứ $|V|$ được dùng để phát hiện chu trình âm.

```

1 def bellman_ford(graph: dict, all_nodes: set, start_node: str):
2     """
3     Implements the Bellman-Ford algorithm.
4     Handles negative weights and detects negative cycles.
5
6     :param graph: Adjacency list representation {'u': [('v', w), ...]}
7     :param all_nodes: A set of all unique node identifiers.
8     :param start_node: The source node.
9     :return: A tuple (distances, predecessors). If a negative cycle is
10             found,
11             distances for affected nodes are set to -infinity.

```

```

11 """
12 num_nodes = len(all_nodes)
13 distances = {node: float('inf') for node in all_nodes}
14 predecessors = {node: None for node in all_nodes}
15 distances[start_node] = 0
16
17 # 1. Relax edges |V| - 1 times
18 for _ in range(num_nodes - 1):
19     for u in graph:
20         for v, w in graph[u]:
21             if distances[u] != float('inf') and distances[u] + w <
distances[v]:
22                 distances[v] = distances[u] + w
23                 predecessors[v] = u
24
25 # 2. Check for negative-weight cycles
26 # If we can still relax an edge, it must be part of a negative cycle.
27 for u in graph:
28     for v, w in graph[u]:
29         if distances[u] != float('inf') and distances[u] + w <
distances[v]:
30             # Mark as part of a negative cycle
31             distances[v] = float('-inf')
32
33 return distances, predecessors
34
35 # --- Example with negative weights and a negative cycle ---
36 # graph_neg = {
37 #     'S': [('A', 5), ('C', 2)],
38 #     'A': [('B', 2)],
39 #     'C': [('A', -4)], # Negative edge
40 #     'D': [('E', -1)], 'E': [('F', -1)], 'F': [('D', -1)] # Negative
cycle
41 # }
42 # all_nodes_neg = {'S', 'A', 'B', 'C', 'D', 'E', 'F'}
43 # dist, pred = bellman_ford(graph_neg, all_nodes_neg, 'S')
44 # print(f"Bellman-Ford distances: {dist}")
45 # Expected output: S:0, A: -2, B:0, C:2, D:-inf, E:-inf, F:-inf

```

Listing 7: Cài đặt Bellman-Ford trong Python, có phát hiện chu trình âm.

```

1 // Define INF and N_INF for readability
2 const ll INF = std::numeric_limits<ll>::max();
3 const ll N_INF = std::numeric_limits<ll>::min();
4
5 class BellmanFordSolver {
6 public:
7     std::vector<ll> distances;
8     int num_nodes;
9
10    BellmanFordSolver(int n) : num_nodes(n) {
11        distances.resize(n, INF);
12    }
13
14    void solve(const Graph& adj, int start_node) {
15        distances[start_node] = 0;

```

```

16
17 // Relax all edges |V| - 1 times
18 for (int i = 0; i < num_nodes - 1; ++i) {
19     for (int u = 0; u < num_nodes; ++u) {
20         if (distances[u] == INF) continue;
21         for (const auto& edge : adj[u]) {
22             int weight = edge.first;
23             int v = edge.second;
24             if (distances[u] + weight < distances[v]) {
25                 distances[v] = distances[u] + weight;
26             }
27         }
28     }
29 }
30
31 // Run a final iteration to detect negative cycles
32 for (int i = 0; i < num_nodes - 1; ++i) { // Propagate N_INF
33     for (int u = 0; u < num_nodes; ++u) {
34         if (distances[u] == INF) continue;
35         for (const auto& edge : adj[u]) {
36             int weight = edge.first;
37             int v = edge.second;
38             if (distances[u] + weight < distances[v]) {
39                 // Node is part of or reachable from a negative
cycle
40                 distances[v] = N_INF;
41             }
42         }
43     }
44 }
45 }
46 };

```

Listing 8: Cài đặt Bellman-Ford trong C++, có phát hiện chu trình âm.

3.5.2 Thuật toán SPFA (Shortest Path Faster Algorithm)

SPFA là một biến thể của Bellman-Ford. Thay vì mù quáng nối lỏng tất cả các cạnh, nó chỉ nối lỏng các cạnh từ những đỉnh vừa được cập nhật khoảng cách. Nó sử dụng một hàng đợi (queue) để quản lý các đỉnh này.

Nguyên lý hoạt động: SPFA hoạt động tương tự BFS. Khi khoảng cách đến một đỉnh v được cải thiện, v sẽ được thêm vào hàng đợi (nếu nó chưa có trong đó). Thuật toán kết thúc khi hàng đợi rỗng. Để phát hiện chu trình âm, ta đếm số lần mỗi đỉnh được đưa vào hàng đợi; nếu một đỉnh được đưa vào $\geq |V|$ lần, nó phải nằm trong một chu trình âm.

```

1 from collections import deque
2
3 def spfa(graph: dict, all_nodes: set, start_node: str):
4     """
5     Implements the Shortest Path Faster Algorithm (SPFA).

```

```

6     Handles negative weights and detects negative cycles.
7     """
8     num_nodes = len(all_nodes)
9     distances = {node: float('inf') for node in all_nodes}
10    predecessors = {node: None for node in all_nodes}
11    count = {node: 0 for node in all_nodes}
12    in_queue = {node: False for node in all_nodes}
13    queue = deque()
14
15    distances[start_node] = 0
16    queue.append(start_node)
17    in_queue[start_node] = True
18    count[start_node] = 1
19
20    while queue:
21        u = queue.popleft()
22        in_queue[u] = False
23
24        if u not in graph: continue
25
26        for v, w in graph[u]:
27            if distances[u] + w < distances[v]:
28                distances[v] = distances[u] + w
29                predecessors[v] = u
30
31                if not in_queue[v]:
32                    queue.append(v)
33                    in_queue[v] = True
34                    count[v] += 1
35                    if count[v] >= num_nodes:
36                        # Negative cycle detected
37                        print(f"Negative cycle detected involving node {v}")
38
39                return None, None # Indicate error
40
41    return distances, predecessors

```

Listing 9: Cài đặt SPFA trong Python.

```

1 #include <queue> // For std::queue
2
3 class SpfaSolver {
4 public:
5     std::vector<ll> distances;
6     int num_nodes;
7
8     SpfaSolver(int n) : num_nodes(n) {
9         distances.resize(n, INF);
10    }
11
12    // Returns true if no negative cycle is detected, false otherwise.
13    bool solve(const Graph& adj, int start_node) {
14        std::vector<int> count(num_nodes, 0);
15        std::vector<bool> in_queue(num_nodes, false);
16        std::queue<int> q;
17

```

```

18     distances[start_node] = 0;
19     q.push(start_node);
20     in_queue[start_node] = true;
21     count[start_node]++;
22
23     while (!q.empty()) {
24         int u = q.front();
25         q.pop();
26         in_queue[u] = false;
27
28         for (const auto& edge : adj[u]) {
29             int weight = edge.first;
30             int v = edge.second;
31
32             if (distances[u] != INF && distances[u] + weight <
distances[v]) {
33                 distances[v] = distances[u] + weight;
34                 if (!in_queue[v]) {
35                     q.push(v);
36                     in_queue[v] = true;
37                     count[v]++;
38                     if (count[v] >= num_nodes) {
39                         // Negative cycle detected
40                         return false;
41                     }
42                 }
43             }
44         }
45     }
46     return true; // Success
47 }
48 };

```

Listing 10: Cài đặt SPFA trong C++.

3.6 Kết luận cho Bài toán 16

1. Kết luận lý thuyết: Thuật toán Dijkstra không thể được áp dụng một cách an toàn cho một đồ thị tổng quát thực sự vì nó có thể cho kết quả sai khi có trọng số âm.
2. Kết luận thực hành: Để "cài đặt Dijkstra" theo yêu cầu, chúng ta đã phải đưa ra giả định rằng tất cả các trọng số cạnh đều không âm. Dưới giả định này, việc cài đặt tương tự như với đa đồ thị.
3. Giải pháp đúng đắn: Đối với đồ thị tổng quát thực sự (có thể có trọng số âm), Bellman-Ford là lựa chọn tiêu chuẩn, an toàn và có khả năng phát hiện chu trình âm. SPFA là một giải pháp thay thế, thường nhanh hơn trong thực tế nhưng có cùng độ phức tạp trong trường hợp xấu nhất.

3.7 Kết luận cho Bài toán 16

Bài toán 16 đặt ra một câu hỏi quan trọng về việc lựa chọn công cụ phù hợp.

1. Kết luận lý thuyết: Thuật toán Dijkstra không thể được áp dụng một cách an toàn cho một đồ thị tổng quát thực sự vì nó có thể cho kết quả sai hoặc không xác định khi có trọng số âm hoặc chu trình âm.
2. Kết luận thực hành: Để "cài đặt Dijkstra" theo yêu cầu, chúng ta đã phải đưa ra một giả định mạnh mẽ là tất cả các trọng số cạnh đều không âm. Dưới giả định này, việc cài đặt tương tự như với đa đồ thị, vì các khuyên có trọng số không âm không ảnh hưởng đến kết quả.
3. Bài học rút ra: Bài toán này nhấn mạnh tầm quan trọng của việc phân tích các điều kiện tiên quyết của một thuật toán trước khi áp dụng. Đối với đồ thị tổng quát có thể có trọng số âm, Bellman-Ford là lựa chọn đúng đắn và an toàn.