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

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	-	$\emptyset$	$d = \{A:0, \cdots:\infty\}$, 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), ...}
3	B	$\{A,C,B\}$	$d[D] = 6$. PQ={(5,E), (6,D), ...}
4	E	$\{A,C,B,E\}$	Không cập nhật. PQ={(6,D), ...}
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

```python
import heapq

def dijkstra_simple_graph(graph: dict, start_node: str):
    """
    Implements Dijkstra's algorithm for a simple graph.

    This function calculates the shortest paths from a single source node to all other
    nodes in a simple graph with non-negative edge weights.

    :param graph: A dictionary of dictionaries representing the graph.
                  Example: {'A': {'B': 10, 'C': 3}, 'B': {'A': 10}}
    :param start_node: The starting node for the algorithm.
    :return: A tuple containing two dictionaries: (distances, predecessors).
             'distances' maps each node to its shortest distance from the source.
             'predecessors' maps each node to its preceding node in the shortest path.
    """
    # --- Initialization ---

    # Initialize distances for all nodes to infinity, and the start node to 0.
    distances = {node: float('inf') for node in graph}
    distances[start_node] = 0

    # This dictionary will store the path for reconstruction.
    predecessors = {node: None for node in graph}

    # The priority queue stores tuples of (distance, node).
    pq = [(0, start_node)]

    # --- Main Loop ---

    while pq:
```

```python
        # Get the node with the smallest distance from the priority queue.
        current_distance, current_node = heapq.heappop(pq)

        # Optimization: If we've already found a better path, skip.
        if current_distance > distances[current_node]:
            continue

        # --- Relaxation Step ---

        # Iterate through all neighbors of the current node.
        # .items() is used as the value of each graph key is another dictionary.
        for neighbor, weight in graph[current_node].items():
            distance_through_u = current_distance + weight

            # If we found a new, shorter path to the neighbor...
            if distance_through_u < distances[neighbor]:
                # ...update the new shortest distance.
                distances[neighbor] = distance_through_u
                # ...record the predecessor for path reconstruction.
                predecessors[neighbor] = current_node
                # ...and push the new path information to the priority queue.
                heapq.heappush(pq, (distance_through_u, neighbor))

    return distances, predecessors

def reconstruct_path(predecessors: dict, start_node: str, end_node: str):
    """
    Reconstructs the shortest path from the predecessors dictionary.
    """
    path = []
    current_node = end_node
    while current_node is not None:
        path.insert(0, current_node)
        current_node = predecessors.get(current_node)

    if path and path[0] == start_node:
        return path
    else:
        return None


# ==============================================================================
# MAIN EXECUTION BLOCK - DEMONSTRATION FOR PROBLEM 14 (SIMPLE GRAPH)
# ==============================================================================
if __name__ == "__main__":
    # --- 1. Graph Creation for a Simple Graph ---
    # A dictionary of dictionaries is a natural way to represent a simple graph.
    # The keys of the inner dicts are unique, preventing parallel edges.
    simple_graph = {
        'A': {'B': 10, 'C': 3},
        'B': {'A': 10, 'C': 1, 'D': 2},
        'C': {'A': 3, 'B': 1, 'D': 8, 'E': 2},
        'D': {'B': 2, 'C': 8, 'E': 5},
        'E': {'C': 2, 'D': 5}
```

```python
    }

    start_node = 'A'

    # --- 2. Execution ---
    distances, predecessors = dijkstra_simple_graph(simple_graph, start_node)

    # --- 3. Reporting Results ---
    print("=" * 60)
    print("Dijkstra's Algorithm Report for Simple Graph (Problem 14)")
    print(f"Source Node: '{start_node}'")
    print("=" * 60)

    # Sort nodes for consistent output order.
    sorted_nodes = sorted(simple_graph.keys())

    for node in sorted_nodes:
        print(f"\n--- Destination: '{node}' ---")
        dist = distances.get(node)

        print(f"  Shortest Distance: ", end="")
        if dist == float('inf'):
            print("UNREACHABLE")
        else:
            print(dist)

            path = reconstruct_path(predecessors, start_node, node)
            print("  Path: ", end="")
            if path:
                print(" -> ".join(path))
            else:
                print(f"'{node}' (Source Node)")

    print("=" * 60)
```

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

1.4.2 Cài đặt bằng C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <string>
#include <map>
#include <algorithm>
#include <functional> // For std::greater

// Use long long to avoid overflow with long paths.
using ll = long long;
// Use type aliases for better readability.
using Edge = std::pair<int, int>; // {weight, neighbor_node}
using Graph = std::vector<std::vector<Edge>>;

/**
 * @class DijkstraSolver
```

```
 * @brief A class to compute shortest paths from a single source using Dijkstra's algorithm.
 *
 * This class is designed for simple graphs and encapsulates the data and logic for the algorithm.
 */
class DijkstraSolver {
public:
    std::vector<ll> distances;
    std::vector<int> predecessors;
    int num_nodes;

    /**
     * @brief Constructs a DijkstraSolver for a graph with a given number of nodes.
     * @param n The number of nodes in the graph.
     */
    DijkstraSolver(int n) : num_nodes(n) {
        // Initialize distances to infinity.
        distances.resize(n, std::numeric_limits<ll>::max());
        // Initialize predecessors to -1 (indicating no predecessor).
        predecessors.resize(n, -1);
    }

    /**
     * @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.
     */
    void solve(const Graph& adj, int start_node) {
        // The distance from the start node to itself is 0.
        distances[start_node] = 0;

        // Min-heap: stores {distance, node}.
        // std::greater is used to make the priority_queue a min-heap.
        std::priority_queue<std::pair<ll, int>,
                            std::vector<std::pair<ll, int>>,
                            std::greater<std::pair<ll, int>>> pq;
        pq.push({0, start_node});

        while (!pq.empty()) {
            ll d = pq.top().first;
            int u = pq.top().second;
            pq.pop();

            // Optimization: if we've found a better path already, skip this one.
            if (d > distances[u]) {
                continue;
            }

            // Iterate through all outgoing edges of node u.
            for (const auto& edge : adj[u]) {
                int weight = edge.first;
                int v = edge.second;

                // Relaxation step: if we found a shorter path to v
```

```
through u...
                if (distances[u] + weight < distances[v]) {
                    // ...update the distance and predecessor...
                    distances[v] = distances[u] + weight;
                    predecessors[v] = u;
                    // ...and push the new, better path to the priority queue.
                    pq.push({distances[v], v});
                }
            }
        }
    }

    /**
     * @brief Reconstructs the shortest path to a given end node.
     * @param end_node The destination node.
     * @return A vector of integers representing the path from the start node to the end node.
     */
    std::vector<int> reconstruct_path(int end_node) {
        std::vector<int> path;
        for (int at = end_node; at != -1; at = predecessors[at]) {
            path.push_back(at);
        }
        std::reverse(path.begin(), path.end());

        if (!path.empty() && path[0] == 0) { // Assuming start_node is always 0
            return path;
        }
        return {}; // Return an empty path if no path was found.
    }
};

// ============================================================================
// MAIN FUNCTION - DEMONSTRATION FOR PROBLEM 14 (SIMPLE GRAPH)
// ============================================================================
int main() {
    int num_nodes = 5;
    int start_node = 0; // Starting from node 'A' (index 0)

    std::map<int, std::string> node_names = {{0,"A"}, {1,"B"}, {2,"C"}, {3,"D"}, {4,"E"}};

    // --- Graph Creation for a Simple Graph ---
    // Since it's a simple, undirected graph, for each edge (u, v), we add both
    // u -> v and v -> u to the adjacency list.
    Graph adj(num_nodes);

    // Node A (0)
    adj[0].push_back({10, 1}); // A -> B (weight 10)
    adj[0].push_back({3, 2});  // A -> C (weight 3)
    // Node B (1)
    adj[1].push_back({10, 0}); // B -> A
    adj[1].push_back({1, 2});  // B -> C
    adj[1].push_back({2, 3});  // B -> D
    // Node C (2)
```

```cpp
    adj[2].push_back({3, 0});  // C -> A
    adj[2].push_back({1, 1});  // C -> B
    adj[2].push_back({8, 3});  // C -> D
    adj[2].push_back({2, 4});  // C -> E
    // Node D (3)
    adj[3].push_back({2, 1});  // D -> B
    adj[3].push_back({8, 2});  // D -> C
    adj[3].push_back({5, 4});  // D -> E
    // Node E (4)
    adj[4].push_back({2, 2});  // E -> C
    adj[4].push_back({5, 3});  // E -> D

    // --- Execution and Output ---

    DijkstraSolver solver(num_nodes);
    solver.solve(adj, start_node);

    std::cout << "============================================================" << std::endl;
    std::cout << "Dijkstra's Algorithm Report for Simple Graph (Problem 14)" << std::endl;
    std::cout << "Source Node: '" << node_names[start_node] << "'" << std::endl;
    std::cout << "============================================================" << std::endl;

    for (int i = 0; i < num_nodes; ++i) {
        std::cout << "\n--- Destination: '" << node_names[i] << "' ---" << std::endl;
        std::cout << "  Shortest Distance: ";
        if (solver.distances[i] == std::numeric_limits<ll>::max()) {
            std::cout << "UNREACHABLE" << std::endl;
        } else {
            std::cout << solver.distances[i] << std::endl;
            std::vector<int> path = solver.reconstruct_path(i);
            std::cout << "  Path: ";
            if (path.empty()) {
                std::cout << "N/A" << std::endl;
            } else {
                for (size_t j = 0; j < path.size(); ++j) {
                    std::cout << node_names[path[j]] << (j == path.size() - 1 ? "" : " -> ");
                }
                std::cout << std::endl;
            }
        }
    }
    std::cout << "============================================================" << std::endl;

    return 0;
}
```

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

Đề 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 $B \to D$ (trọng số 2), thêm cạnh mới $B \to D$ (trọng số 1).
- Cạnh song song A-C: Cạnh gốc $A \to C$ (trọng số 3), thêm cạnh mới $A \to C$ (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: $d = \{A : 0, \cdots : \infty\}$, PQ={(0, A)}
1	A	$\{A\}$	Lấy '(0, A)'. Relax (A,B,10) $\to d[B] = 10$. Relax (A,C,3) $\to d[C] = 3$. Relax (A,C,8) $\to 0+8 \not< 3$. Không cập nhật. PQ={(3,C), (10,B)}
2	C	$\{A,C\}$	Lấy '(3, C)'. Relax (C,B,1) $\to 3+1 < d[B] = 10 \to d[B] = 4$. Relax (C,D,8) $\to 3+8 < \infty \to d[D] = 11$. Relax (C,E,2) $\to 3+2 < \infty \to d[E] = 5$. PQ={(4,B), (5,E), (10,B), (11,D)}
3	B	$\{A,C,B\}$	Lấy '(4, B)'. Relax (B,D,2) $\to 4+2 < d[D] = 11 \to d[D] = 6$. Relax (B,D,1) $\to 4+1 < d[D] = 6 \to \mathbf{d[D] = 5}$. PQ={(5,D), (5,E), (6,D), (10,B), ...}
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), ...}
5	E	$\{A,C,B,D,E\}$	Lấy '(5, E)'. Relax (E,D,5) $\to 5+5 \not< d[D] = 5$. 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 $B \to D$ 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)'.

```python
import heapq

def dijkstra_multigraph_python(graph: dict, all_nodes: set, start_node: str):
    """
    Implements Dijkstra's algorithm for a multigraph.

    This function calculates the shortest paths from a single source node to all other
    nodes in a graph that may contain parallel edges.

    :param graph: A dictionary representing the multigraph.
                  Keys are source nodes, and values are lists of (neighbor, weight) tuples.
                  Example: {'A': [('B', 10), ('B', 12)]}
    :param all_nodes: A set containing all unique nodes in the graph. This is necessary
                      to initialize distances for nodes that may only be destinations.
    :param start_node: The starting node for the algorithm.
    :return: A tuple containing two dictionaries: (distances, predecessors).
             'distances' maps each node to its shortest distance from the source.
             'predecessors' maps each node to its preceding node in the shortest path.
    """
    # --- Initialization ---

    # Initialize distances for all nodes to infinity, and the start node to 0.
    distances = {node: float('inf') for node in all_nodes}
    distances[start_node] = 0

    # This dictionary will store the path.
    predecessors = {node: None for node in all_nodes}

    # The priority queue stores tuples of (distance, node).
    # heapq implements a min-heap, so it will always give us the item with the smallest distance.
    pq = [(0, start_node)]

    # --- Main Loop ---

    while pq:
        # Get the node with the smallest distance from the priority queue.
        current_distance, current_node = heapq.heappop(pq)

        # Optimization: If we have already found a shorter path to this node, skip.
        if current_distance > distances[current_node]:
            continue

```

```
        # --- Relaxation Step ---

        # This is the key part for multigraphs. We iterate through the list of edges.
        # If there are parallel edges, this loop will process all of them.
        if current_node in graph:
            for neighbor, weight in graph[current_node]:
                distance_through_u = current_distance + weight

                # If we found a new, shorter path to the neighbor...
                if distance_through_u < distances[neighbor]:
                    # ...update the new shortest distance.
                    distances[neighbor] = distance_through_u
                    # ...record that we reached this neighbor via the current node.
                    predecessors[neighbor] = current_node
                    # ...and push the new path information to the priority queue.
                    heapq.heappush(pq, (distance_through_u, neighbor))

    return distances, predecessors

def reconstruct_path(predecessors: dict, start_node: str, end_node: str):
    """
    Reconstructs the shortest path from the predecessors dictionary.

    :param predecessors: The dictionary mapping each node to its predecessor.
    :param start_node: The starting node of the path.
    :param end_node: The ending node of the path.
    :return: A list of nodes representing the path, or None if no path exists.
    """
    path = []
    current_node = end_node
    # Backtrack from the end node to the start node.
    while current_node is not None:
        path.insert(0, current_node) # Insert at the beginning to build the path in correct order
        current_node = predecessors.get(current_node) # Use .get for safety

    # If the path starts with the start_node, it's a valid path.
    if path and path[0] == start_node:
        return path
    else:
        return None


# ==============================================================================
# MAIN EXECUTION BLOCK - DEMONSTRATION FOR PROBLEM 15 (MULTIGRAPH)
# ==============================================================================
if __name__ == "__main__":
    # --- 1. Graph Creation for a Multigraph ---
    # The data structure is a dictionary where keys are nodes and values are
    # lists of (neighbor, weight) tuples. This allows for parallel edges.
```

```python
    multigraph = {
        'A': [('B', 10), ('C', 3), ('C', 8)],  # Parallel edge A->C
        'B': [('A', 10), ('C', 1), ('D', 2), ('D', 1)], # Parallel edge B->D
        'C': [('A', 3), ('A', 8), ('B', 1), ('D', 8), ('E', 2)],
        'D': [('B', 2), ('B', 1), ('C', 8), ('E', 5)],
        'E': [('C', 2), ('D', 5)]
    }

    start_node = 'A'

    # --- 2. Pre-processing: Get all unique nodes ---
    # This is important for multigraphs where a node might only be a destination.
    all_nodes = set(multigraph.keys())
    for node in multigraph:
        for neighbor, weight in multigraph[node]:
            all_nodes.add(neighbor)

    # --- 3. Execution ---
    # Run the Dijkstra algorithm on the defined graph.
    distances, predecessors = dijkstra_multigraph_python(multigraph, all_nodes, start_node)

    # --- 4. Reporting Results ---
    print("=" * 60)
    print("Dijkstra's Algorithm Report for Multigraph (Problem 15)")
    print(f"Source Node: '{start_node}'")
    print("=" * 60)

    # Sort nodes for consistent output order.
    sorted_nodes = sorted(list(all_nodes))

    for node in sorted_nodes:
        print(f"\n--- Destination: '{node}' ---")
        dist = distances.get(node)

        print(f"  Shortest Distance: ", end="")
        if dist == float('inf'):
            print("UNREACHABLE")
        else:
            print(dist)

            # Reconstruct and print the path
            path = reconstruct_path(predecessors, start_node, node)
            print("  Path: ", end="")
            if path:
                print(" -> ".join(path))
            else:
                # This case handles the source node itself, which has no path to reconstruct.
                print(f"'{node}' (Source Node)")

    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.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <string>
#include <map>
#include <algorithm>
#include <functional> // For std::greater

// Use long long to avoid overflow with long paths.
using ll = long long;
// Use type aliases for better readability.
using Edge = std::pair<int, int>; // {weight, neighbor_node}
using Graph = std::vector<std::vector<Edge>>;

/**
 * @class DijkstraSolver
 * @brief A class to compute the shortest paths from a single source using Dijkstra's algorithm.
 *
 * This class encapsulates the data structures and logic needed for the algorithm,
 * including storing distances, predecessors for path reconstruction, and the main solving logic.
 * It is designed to work with graphs represented by an adjacency list.
 */
class DijkstraSolver {
public:
    std::vector<ll> distances;
    std::vector<int> predecessors;
    int num_nodes;

    /**
     * @brief Constructs a DijkstraSolver for a graph with a given number of nodes.
     * @param n The number of nodes in the graph.
     */
    DijkstraSolver(int n) : num_nodes(n) {
        // Initialize distances to infinity.
        distances.resize(n, std::numeric_limits<ll>::max());
        // Initialize predecessors to -1 (indicating no predecessor).
        predecessors.resize(n, -1);
    }

    /**
     * @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.
     */
    void solve(const Graph& adj, int start_node) {
```

```
        // The distance from the start node to itself is 0.
        distances[start_node] = 0;

        // Min-heap: stores {distance, node}.
        // std::greater is used to make the priority_queue a min-heap.
        std::priority_queue<std::pair<ll, int>,
                            std::vector<std::pair<ll, int>>,
                            std::greater<std::pair<ll, int>>> pq;
        pq.push({0, start_node});

        while (!pq.empty()) {
            ll d = pq.top().first;
            int u = pq.top().second;
            pq.pop();

            // Optimization: if we've found a better path already, skip this one.
            // This is crucial for correctness and performance.
            if (d > distances[u]) {
                continue;
            }

            // Iterate through all outgoing edges of node u.
            // Use const auto& to avoid unnecessary copies.
            for (const auto& edge : adj[u]) {
                int weight = edge.first;
                int v = edge.second;

                // Relaxation step: if we found a shorter path to v through u...
                if (distances[u] + weight < distances[v]) {
                    // ...update the distance and predecessor...
                    distances[v] = distances[u] + weight;
                    predecessors[v] = u;
                    // ...and push the new, better path to the priority queue.
                    pq.push({distances[v], v});
                }
            }
        }
    }

    /**
     * @brief Reconstructs the shortest path to a given end node.
     * @param end_node The destination node.
     * @return A vector of integers representing the path from the start node to the end node.
     *         Returns an empty vector if no path exists.
     */
    std::vector<int> reconstruct_path(int end_node) {
        std::vector<int> path;
        // Backtrack from the end node using the predecessors array.
        for (int at = end_node; at != -1; at = predecessors[at]) {
            path.push_back(at);
        }
        // The path is constructed in reverse, so we need to reverse it back.
        std::reverse(path.begin(), path.end());
```

```
        // If the path does not start with the source node, it's not a valid path.
        if (!path.empty() && path[0] == 0) { // Assuming start_node is always 0 in this context
            return path;
        }
        return {}; // Return an empty path if no path was found.
    }
};

// =====================================================================
// MAIN FUNCTION - DEMONSTRATION FOR PROBLEM 15 (MULTIGRAPH)
// =====================================================================
int main() {
    int num_nodes = 5;
    int start_node = 0; // Starting from node 'A' (index 0)

    // Map integer indices to human-readable names for clear output.
    std::map<int, std::string> node_names = {{0,"A"}, {1,"B"}, {2,"C"}, {3,"D"}, {4,"E"}};

    // --- Graph Creation for a Multigraph ---
    // The adjacency list naturally handles multigraphs by allowing multiple
    // entries for the same neighbor in the list.
    Graph adj(num_nodes);

    // Add ALL edges, including parallel ones.
    // Node A (0)
    adj[0].push_back({10, 1}); // A -> B
    adj[0].push_back({3, 2});  // A -> C (edge 1)
    adj[0].push_back({8, 2});  // A -> C (edge 2, parallel)

    // Node B (1)
    adj[1].push_back({10, 0});
    adj[1].push_back({1, 2});  // B -> C
    adj[1].push_back({2, 3});  // B -> D (edge 1)
    adj[1].push_back({1, 3});  // B -> D (edge 2, parallel, better weight)

    // Node C (2)
    adj[2].push_back({3, 0});  // C -> A (edge 1)
    adj[2].push_back({8, 0});  // C -> A (edge 2, parallel)
    adj[2].push_back({1, 1});
    adj[2].push_back({8, 3});
    adj[2].push_back({2, 4});

    // Node D (3)
    adj[3].push_back({2, 1});  // D -> B (edge 1)
    adj[3].push_back({1, 1});  // D -> B (edge 2, parallel)
    adj[3].push_back({8, 2});
    adj[3].push_back({5, 4});

    // Node E (4)
    adj[4].push_back({2, 2});
    adj[4].push_back({5, 3});

```

```cpp
    // --- Execution and Output ---

    // Create an instance of the solver.
    DijkstraSolver solver(num_nodes);

    // Run the algorithm.
    solver.solve(adj, start_node);

    // Print a detailed report of the results.
    std::cout << "============================================================" << std::endl;
    std::cout << "Dijkstra's Algorithm Report for Multigraph (Problem 15)" << std::endl;
    std::cout << "Source Node: '" << node_names[start_node] << "'" << std::endl;
    std::cout << "============================================================" << std::endl;

    for (int i = 0; i < num_nodes; ++i) {
        std::cout << "\n--- Destination: '" << node_names[i] << "' ---" << std::endl;
        std::cout << "  Shortest Distance: ";
        if (solver.distances[i] == std::numeric_limits<ll>::max()) {
            std::cout << "UNREACHABLE" << std::endl;
        } else {
            std::cout << solver.distances[i] << std::endl;

            // Reconstruct and print the path
            std::vector<int> path = solver.reconstruct_path(i);
            std::cout << "  Path: ";
            if (path.empty()) {
                std::cout << "N/A" << std::endl;
            } else {
                for (size_t j = 0; j < path.size(); ++j) {
                    std::cout << node_names[path[j]] << (j == path.size() - 1 ? "" : " -> ");
                }
                std::cout << std::endl;
            }
        }
    }
    std::cout << "============================================================" << std::endl;

    return 0;
}
```

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

Đề 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 nới 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: $S \to A$ (trọng số 5), $S \to B$ (trọng số 2), $A \to B$ (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: $A \to B$ (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 $S \to A \to B$ với độ dài là 1.

3.3 Cài đặt (Với Giả định Bắt 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 bắt 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.

```
import heapq

def dijkstra_general_graph(graph: dict, all_nodes: set, start_node: str):
    """
    Implements Dijkstra's algorithm.

    WARNING: This implementation is ONLY correct if all edge weights in the
    'general graph' are non-negative. It can handle parallel edges and
    non-negative self-loops, but it will fail with negative weights.

    :param graph: A dict mapping a node to a list of (neighbor, weight) tuples.
    :param all_nodes: A set containing all unique nodes in the graph.
    :param start_node: The starting node for the algorithm.
```

```
    :return: A tuple containing two dictionaries: (distances, predecessors).
    """
    # --- Initialization ---
    distances = {node: float('inf') for node in all_nodes}
    distances[start_node] = 0
    predecessors = {node: None for node in all_nodes}
    pq = [(0, start_node)]

    # --- Main Loop ---
    while pq:
        current_distance, current_node = heapq.heappop(pq)

        if current_distance > distances[current_node]:
            continue

        # --- Relaxation Step ---
        # The loop structure naturally handles parallel edges and self-loops.
        if current_node in graph:
            for neighbor, weight in graph[current_node]:
                # Critical assumption: weight >= 0.
                # If weight were negative, this greedy logic would be flawed.
                distance_through_u = current_distance + weight

                if distance_through_u < distances[neighbor]:
                    distances[neighbor] = distance_through_u
                    predecessors[neighbor] = current_node
                    heapq.heappush(pq, (distance_through_u, neighbor))

    return distances, predecessors

def reconstruct_path(predecessors: dict, start_node: str, end_node: str):
    """
    Reconstructs the shortest path from the predecessors dictionary.
    """
    path = []
    current_node = end_node
    while current_node is not None:
        path.insert(0, current_node)
        current_node = predecessors.get(current_node)

    if path and path[0] == start_node:
        return path
    else:
        return None

# ==============================================================================
# MAIN EXECUTION BLOCK - DEMONSTRATION FOR PROBLEM 16 (GENERAL GRAPH)
# ==============================================================================
if __name__ == "__main__":
    # --- 1. Graph Creation for a General Graph ---
    # This graph includes parallel edges and a self-loop.
    # CRITICAL ASSUMPTION: All weights are non-negative for Dijkstra to be valid.
    general_graph = {
        'A': [('B', 10), ('C', 3)],
```

```python
        'B': [('D', 2), ('D', 1)],  # Parallel edge B->D
        'C': [('A', 3), ('D', 8), ('E', 2)],
        'D': [('E', 5)],
        'E': [('C', 1), ('E', 4)]   # Self-loop E->E with non-negative weight
    }

    start_node = 'A'

    # --- 2. Pre-processing: Get all unique nodes ---
    all_nodes = set(general_graph.keys())
    for node in general_graph:
        for neighbor, weight in general_graph[node]:
            all_nodes.add(neighbor)

    # --- 3. Execution ---
    distances, predecessors = dijkstra_general_graph(general_graph, all_nodes, start_node)

    # --- 4. Reporting Results ---
    print("=" * 70)
    print("Dijkstra's Algorithm Report for General Graph (Problem 16)")
    print(f"Source Node: '{start_node}'")
    print("-" * 70)
    print("WARNING: These results are correct ONLY under the assumption that")
    print("         all edge weights in the graph are non-negative.")
    print("         Dijkstra's algorithm fails with negative edge weights.")
    print("=" * 70)

    sorted_nodes = sorted(list(all_nodes))

    for node in sorted_nodes:
        print(f"\n--- Destination: '{node}' ---")
        dist = distances.get(node)

        print(f"  Shortest Distance: ", end="")
        if dist == float('inf'):
            print("UNREACHABLE")
        else:
            print(dist)

            path = reconstruct_path(predecessors, start_node, node)
            print("  Path: ", end="")
            if path:
                print(" -> ".join(path))
            else:
                print(f"'{node}' (Source Node)")

    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.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <string>
#include <map>
#include <algorithm>
#include <functional> // For std::greater

// Use long long to avoid overflow with long paths.
using ll = long long;
// Use type aliases for better readability.
using Edge = std::pair<int, int>; // {weight, neighbor_node}
using Graph = std::vector<std::vector<Edge>>;

/**
 * @class DijkstraSolver
 * @brief A class to compute shortest paths from a single source using
   Dijkstra's algorithm.
 *
 * This implementation is suitable for graphs with non-negative edge
   weights.
 * It can handle parallel edges and self-loops as long as their weights
   are not negative.
 */
class DijkstraSolver {
public:
    std::vector<ll> distances;
    std::vector<int> predecessors;
    int num_nodes;

    /**
     * @brief Constructs a DijkstraSolver for a graph with a given number
     of nodes.
     * @param n The number of nodes in the graph.
     */
    DijkstraSolver(int n) : num_nodes(n) {
        distances.resize(n, std::numeric_limits<ll>::max());
        predecessors.resize(n, -1); // -1 indicates no predecessor
    }

    /**
     * @brief Executes Dijkstra's algorithm.
     * @warning This method is ONLY correct for graphs with non-negative
    edge weights.
     * @param adj The graph, represented as an adjacency list.
     * @param start_node The index of the source node.
     */
    void solve(const Graph& adj, int start_node) {
        distances[start_node] = 0;

        std::priority_queue<std::pair<ll, int>,
                            std::vector<std::pair<ll, int>>,
                            std::greater<std::pair<ll, int>>> pq;
        pq.push({0, start_node});

        while (!pq.empty()) {
            ll d = pq.top().first;
```

```
            int u = pq.top().second;
            pq.pop();

            if (d > distances[u]) {
                continue;
            }

            for (const auto& edge : adj[u]) {
                int weight = edge.first;
                int v = edge.second;

                // The core logic of Dijkstra's algorithm.
                // Assumes 'weight' is non-negative.
                if (distances[u] + weight < distances[v]) {
                    distances[v] = distances[u] + weight;
                    predecessors[v] = u;
                    pq.push({distances[v], v});
                }
            }
        }
    }

    /**
     * @brief Reconstructs the shortest path to a given end node.
     * @param end_node The destination node.
     * @return A vector of integers representing the path.
     */
    std::vector<int> reconstruct_path(int end_node) {
        std::vector<int> path;
        for (int at = end_node; at != -1; at = predecessors[at]) {
            path.push_back(at);
        }
        std::reverse(path.begin(), path.end());

        if (!path.empty() && path[0] == 0) { // Assuming start_node is always 0
            return path;
        }
        return {};
    }
};

// ==========================================================================
// MAIN FUNCTION - DEMONSTRATION FOR PROBLEM 16 (GENERAL GRAPH)
// ==========================================================================
int main() {
    int num_nodes = 5;
    int start_node = 0;

    std::map<int, std::string> node_names = {{0,"A"}, {1,"B"}, {2,"C"}, {3,"D"}, {4,"E"}};

    // --- Graph Creation for a General Graph ---
    // This graph includes parallel edges and a self-loop.
    // CRITICAL ASSUMPTION: All weights are non-negative.
    Graph adj(num_nodes);

    // Node A (0)
```

```
    adj[0].push_back({10, 1}); // A -> B
    adj[0].push_back({3, 2});  // A -> C
    // Node B (1)
    adj[1].push_back({2, 3});  // B -> D (edge 1)
    adj[1].push_back({1, 3});  // B -> D (parallel edge with better weight)
    // Node C (2)
    adj[2].push_back({3, 0});
    adj[2].push_back({8, 3});
    adj[2].push_back({2, 4});
    // Node D (3)
    adj[3].push_back({5, 4});
    // Node E (4)
    adj[4].push_back({1, 2});
    adj[4].push_back({4, 4});  // E -> E (self-loop with non-negative weight)

    // --- Execution and Output ---

    DijkstraSolver solver(num_nodes);
    solver.solve(adj, start_node);

    std::cout << "============================================================" << std::endl;
    std::cout << "Dijkstra's Algorithm Report for General Graph (Problem 16)" << std::endl;
    std::cout << "Source Node: '" << node_names[start_node] << "'" << std::endl;
    std::cout << "WARNING: Results are correct ONLY under the assumption that" << std::endl;
    std::cout << "         all edge weights are non-negative." << std::endl;
    std::cout << "============================================================" << std::endl;

    for (int i = 0; i < num_nodes; ++i) {
        std::cout << "\n--- Destination: '" << node_names[i] << "' ---" << std::endl;
        std::cout << "  Shortest Distance: ";
        if (solver.distances[i] == std::numeric_limits<ll>::max()) {
            std::cout << "UNREACHABLE" << std::endl;
        } else {
            std::cout << solver.distances[i] << std::endl;
            std::vector<int> path = solver.reconstruct_path(i);
            std::cout << "  Path: ";
            if (path.empty()) {
                std::cout << "N/A" << std::endl;
            } else {
                for (size_t j = 0; j < path.size(); ++j) {
                    std::cout << node_names[path[j]] << (j == path.size() - 1 ? "" : " -> ");
                }
                std::cout << std::endl;
            }
        }
    }
```

```
    std::cout << "
================================================================================" <<
   std::endl;

    return 0;
}
```

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.

```
def bellman_ford(graph: dict, all_nodes: set, start_node: str):
    """
    Implements the Bellman-Ford algorithm.
    Handles negative weights and detects negative cycles.

    :param graph: Adjacency list representation {'u': [('v', w), ...]}
    :param all_nodes: A set of all unique node identifiers.
    :param start_node: The source node.
    :return: A tuple (distances, predecessors). If a negative cycle is
   found,
             distances for affected nodes are set to -infinity.
```

```python
    """
    num_nodes = len(all_nodes)
    distances = {node: float('inf') for node in all_nodes}
    predecessors = {node: None for node in all_nodes}
    distances[start_node] = 0

    # 1. Relax edges |V| - 1 times
    for _ in range(num_nodes - 1):
        for u in graph:
            for v, w in graph[u]:
                if distances[u] != float('inf') and distances[u] + w < distances[v]:
                    distances[v] = distances[u] + w
                    predecessors[v] = v

    # 2. Check for negative-weight cycles
    # If we can still relax an edge, it must be part of a negative cycle.
    for u in graph:
        for v, w in graph[u]:
            if distances[u] != float('inf') and distances[u] + w < distances[v]:
                # Mark as part of a negative cycle
                distances[v] = float('-inf')

    return distances, predecessors

# --- Example with negative weights and a negative cycle ---
# graph_neg = {
#     'S': [('A', 5), ('C', 2)],
#     'A': [('B', 2)],
#     'C': [('A', -4)],  # Negative edge
#     'D': [('E', -1)], 'E': [('F', -1)], 'F': [('D', -1)] # Negative cycle
# }
# all_nodes_neg = {'S', 'A', 'B', 'C', 'D', 'E', 'F'}
# dist, pred = bellman_ford(graph_neg, all_nodes_neg, 'S')
# print(f"Bellman-Ford distances: {dist}")
# 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.

```cpp
// Define INF and N_INF for readability
const ll INF = std::numeric_limits<ll>::max();
const ll N_INF = std::numeric_limits<ll>::min();

class BellmanFordSolver {
public:
    std::vector<ll> distances;
    int num_nodes;

    BellmanFordSolver(int n) : num_nodes(n) {
        distances.resize(n, INF);
    }

    void solve(const Graph& adj, int start_node) {
        distances[start_node] = 0;
```

```cpp

        // Relax all edges |V| - 1 times
        for (int i = 0; i < num_nodes - 1; ++i) {
            for (int u = 0; u < num_nodes; ++u) {
                if (distances[u] == INF) continue;
                for (const auto& edge : adj[u]) {
                    int weight = edge.first;
                    int v = edge.second;
                    if (distances[u] + weight < distances[v]) {
                        distances[v] = distances[u] + weight;
                    }
                }
            }
        }

        // Run a final iteration to detect negative cycles
        for (int i = 0; i < num_nodes - 1; ++i) { // Propagate N_INF
            for (int u = 0; u < num_nodes; ++u) {
                if (distances[u] == INF) continue;
                for (const auto& edge : adj[u]) {
                    int weight = edge.first;
                    int v = edge.second;
                    if (distances[u] + weight < distances[v]) {
                        // Node is part of or reachable from a negative cycle
                        distances[v] = N_INF;
                    }
                }
            }
        }
    }
};
```

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 nới lỏng tất cả các cạnh, nó chỉ nới 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.

```python
from collections import deque

def spfa(graph: dict, all_nodes: set, start_node: str):
    """
    Implements the Shortest Path Faster Algorithm (SPFA).
```

```
    Handles negative weights and detects negative cycles.
    """
    num_nodes = len(all_nodes)
    distances = {node: float('inf') for node in all_nodes}
    predecessors = {node: None for node in all_nodes}
    count = {node: 0 for node in all_nodes}
    in_queue = {node: False for node in all_nodes}
    queue = deque()

    distances[start_node] = 0
    queue.append(start_node)
    in_queue[start_node] = True
    count[start_node] = 1

    while queue:
        u = queue.popleft()
        in_queue[u] = False

        if u not in graph: continue

        for v, w in graph[u]:
            if distances[u] + w < distances[v]:
                distances[v] = distances[u] + w
                predecessors[v] = u

                if not in_queue[v]:
                    queue.append(v)
                    in_queue[v] = True
                    count[v] += 1
                    if count[v] >= num_nodes:
                        # Negative cycle detected
                        print(f"Negative cycle detected involving node {v}")
                        return None, None # Indicate error

    return distances, predecessors
```

Listing 9: Cài đặt SPFA trong Python.

```
#include <queue> // For std::queue

class SpfaSolver {
public:
    std::vector<ll> distances;
    int num_nodes;

    SpfaSolver(int n) : num_nodes(n) {
        distances.resize(n, INF);
    }

    // Returns true if no negative cycle is detected, false otherwise.
    bool solve(const Graph& adj, int start_node) {
        std::vector<int> count(num_nodes, 0);
        std::vector<bool> in_queue(num_nodes, false);
        std::queue<int> q;

```

```cpp
        distances[start_node] = 0;
        q.push(start_node);
        in_queue[start_node] = true;
        count[start_node]++;

        while (!q.empty()) {
            int u = q.front();
            q.pop();
            in_queue[u] = false;

            for (const auto& edge : adj[u]) {
                int weight = edge.first;
                int v = edge.second;

                if (distances[u] != INF && distances[u] + weight < distances[v]) {
                    distances[v] = distances[u] + weight;
                    if (!in_queue[v]) {
                        q.push(v);
                        in_queue[v] = true;
                        count[v]++;
                        if (count[v] >= num_nodes) {
                            // Negative cycle detected
                            return false;
                        }
                    }
                }
            }
        }
        return true; // Success
    }
};
```

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.