Đồ án cuối kì

Hoàng Quang Huy

20/07/2025

1 Project 5: Shortest Path Problems on Graphs

Dự án này thực hiện thuật toán Dijkstra để giải quyết bài toán tìm đường đi ngắn nhất trên ba loại đồ thị khác nhau:

- Đồ thị đơn (Simple Graph)
- Đa đồ thị (Multigraph)
- Đồ thị tổng quát (General Graph)

Tài nguyên tham khảo: Wikipedia/shortest path problem

2 Bài toán 14: Thuật toán Dijkstra cho Đồ thị đơn

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

Listing 1: Thuật toán Dijkstra cho đồ thị đơn

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <algorithm>
using namespace std;

struct Edge {
    int to;
    int weight;

    Edge(int t, int w) : to(t), weight(w) {}
};

class SimpleGraph {
private:
    int vertices;
    vector<vector<Edge>> adjList;

public:
    SimpleGraph(int v) : vertices(v) {
        adjList.resize(v);
    }

    // Add edge for simple graph (no parallel edges, no self-loops)
```

```
26      void addEdge(int from, int to, int weight) {
27          if (from == to) {
28              cout << "Warning: Self-loops not allowed in simple graph" <<
                    endl;
29              return;
30          }
31
32          // Check if edge already exists
33          for (const Edge& e : adjList[from]) {
34              if (e.to == to) {
35                  cout << "Warning: Parallel edges not allowed in simple graph
                            " << endl;
36                  return;
37              }
38          }
39
40          adjList[from].push_back(Edge(to, weight));
41          adjList[to].push_back(Edge(from, weight)); // undirected
42      }
43
44      // Dijkstra's algorithm implementation
45      vector<int> dijkstra(int source, vector<int>& parent) {
46          vector<int> dist(vertices, INT_MAX);
47          priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<
                int, int>>> pq;
48
49          parent.assign(vertices, -1);
50          dist[source] = 0;
51          pq.push({0, source});
52
53          while (!pq.empty()) {
54              int u = pq.top().second;
55              int d = pq.top().first;
56              pq.pop();
57
58              if (d > dist[u]) continue;
59
60              for (const Edge& edge : adjList[u]) {
61                  int v = edge.to;
62                  int weight = edge.weight;
63
64                  if (dist[u] + weight < dist[v]) {
65                      dist[v] = dist[u] + weight;
66                      parent[v] = u;
67                      pq.push({dist[v], v});
68                  }
69              }
70          }
71
72          return dist;
73      }
74
75      // Print shortest path from source to target
76      void printPath(int source, int target, const vector<int>& parent) {
77          if (parent[target] == -1 && source != target) {
78              cout << "No path exists from " << source << " to " << target <<
                    endl;
79              return;
80          }
81
```

```
        vector<int> path;
        int current = target;
        while (current != -1) {
            path.push_back(current);
            current = parent[current];
        }

        reverse(path.begin(), path.end());

        cout << "Shortest path from " << source << " to " << target << ": ";
        for (int i = 0; i < path.size(); i++) {
            cout << path[i];
            if (i < path.size() - 1) cout << " -> ";
        }
        cout << endl;
    }

    void printGraph() {
        cout << "Simple Graph adjacency list:" << endl;
        for (int i = 0; i < vertices; i++) {
            cout << "Vertex " << i << ": ";
            for (const Edge& e : adjList[i]) {
                cout << "(" << e.to << "," << e.weight << ") ";
            }
            cout << endl;
        }
    }
};

int main() {
    cout << "=== Dijkstra Algorithm for Simple Graph ===" << endl;

    // Create a simple graph with 6 vertices
    SimpleGraph graph(6);

    // Add edges (from, to, weight)
    graph.addEdge(0, 1, 4);
    graph.addEdge(0, 2, 3);
    graph.addEdge(1, 2, 1);
    graph.addEdge(1, 3, 2);
    graph.addEdge(2, 3, 4);
    graph.addEdge(3, 4, 2);
    graph.addEdge(4, 5, 6);

    graph.printGraph();
    cout << endl;

    int source = 0;
    vector<int> parent;
    vector<int> distances = graph.dijkstra(source, parent);

    cout << "Shortest distances from vertex " << source << ":" << endl;
    for (int i = 0; i < distances.size(); i++) {
        cout << "To vertex " << i << ": ";
        if (distances[i] == INT_MAX) {
            cout << "INFINITY" << endl;
        } else {
            cout << distances[i] << endl;
        }
    }
```

```
    cout << endl;

    // Print shortest paths
    for (int i = 1; i < 6; i++) {
        graph.printPath(source, i, parent);
    }

    return 0;
}
```

3 Bài toán 15: Thuật toán Dijkstra cho Đa đồ thị

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

Listing 2: Thuật toán Dijkstra cho đa đồ thị

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <algorithm>
using namespace std;

struct MultiEdge {
    int to;
    int weight;
    int edgeId; // To distinguish parallel edges

    MultiEdge(int t, int w, int id) : to(t), weight(w), edgeId(id) {}
};

class Multigraph {
private:
    int vertices;
    vector<vector<MultiEdge>> adjList;
    int nextEdgeId;

public:
    Multigraph(int v) : vertices(v), nextEdgeId(0) {
        adjList.resize(v);
    }

    // Add edge for multigraph (allows parallel edges, no self-loops for
       simplicity)
    void addEdge(int from, int to, int weight) {
        if (from == to) {
            cout << "Note: Self-loops allowed but skipped in this
               implementation" << endl;
            return;
        }

        adjList[from].push_back(MultiEdge(to, weight, nextEdgeId));
        adjList[to].push_back(MultiEdge(from, weight, nextEdgeId));
        nextEdgeId++;

        cout << "Added edge " << from << "-" << to << " with weight " <<
           weight
             << " (Edge ID: " << nextEdgeId-1 << ")" << endl;
```

```
    }

    // Dijkstra's algorithm for multigraph
    vector<int> dijkstra(int source, vector<int>& parent, vector<int>&
        usedEdge) {
        vector<int> dist(vertices, INT_MAX);
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<
            int, int>>> pq;

        parent.assign(vertices, -1);
        usedEdge.assign(vertices, -1);
        dist[source] = 0;
        pq.push({0, source});

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

            if (d > dist[u]) continue;

            for (const MultiEdge& edge : adjList[u]) {
                int v = edge.to;
                int weight = edge.weight;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    parent[v] = u;
                    usedEdge[v] = edge.edgeId;
                    pq.push({dist[v], v});
                }
            }
        }

        return dist;
    }

    void printPath(int source, int target, const vector<int>& parent, const
        vector<int>& usedEdge) {
        if (parent[target] == -1 && source != target) {
            cout << "No path exists from " << source << " to " << target <<
                endl;
            return;
        }

        vector<int> path;
        vector<int> edges;
        int current = target;

        while (current != source && current != -1) {
            path.push_back(current);
            edges.push_back(usedEdge[current]);
            current = parent[current];
        }

        if (current == source) {
            path.push_back(source);
            reverse(path.begin(), path.end());
            reverse(edges.begin(), edges.end());

```

```
            cout << "Shortest path from " << source << " to " << target << "
                : ";
            for (int i = 0; i < path.size(); i++) {
                cout << path[i];
                if (i < path.size() - 1) {
                    cout << " --(edge " << edges[i] << ")--> ";
                }
            }
            cout << endl;
        }
    }

    void printGraph() {
        cout << "Multigraph adjacency list:" << endl;
        for (int i = 0; i < vertices; i++) {
            cout << "Vertex " << i << ": ";
            for (const MultiEdge& e : adjList[i]) {
                cout << "(" << e.to << "," << e.weight << ",E" << e.edgeId
                    << ") ";
            }
            cout << endl;
        }
    }
};

int main() {
    cout << "=== Dijkstra Algorithm for Multigraph ===" << endl;

    // Create a multigraph with 5 vertices
    Multigraph graph(5);

    // Add edges including parallel edges
    graph.addEdge(0, 1, 10);
    graph.addEdge(0, 1, 5);  // Parallel edge with different weight
    graph.addEdge(0, 2, 3);
    graph.addEdge(1, 2, 2);
    graph.addEdge(1, 3, 1);
    graph.addEdge(2, 3, 8);
    graph.addEdge(2, 3, 4);  // Another parallel edge
    graph.addEdge(3, 4, 2);

    cout << endl;
    graph.printGraph();
    cout << endl;

    int source = 0;
    vector<int> parent, usedEdge;
    vector<int> distances = graph.dijkstra(source, parent, usedEdge);

    cout << "Shortest distances from vertex " << source << ":" << endl;
    for (int i = 0; i < distances.size(); i++) {
        cout << "To vertex " << i << ": ";
        if (distances[i] == INT_MAX) {
            cout << "INFINITY" << endl;
        } else {
            cout << distances[i] << endl;
        }
    }
    cout << endl;

```

```
    // Print shortest paths with edge information
    for (int i = 1; i < 5; i++) {
        graph.printPath(source, i, parent, usedEdge);
    }

    return 0;
}
```

4 Bài toán 16: Thuật toán Dijkstra cho Đồ thị tổng quát

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

Listing 3: Thuật toán Dijkstra cho đồ thị tổng quát

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <algorithm>
#include <set>
using namespace std;

struct GeneralEdge {
    int to;
    int weight;
    int edgeId;
    bool isSelfLoop;

    GeneralEdge(int t, int w, int id, bool selfLoop = false)
        : to(t), weight(w), edgeId(id), isSelfLoop(selfLoop) {}
};

class GeneralGraph {
private:
    int vertices;
    vector<vector<GeneralEdge>> adjList;
    int nextEdgeId;
    set<pair<int, int>> edgeSet; // To track parallel edges

public:
    GeneralGraph(int v) : vertices(v), nextEdgeId(0) {
        adjList.resize(v);
    }

    // Add edge for general graph (allows parallel edges and self-loops)
    void addEdge(int from, int to, int weight) {
        bool isSelfLoop = (from == to);

        if (isSelfLoop) {
            adjList[from].push_back(GeneralEdge(to, weight, nextEdgeId, true
                ));
            cout << "Added self-loop at vertex " << from << " with weight "
                << weight
                  << " (Edge ID: " << nextEdgeId << ")" << endl;
        } else {
            adjList[from].push_back(GeneralEdge(to, weight, nextEdgeId));
            adjList[to].push_back(GeneralEdge(from, weight, nextEdgeId));

```

```
            // Check if this creates a parallel edge
            pair<int, int> edge = {min(from, to), max(from, to)};
            if (edgeSet.count(edge)) {
                cout << "Added parallel edge " << from << "-" << to << " with weight " << weight
                     << " (Edge ID: " << nextEdgeId << ")" << endl;
            } else {
                cout << "Added edge " << from << "-" << to << " with weight " << weight
                     << " (Edge ID: " << nextEdgeId << ")" << endl;
                edgeSet.insert(edge);
            }
        }
        nextEdgeId++;
    }

    // Enhanced Dijkstra's algorithm for general graph
    vector<int> dijkstra(int source, vector<int>& parent, vector<int>& usedEdge) {
        vector<int> dist(vertices, INT_MAX);
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
        vector<bool> visited(vertices, false);

        parent.assign(vertices, -1);
        usedEdge.assign(vertices, -1);
        dist[source] = 0;
        pq.push({0, source});

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

            if (visited[u]) continue;
            visited[u] = true;

            for (const GeneralEdge& edge : adjList[u]) {
                int v = edge.to;
                int weight = edge.weight;

                // Handle self-loops specially
                if (edge.isSelfLoop) {
                    cout << "Processing self-loop at vertex " << u << " (weight: " << weight << ")" << endl;
                    continue; // Self-loops don't contribute to shortest paths to other vertices
                }

                if (!visited[v] && dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    parent[v] = u;
                    usedEdge[v] = edge.edgeId;
                    pq.push({dist[v], v});
                }
            }
        }

        return dist;
    }
```

```
97
98      void printPath(int source, int target, const vector<int>& parent, const
            vector<int>& usedEdge) {
99          if (parent[target] == -1 && source != target) {
100             cout << "No path exists from " << source << " to " << target <<
                endl;
101             return;
102         }
103
104         if (source == target) {
105             cout << "Path from " << source << " to " << target << ": " <<
                source << " (same vertex)" << endl;
106             return;
107         }
108
109         vector<int> path;
110         vector<int> edges;
111         int current = target;
112
113         while (current != source && current != -1) {
114             path.push_back(current);
115             edges.push_back(usedEdge[current]);
116             current = parent[current];
117         }
118
119         if (current == source) {
120             path.push_back(source);
121             reverse(path.begin(), path.end());
122             reverse(edges.begin(), edges.end());
123
124             cout << "Shortest path from " << source << " to " << target << "
                : ";
125             for (int i = 0; i < path.size(); i++) {
126                 cout << path[i];
127                 if (i < path.size() - 1) {
128                     cout << " --(E" << edges[i] << ")--> ";
129                 }
130             }
131             cout << endl;
132         }
133     }
134
135     void printGraph() {
136         cout << "General Graph adjacency list:" << endl;
137         for (int i = 0; i < vertices; i++) {
138             cout << "Vertex " << i << ": ";
139             for (const GeneralEdge& e : adjList[i]) {
140                 cout << "(" << e.to << "," << e.weight << ",E" << e.edgeId;
141                 if (e.isSelfLoop) cout << ",SELF";
142                 cout << ") ";
143             }
144             cout << endl;
145         }
146     }
147
148     void printGraphStatistics() {
149         int totalEdges = nextEdgeId;
150         int selfLoops = 0;
151         int parallelEdges = 0;
152
```

```
        set<pair<int, int>> uniqueEdges;
        for (int i = 0; i < vertices; i++) {
            for (const GeneralEdge& e : adjList[i]) {
                if (e.isSelfLoop) {
                    selfLoops++;
                } else if (i <= e.to) { // Count each undirected edge once
                    pair<int, int> edge = {i, e.to};
                    if (uniqueEdges.count(edge)) {
                        parallelEdges++;
                    } else {
                        uniqueEdges.insert(edge);
                    }
                }
            }
        }

        cout << "Graph Statistics:" << endl;
        cout << "Total vertices: " << vertices << endl;
        cout << "Total edges: " << totalEdges << endl;
        cout << "Self-loops: " << selfLoops << endl;
        cout << "Parallel edges: " << parallelEdges << endl;
        cout << "Unique edges: " << uniqueEdges.size() << endl;
    }
};

int main() {
    cout << "=== Dijkstra Algorithm for General Graph ===" << endl;

    // Create a general graph with 6 vertices
    GeneralGraph graph(6);

    // Add various types of edges
    graph.addEdge(0, 1, 4);
    graph.addEdge(0, 2, 3);
    graph.addEdge(1, 1, 2);    // Self-loop
    graph.addEdge(1, 2, 1);
    graph.addEdge(1, 2, 5);    // Parallel edge with different weight
    graph.addEdge(2, 3, 2);
    graph.addEdge(3, 3, 1);    // Another self-loop
    graph.addEdge(3, 4, 3);
    graph.addEdge(2, 4, 8);
    graph.addEdge(2, 4, 6);    // Another parallel edge
    graph.addEdge(4, 5, 2);
    graph.addEdge(0, 5, 10);

    cout << endl;
    graph.printGraphStatistics();
    cout << endl;
    graph.printGraph();
    cout << endl;

    int source = 0;
    vector<int> parent, usedEdge;
    vector<int> distances = graph.dijkstra(source, parent, usedEdge);

    cout << endl << "Shortest distances from vertex " << source << ":" <<
       endl;
    for (int i = 0; i < distances.size(); i++) {
        cout << "To vertex " << i << ": ";
        if (distances[i] == INT_MAX) {
```

```
            cout << "INFINITY" << endl;
        } else {
            cout << distances[i] << endl;
        }
    }
    cout << endl;

    // Print shortest paths
    for (int i = 0; i < 6; i++) {
        graph.printPath(source, i, parent, usedEdge);
    }

    return 0;
}
```

5 So sánh và Phân tích

5.1 Độ phức tạp thuật toán

Đối với tất cả ba loại đồ thị, độ phức tạp của thuật toán Dijkstra là:

- **Thời gian:** $O((V + E) \log V)$ khi sử dụng priority queue
- **Không gian:** $O(V + E)$ để lưu trữ đồ thị và các mảng phụ trợ

5.2 Đặc điểm của từng loại đồ thị

Đồ thị đơn (Simple Graph):

- Không có cạnh song song
- Không có khuyên (self-loop)
- Thuật toán Dijkstra hoạt động hiệu quả nhất

Đa đồ thị (Multigraph):

- Cho phép cạnh song song
- Thuật toán tự động chọn cạnh có trọng số nhỏ nhất trong các cạnh song song
- Cần theo dõi ID của cạnh được sử dụng

Đồ thị tổng quát (General Graph):

- Cho phép cả cạnh song song và khuyên
- Khuyên thường không ảnh hưởng đến đường đi ngắn nhất giữa các đỉnh khác nhau
- Cần xử lý đặc biệt cho khuyên

5.3 Ứng dụng thực tế

- **Định tuyến mạng:** Tìm đường đi ngắn nhất giữa các router
- **GPS Navigation:** Tìm đường đi ngắn nhất giữa hai địa điểm
- **Game Development:** Pathfinding cho NPC
- **Social Networks:** Tìm mức độ kết nối giữa người dùng

6 Kết luận

Thuật toán Dijkstra là một trong những thuật toán quan trọng nhất trong lý thuyết đồ thị để giải quyết bài toán đường đi ngắn nhất. Việc triển khai thuật toán cho các loại đồ thị khác nhau đòi hỏi:

- Hiểu rõ đặc điểm của từng loại đồ thị
- Xử lý các trường hợp đặc biệt như cạnh song song và khuyên
- Tối ưu hóa cấu trúc dữ liệu để đạt hiệu suất tốt nhất

Ba implementation cung cấp nền tảng vững chắc để giải quyết các bài toán đường đi ngắn nhất trong thực tế.