

# 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

```
1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 #include <climits>
5 #include <algorithm>
6 using namespace std;
7
8 struct Edge {
9     int to;
10    int weight;
11
12    Edge(int t, int w) : to(t), weight(w) {}
13 };
14
15 class SimpleGraph {
16 private:
17     int vertices;
18     vector<vector<Edge>> adjList;
19
20 public:
21     SimpleGraph(int v) : vertices(v) {
22         adjList.resize(v);
23     }
24
25     // 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" <<
29             endl;
30         return;
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
36                 " << endl;
37             return;
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<
48         int, int>>> pq;
49
50     parent.assign(vertices, -1);
51     dist[source] = 0;
52     pq.push({0, source});
53
54     while (!pq.empty()) {
55         int u = pq.top().second;
56         int d = pq.top().first;
57         pq.pop();
58
59         if (d > dist[u]) continue;
60
61         for (const Edge& edge : adjList[u]) {
62             int v = edge.to;
63             int weight = edge.weight;
64
65             if (dist[u] + weight < dist[v]) {
66                 dist[v] = dist[u] + weight;
67                 parent[v] = u;
68                 pq.push({dist[v], v});
69             }
70         }
71     }
72
73     return dist;
74 }
75
76 // Print shortest path from source to target
77 void printPath(int source, int target, const vector<int>& parent) {
78     if (parent[target] == -1 && source != target) {
79         cout << "No path exists from " << source << " to " << target <<
80             endl;
81         return;
82     }

```

```

82     vector<int> path;
83     int current = target;
84     while (current != -1) {
85         path.push_back(current);
86         current = parent[current];
87     }
88
89     reverse(path.begin(), path.end());
90
91     cout << "Shortest path from " << source << " to " << target << ": ";
92     for (int i = 0; i < path.size(); i++) {
93         cout << path[i];
94         if (i < path.size() - 1) cout << " -> ";
95     }
96     cout << endl;
97 }
98
99 void printGraph() {
100     cout << "Simple Graph adjacency list:" << endl;
101     for (int i = 0; i < vertices; i++) {
102         cout << "Vertex " << i << ": ";
103         for (const Edge& e : adjList[i]) {
104             cout << "(" << e.to << ", " << e.weight << ") ";
105         }
106         cout << endl;
107     }
108 }
109 };
110
111 int main() {
112     cout << "=== Dijkstra Algorithm for Simple Graph ===" << endl;
113
114     // Create a simple graph with 6 vertices
115     SimpleGraph graph(6);
116
117     // Add edges (from, to, weight)
118     graph.addEdge(0, 1, 4);
119     graph.addEdge(0, 2, 3);
120     graph.addEdge(1, 2, 1);
121     graph.addEdge(1, 3, 2);
122     graph.addEdge(2, 3, 4);
123     graph.addEdge(3, 4, 2);
124     graph.addEdge(4, 5, 6);
125
126     graph.printGraph();
127     cout << endl;
128
129     int source = 0;
130     vector<int> parent;
131     vector<int> distances = graph.dijkstra(source, parent);
132
133     cout << "Shortest distances from vertex " << source << ":" << endl;
134     for (int i = 0; i < distances.size(); i++) {
135         cout << "To vertex " << i << ": ";
136         if (distances[i] == INT_MAX) {
137             cout << "INFINITY" << endl;
138         } else {
139             cout << distances[i] << endl;
140         }
141     }

```

```

142     cout << endl;
143
144     // Print shortest paths
145     for (int i = 1; i < 6; i++) {
146         graph.printPath(source, i, parent);
147     }
148
149     return 0;
150 }

```

### 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

```

1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <climits>
5  #include <algorithm>
6  using namespace std;
7
8  struct MultiEdge {
9      int to;
10     int weight;
11     int edgeId; // To distinguish parallel edges
12
13     MultiEdge(int t, int w, int id) : to(t), weight(w), edgeId(id) {}
14 };
15
16 class Multigraph {
17 private:
18     int vertices;
19     vector<vector<MultiEdge>> adjList;
20     int nextEdgeId;
21
22 public:
23     Multigraph(int v) : vertices(v), nextEdgeId(0) {
24         adjList.resize(v);
25     }
26
27     // Add edge for multigraph (allows parallel edges, no self-loops for
28     // simplicity)
29     void addEdge(int from, int to, int weight) {
30         if (from == to) {
31             cout << "Note: Self-loops allowed but skipped in this
32             implementation" << endl;
33             return;
34         }
35
36         adjList[from].push_back(MultiEdge(to, weight, nextEdgeId));
37         adjList[to].push_back(MultiEdge(from, weight, nextEdgeId));
38         nextEdgeId++;
39
40         cout << "Added edge " << from << "-" << to << " with weight " <<
41             weight
42             << " (Edge ID: " << nextEdgeId-1 << ")" << endl;

```

```

40 }
41
42 // Dijkstra's algorithm for multigraph
43 vector<int> dijkstra(int source, vector<int>& parent, vector<int>&
    usedEdge) {
44     vector<int> dist(vertices, INT_MAX);
45     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<
        int, int>>> pq;
46
47     parent.assign(vertices, -1);
48     usedEdge.assign(vertices, -1);
49     dist[source] = 0;
50     pq.push({0, source});
51
52     while (!pq.empty()) {
53         int u = pq.top().second;
54         int d = pq.top().first;
55         pq.pop();
56
57         if (d > dist[u]) continue;
58
59         for (const MultiEdge& edge : adjList[u]) {
60             int v = edge.to;
61             int weight = edge.weight;
62
63             if (dist[u] + weight < dist[v]) {
64                 dist[v] = dist[u] + weight;
65                 parent[v] = u;
66                 usedEdge[v] = edge.edgeId;
67                 pq.push({dist[v], v});
68             }
69         }
70     }
71
72     return dist;
73 }
74
75 void printPath(int source, int target, const vector<int>& parent, const
    vector<int>& usedEdge) {
76     if (parent[target] == -1 && source != target) {
77         cout << "No path exists from " << source << " to " << target <<
            endl;
78         return;
79     }
80
81     vector<int> path;
82     vector<int> edges;
83     int current = target;
84
85     while (current != source && current != -1) {
86         path.push_back(current);
87         edges.push_back(usedEdge[current]);
88         current = parent[current];
89     }
90
91     if (current == source) {
92         path.push_back(source);
93         reverse(path.begin(), path.end());
94         reverse(edges.begin(), edges.end());
95     }

```

```

96         cout << "Shortest path from " << source << " to " << target << "
97         : ";
98         for (int i = 0; i < path.size(); i++) {
99             cout << path[i];
100             if (i < path.size() - 1) {
101                 cout << " --(edge " << edges[i] << ")--> ";
102             }
103             cout << endl;
104         }
105     }
106
107     void printGraph() {
108         cout << "Multigraph adjacency list:" << endl;
109         for (int i = 0; i < vertices; i++) {
110             cout << "Vertex " << i << ": ";
111             for (const MultiEdge& e : adjList[i]) {
112                 cout << "(" << e.to << ", " << e.weight << ",E" << e.edgeId
113                 << ") ";
114             }
115             cout << endl;
116         }
117     };
118
119     int main() {
120         cout << "=== Dijkstra Algorithm for Multigraph ===" << endl;
121
122         // Create a multigraph with 5 vertices
123         Multigraph graph(5);
124
125         // Add edges including parallel edges
126         graph.addEdge(0, 1, 10);
127         graph.addEdge(0, 1, 5); // Parallel edge with different weight
128         graph.addEdge(0, 2, 3);
129         graph.addEdge(1, 2, 2);
130         graph.addEdge(1, 3, 1);
131         graph.addEdge(2, 3, 8);
132         graph.addEdge(2, 3, 4); // Another parallel edge
133         graph.addEdge(3, 4, 2);
134
135         cout << endl;
136         graph.printGraph();
137         cout << endl;
138
139         int source = 0;
140         vector<int> parent, usedEdge;
141         vector<int> distances = graph.dijkstra(source, parent, usedEdge);
142
143         cout << "Shortest distances from vertex " << source << ":" << endl;
144         for (int i = 0; i < distances.size(); i++) {
145             cout << "To vertex " << i << ": ";
146             if (distances[i] == INT_MAX) {
147                 cout << "INFINITY" << endl;
148             } else {
149                 cout << distances[i] << endl;
150             }
151         }
152         cout << endl;
153     }

```

```

154 // Print shortest paths with edge information
155 for (int i = 1; i < 5; i++) {
156     graph.printPath(source, i, parent, usedEdge);
157 }
158
159 return 0;
160 }

```

## 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

```

1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <climits>
5  #include <algorithm>
6  #include <set>
7  using namespace std;
8
9  struct GeneralEdge {
10     int to;
11     int weight;
12     int edgeId;
13     bool isSelfLoop;
14
15     GeneralEdge(int t, int w, int id, bool selfLoop = false)
16         : to(t), weight(w), edgeId(id), isSelfLoop(selfLoop) {}
17 };
18
19 class GeneralGraph {
20 private:
21     int vertices;
22     vector<vector<GeneralEdge>> adjList;
23     int nextEdgeId;
24     set<pair<int, int>> edgeSet; // To track parallel edges
25
26 public:
27     GeneralGraph(int v) : vertices(v), nextEdgeId(0) {
28         adjList.resize(v);
29     }
30
31     // Add edge for general graph (allows parallel edges and self-loops)
32     void addEdge(int from, int to, int weight) {
33         bool isSelfLoop = (from == to);
34
35         if (isSelfLoop) {
36             adjList[from].push_back(GeneralEdge(to, weight, nextEdgeId, true));
37             cout << "Added self-loop at vertex " << from << " with weight "
38                 << weight
39                 << " (Edge ID: " << nextEdgeId << ")" << endl;
40         } else {
41             adjList[from].push_back(GeneralEdge(to, weight, nextEdgeId));
42             adjList[to].push_back(GeneralEdge(from, weight, nextEdgeId));
43         }
44     }
45 };

```

```

43     // Check if this creates a parallel edge
44     pair<int, int> edge = {min(from, to), max(from, to)};
45     if (edgeSet.count(edge)) {
46         cout << "Added parallel edge " << from << "-" << to << "
47             with weight " << weight
48             << " (Edge ID: " << nextEdgeId << ")" << endl;
49     } else {
50         cout << "Added edge " << from << "-" << to << " with weight
51             " << weight
52             << " (Edge ID: " << nextEdgeId << ")" << endl;
53         edgeSet.insert(edge);
54     }
55     nextEdgeId++;
56 }
57
58 // Enhanced Dijkstra's algorithm for general graph
59 vector<int> dijkstra(int source, vector<int>& parent, vector<int>&
60     usedEdge) {
61     vector<int> dist(vertices, INT_MAX);
62     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<
63         int, int>>> pq;
64     vector<bool> visited(vertices, false);
65
66     parent.assign(vertices, -1);
67     usedEdge.assign(vertices, -1);
68     dist[source] = 0;
69     pq.push({0, source});
70
71     while (!pq.empty()) {
72         int u = pq.top().second;
73         int d = pq.top().first;
74         pq.pop();
75
76         if (visited[u]) continue;
77         visited[u] = true;
78
79         for (const GeneralEdge& edge : adjList[u]) {
80             int v = edge.to;
81             int weight = edge.weight;
82
83             // Handle self-loops specially
84             if (edge.isSelfLoop) {
85                 cout << "Processing self-loop at vertex " << u << " (
86                     weight: " << weight << ")" << endl;
87                 continue; // Self-loops don't contribute to shortest
88                     paths to other vertices
89             }
90
91             if (!visited[v] && dist[u] + weight < dist[v]) {
92                 dist[v] = dist[u] + weight;
93                 parent[v] = u;
94                 usedEdge[v] = edge.edgeId;
95                 pq.push({dist[v], v});
96             }
97         }
98     }
99
100     return dist;
101 }

```



```

97
98 void printPath(int source, int target, const vector<int>& parent, const
99 vector<int>& usedEdge) {
100     if (parent[target] == -1 && source != target) {
101         cout << "No path exists from " << source << " to " << target <<
102         endl;
103         return;
104     }
105     if (source == target) {
106         cout << "Path from " << source << " to " << target << ": " <<
107         source << " (same vertex)" << endl;
108         return;
109     }
110     vector<int> path;
111     vector<int> edges;
112     int current = target;
113     while (current != source && current != -1) {
114         path.push_back(current);
115         edges.push_back(usedEdge[current]);
116         current = parent[current];
117     }
118     if (current == source) {
119         path.push_back(source);
120         reverse(path.begin(), path.end());
121         reverse(edges.begin(), edges.end());
122
123         cout << "Shortest path from " << source << " to " << target << "
124         : ";
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

```

```

153     set<pair<int, int>> uniqueEdges;
154     for (int i = 0; i < vertices; i++) {
155         for (const GeneralEdge& e : adjList[i]) {
156             if (e.isSelfLoop) {
157                 selfLoops++;
158             } else if (i <= e.to) { // Count each undirected edge once
159                 pair<int, int> edge = {i, e.to};
160                 if (uniqueEdges.count(edge)) {
161                     parallelEdges++;
162                 } else {
163                     uniqueEdges.insert(edge);
164                 }
165             }
166         }
167     }
168
169     cout << "Graph Statistics:" << endl;
170     cout << "Total vertices: " << vertices << endl;
171     cout << "Total edges: " << totalEdges << endl;
172     cout << "Self-loops: " << selfLoops << endl;
173     cout << "Parallel edges: " << parallelEdges << endl;
174     cout << "Unique edges: " << uniqueEdges.size() << endl;
175 }
176 };
177
178 int main() {
179     cout << "=== Dijkstra Algorithm for General Graph ===" << endl;
180
181     // Create a general graph with 6 vertices
182     GeneralGraph graph(6);
183
184     // Add various types of edges
185     graph.addEdge(0, 1, 4);
186     graph.addEdge(0, 2, 3);
187     graph.addEdge(1, 1, 2); // Self-loop
188     graph.addEdge(1, 2, 1);
189     graph.addEdge(1, 2, 5); // Parallel edge with different weight
190     graph.addEdge(2, 3, 2);
191     graph.addEdge(3, 3, 1); // Another self-loop
192     graph.addEdge(3, 4, 3);
193     graph.addEdge(2, 4, 8);
194     graph.addEdge(2, 4, 6); // Another parallel edge
195     graph.addEdge(4, 5, 2);
196     graph.addEdge(0, 5, 10);
197
198     cout << endl;
199     graph.printGraphStatistics();
200     cout << endl;
201     graph.printGraph();
202     cout << endl;
203
204     int source = 0;
205     vector<int> parent, usedEdge;
206     vector<int> distances = graph.dijkstra(source, parent, usedEdge);
207
208     cout << endl << "Shortest distances from vertex " << source << ":" <<
209         endl;
210     for (int i = 0; i < distances.size(); i++) {
211         cout << "To vertex " << i << ": ";
212         if (distances[i] == INT_MAX) {

```

```

212         cout << "INFINITY" << endl;
213     } else {
214         cout << distances[i] << endl;
215     }
216 }
217 cout << endl;
218
219 // Print shortest paths
220 for (int i = 0; i < 6; i++) {
221     graph.printPath(source, i, parent, usedEdge);
222 }
223
224 return 0;
225 }

```

## 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.