Graph Algorithms: 5

Problem 1: Shortest Clear Path in Binary Matrix

Problem Statement: Find the shortest path from top-left to bottom-right in a binary `n x n` matrix, moving 8-directionally through 0s.

Approach:

- Use **BFS** since each step has equal cost.
- Track visited cells to avoid revisiting.
- Move in 8 directions using direction vectors.

Time Complexity: O(n^2) **Space Complexity**: O(n^2)

Code:

```
class Solution {
public:
    int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
        int n = grid.size();
        if (grid[0][0] != 0 || grid[n-1][n-1] != 0) return -1;

        vector<vector<bool>> visited(n, vector<bool>(n, false));
        queue<pair<int, int>> q;
        q.push({0, 0});
        visited[0][0] = true;
        int steps = 1;

        int dir[8][2] = {{1, 0}, {0, 1}, {-1, 0}, {0, -1},
                         {1, 1}, {-1, -1}, {1, -1}, {-1, 1}};

        while (!q.empty()) {
            int sz = q.size();
            while (sz--) {
                auto [r, c] = q.front(); q.pop();
                if (r == n-1 && c == n-1) return steps;

                for (auto [dr, dc] : dir) {
                    int nr = r + dr, nc = c + dc;
                    if (nr >= 0 && nc >= 0 && nr < n && nc < n &&
                        !visited[nr][nc] && grid[nr][nc] == 0) {
                        visited[nr][nc] = true;
                        q.push({nr, nc});
                    }
                }
```

```
            }
            steps++;
        }
        return -1;
    }
};
```

Problem 2: Minimum Effort Path

Problem Statement: Find the path from (0,0) to (m-1,n-1) minimizing the maximum absolute difference in heights between steps.

Approach:

- Use **Dijkstra's algorithm** variant.
- Maintain `effort` matrix and update if a better path is found.

Code with Path Printing:

```
class Solution {
public:
    int minimumEffortPath(vector<vector<int>>& heights) {
        int m = heights.size(), n = heights[0].size();
        vector<vector<int>> dist(m, vector<int>(n, INT_MAX));
        vector<vector<pair<int, int>>> parent(m, vector<pair<int, int>>(n, {-1, -1}));

        priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, greater<>>
        dist[0][0] = 0;
        pq.push({0, 0, 0});

        int dir[] = {1, 0, -1, 0, 1};

        while (!pq.empty()) {
            auto [eff, r, c] = pq.top(); pq.pop();
            if (r == m-1 && c == n-1) break;

            for (int i = 0; i < 4; ++i) {
                int nr = r + dir[i], nc = c + dir[i+1];
                if (nr >= 0 && nr < m && nc >= 0 && nc < n) {
                    int newEff = max(eff, abs(heights[r][c] - heights[nr][nc]));
                    if (newEff < dist[nr][nc]) {
                        dist[nr][nc] = newEff;
                        parent[nr][nc] = {r, c};
                        pq.push({newEff, nr, nc});
                    }
                }
            }
        }
```

```
        // Reconstruct path
        vector<pair<int,int>> path;
        for (int r = m-1, c = n-1; r != -1 && c != -1; tie(r, c) = parent[r][c])
            path.push_back({r, c});
        reverse(path.begin(), path.end());

        // Print path (Optional)
        for (auto [r, c] : path)
            cout << "(" << r << ", " << c << ") ";
        cout << endl;

        return dist[m-1][n-1];
    }
};
```

Problem 3: Find Cheapest Price with K Stops

Approach 1: Dijkstra (Modified with Stops)

```
class Solution {
public:
    int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int
        vector<pair<int, int>> adj[n];
        for (auto& flight : flights)
            adj[flight[0]].emplace_back(flight[1], flight[2]);

        queue<tuple<int, int, int>> q; // node, cost, stops
        vector<int> dist(n, INT_MAX);
        q.push({src, 0, 0});
        dist[src] = 0;

        while (!q.empty()) {
            auto [node, cost, stops] = q.front(); q.pop();
            if (stops > k) continue;

            for (auto& [nei, price] : adj[node]) {
                if (cost + price < dist[nei]) {
                    dist[nei] = cost + price;
                    q.push({nei, cost + price, stops + 1});
                }
            }
        }
        return dist[dst] == INT_MAX ? -1 : dist[dst];
    }
};
```

Problem 4: Network Delay Time

Problem Statement: Given a list of travel times as directed edges, return the time it takes for all nodes to receive a signal from node `k`.

Approach:

- Use **Dijkstra's Algorithm** or **BFS-like traversal with early stopping**.
- Update minimum time to each node.

Code (Your Approach using BFS Queue):

```cpp
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<pair<int, int>> adj[n];
        for (auto& t : times)
            adj[t[0]-1].emplace_back(t[1]-1, t[2]);

        vector<int> dist(n, INT_MAX);
        queue<pair<int,int>> q;
        q.push({k-1, 0});
        dist[k-1] = 0;

        while (!q.empty()) {
            auto [node, cost] = q.front(); q.pop();
            for (auto [nei, w] : adj[node]) {
                if (dist[nei] > cost + w) {
                    dist[nei] = cost + w;
                    q.push({nei, dist[nei]});
                }
            }
        }

        int ans = *max_element(dist.begin(), dist.end());
        return ans == INT_MAX ? -1 : ans;
    }
};
```

5. Minimum Cost Path in a Grid

Problem:

Find the minimum cost path from the top-left cell to the bottom-right in a grid where each cell has a

cost.

Approach:

- Use Dijkstra's algorithm.
- Instead of distance, track the cumulative cost.
- Use a min-heap (priority queue) to get the next cell with the smallest cost.

Code with Path Reconstruction:

```
class Solution {
  public:
    struct Compare {
        bool operator()(const tuple<int, int, int>& a, const tuple<int, int, int>& b
            return get<2>(a) > get<2>(b); // Min-heap based on cost
        }
    };

    int minimumCostPath(vector<vector<int>>& grid) {
        int n = grid.size();
        vector<vector<int>> dist(n, vector<int>(n, INT_MAX));
        vector<vector<pair<int, int>>> parent(n, vector<pair<int, int>>(n, {-1, -1}))

        priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, Compare> 
        dist[0][0] = grid[0][0];
        pq.push({0, 0, dist[0][0]});

        int dir[] = {1, 0, -1, 0, 1};

        while (!pq.empty()) {
            auto [r, c, cost] = pq.top();
            pq.pop();

            if (cost > dist[r][c]) continue;

            for (int i = 0; i < 4; i++) {
                int nr = r + dir[i];
                int nc = c + dir[i + 1];

                if (nr >= 0 && nr < n && nc >= 0 && nc < n) {
                    int newCost = dist[r][c] + grid[nr][nc];
                    if (dist[nr][nc] > newCost) {
                        dist[nr][nc] = newCost;
                        parent[nr][nc] = {r, c};
                        pq.push({nr, nc, newCost});
                    }
                }
            }
        }
```

```
        // Path reconstruction
        vector<pair<int, int>> path;
        int r = n - 1, c = n - 1;
        while (r != -1 && c != -1) {
            path.push_back({r, c});
            tie(r, c) = parent[r][c];
        }
        reverse(path.begin(), path.end());

        cout << "Path: ";
        for (auto [x, y] : path) cout << "(" << x << "," << y << ") ";
        cout << endl;

        return dist[n - 1][n - 1];
    }
};
```

---

6. Dijkstra's Algorithm with Path Reconstruction

Problem:

Find the shortest path in a weighted undirected/directed graph from a source to all nodes.

Approach:

- Standard Dijkstra's using a priority queue.
- Track parent of each node for path reconstruction.

Code Sample:

```
class Solution {
public:
    vector<int> dijkstra(int n, vector<vector<pair<int,int>>>& adj, int src) {
        vector<int> dist(n, INT_MAX);
        vector<int> parent(n, -1);
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;

        dist[src] = 0;
        pq.push({0, src});

        while (!pq.empty()) {
            auto [d, u] = pq.top(); pq.pop();
            if (d > dist[u]) continue;
```

```
            for (auto [v, w] : adj[u]) {
                if (dist[v] > dist[u] + w) {
                    dist[v] = dist[u] + w;
                    parent[v] = u;
                    pq.push({dist[v], v});
                }
            }
        }

        // Print paths from source to each node
        for (int i = 0; i < n; i++) {
            if (dist[i] == INT_MAX) continue;
            vector<int> path;
            for (int at = i; at != -1; at = parent[at]) path.push_back(at);
            reverse(path.begin(), path.end());

            cout << "Path to node " << i << ": ";
            for (int p : path) cout << p << " ";
            cout << "\n";
        }

        return dist;
    }
};
```

Summary Table

Problem	Algorithm	Time Complexity	Path Printed
Shortest Clear Path (Matrix)	BFS	O(n^2)	No
Minimum Effort Path	Dijkstra (min-eff)	O(mn log mn)	✅ Yes
Cheapest Price with K Stops	BFS / Modified Dijkstra	O(EK)	No
Network Delay Time	BFS / Dijkstra	O(E log V)	No
