# 📘 Graph-Based Problems Summary - 4

## 🧭 1. Dijkstra's Algorithm (Undirected Graph)

### 🔹 Problem:

Find shortest path from source to all nodes in a **weighted undirected** graph.

### ✅ Approach:

- Build adjacency list.
- Use min-heap (priority_queue with `greater<>`) to store `{dist, node}`.
- Update distance if shorter path found.

### ✅ C++ Code:

```
vector<int> dijkstra(int V, vector<vector<int>> &edges, int src) {
    vector<pair<int,int>> adj[V];
    for (auto &e : edges) {
        int u = e[0], v = e[1], w = e[2];
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    }

    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        for (auto &[v, w] : adj[u]) {
            if (dist[v] > d + w) {
                dist[v] = d + w;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

---

## 🧭 2. Dijkstra on Directed Graph

Just remove reverse edge when building adjacency list:

```
adj[u].push_back({v, w});
```

---

## 🧭 3. Dijkstra with `set` (Faster Update)

### ✅ Why?

Efficient deletion of outdated entries vs `priority_queue`.

### ✅ Code:

```
set<pair<int,int>> s;
s.insert({0, src});

while (!s.empty()) {
    auto [d, u] = *s.begin(); s.erase(s.begin());
    for (auto &[v, w] : adj[u]) {
        if (dist[v] > d + w) {
            s.erase({dist[v], v});
            dist[v] = d + w;
            s.insert({dist[v], v});
        }
    }
}
```

---

## 🧭 4. Minimum Cost Path in Grid

### 🔹 Problem:

Find min cost from `(0,0)` to `(n-1,n-1)` in a grid of weights.

### ✅ Grid-Dijkstra Code:

```cpp
int minimumCostPath(vector<vector<int>>& grid) {
    int n = grid.size();
    vector<vector<int>> dist(n, vector<int>(n, INT_MAX));
    dist[0][0] = grid[0][0];

    priority_queue<tuple<int,int,int>, vector<tuple<int,int,int>>, greater<>> pq;
    pq.push({grid[0][0], 0, 0});
    int dir[] = {1,0,-1,0,1};

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

        for (int i = 0; i < 4; i++) {
            int nr = r + dir[i], nc = c + dir[i+1];
            if (nr >= 0 && nr < n && nc >= 0 && nc < n) {
                int newCost = cost + grid[nr][nc];
                if (dist[nr][nc] > newCost) {
                    dist[nr][nc] = newCost;
                    pq.push({newCost, nr, nc});
                }
            }
        }
    }

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

---

# 🧭 5. Word Ladder I – Shortest Transformation Sequence

## 🔹 Problem:

From `beginWord` to `endWord`, transform by changing one letter at a time, each intermediate word must be in wordList. Return minimum number of transformations.

## ✅ Approach:

- Use **BFS** starting from `beginWord`.
- For each word, change one character at a time to find neighbors in the dictionary.
- Track visited words to avoid cycles.

## ✅ Code:

```cpp
int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
```

```cpp
int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
    unordered_set<string> dict(wordList.begin(), wordList.end());
    if (!dict.count(endWord)) return 0;

    queue<string> q;
    q.push(beginWord);
    unordered_set<string> visited;
    visited.insert(beginWord);

    int level = 1;
    while (!q.empty()) {
        int sz = q.size();
        while (sz--) {
            string word = q.front(); q.pop();
            if (word == endWord) return level;

            for (int i = 0; i < word.size(); i++) {
                char original = word[i];
                for (char c = 'a'; c <= 'z'; c++) {
                    word[i] = c;
                    if (dict.count(word) && !visited.count(word)) {
                        visited.insert(word);
                        q.push(word);
                    }
                }
                word[i] = original;
            }
        }
        level++;
    }

    return 0;
}
```

---

# 🧭 6. Word Ladder II – All Shortest Transformation Sequences

## 🔹 Problem:

Same as above, but return **all** shortest transformation sequences.

## ✅ Approach:

1. **BFS** to build a **graph of predecessors** (`adjList`) and record levels.
2. **DFS/Backtrack** from endWord to beginWord using the graph.

## ✅ Code:

```
void dfs(string word, string beginWord, unordered_map<string, vector<string>>& paren
         vector<string>& path, vector<vector<string>>& res) {
    if (word == beginWord) {
        path.push_back(beginWord);
        reverse(path.begin(), path.end());
        res.push_back(path);
        reverse(path.begin(), path.end());
        path.pop_back();
        return;
    }

    path.push_back(word);
    for (string p : parent[word]) {
        dfs(p, beginWord, parent, path, res);
    }
    path.pop_back();
}

vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>&
    unordered_set<string> dict(wordList.begin(), wordList.end());
    unordered_map<string, vector<string>> parent;
    unordered_map<string, int> level;

    queue<string> q;
    q.push(beginWord);
    level[beginWord] = 0;

    int minLevel = INT_MAX;

    while (!q.empty()) {
        string word = q.front(); q.pop();
        int currLevel = level[word];
        if (currLevel > minLevel) break;

        for (int i = 0; i < word.size(); ++i) {
            string temp = word;
            for (char c = 'a'; c <= 'z'; ++c) {
                temp[i] = c;
                if (dict.count(temp)) {
                    if (!level.count(temp)) {
                        level[temp] = currLevel + 1;
                        q.push(temp);
                    }
                    if (level[temp] == currLevel + 1)
                        parent[temp].push_back(word);
                    if (temp == endWord) minLevel = currLevel + 1;
                }
            }
        }
```

```
    }

    vector<vector<string>> res;
    vector<string> path;
    if (parent.count(endWord))
        dfs(endWord, beginWord, parent, path, res);

    return res;
}
```

---

## 📦 Comparator for `priority_queue`

### ✅ For `tuple` (min-heap by cost):

```
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
    }
};
priority_queue<tuple<int,int,int>, vector<tuple<int,int,int>>, Compare> pq;
```

### ✅ Lambda (cleaner):

```
auto cmp = [](pair<int,int>& a, pair<int,int>& b) {
    return a.second > b.second;
};
priority_queue<pair<int,int>, vector<pair<int,int>>, decltype(cmp)> pq(cmp);
```

---

# ✅ Summary Table

| Problem | Technique | Key Tool / Structure |
|---|---|---|
| Dijkstra Undirected | Min-Heap | `priority_queue` |
| Dijkstra Directed | Min-Heap | Adjacency list (no back edge) |
| Dijkstra with Set | Balanced BST | `set<pair<int,int>>` |

| | | |
|---|---|---|
| Min Cost in Grid | Grid-Dijkstra | `priority_queue<tuple>` |
| Word Ladder I | BFS | Queue + Set |
| Word Ladder II | BFS + DFS Backtrack | Graph of parents |