Graphs are basically a collection of nodes connected by edges. Many ways to implement them, best implementation depends on context.
```cpp
struct Graph {
    struct Node {
        vector<Node*> arc;
    };
    deque<Node> nodes;
    int n = 0;

    Node* newNode() {
        nodes.push_back(Node());
        return &nodes.back();
    }
    void addArc(Node* a, Node* b) {
        a->arc.push_back(b);
    }
};
```
There are many ways to traverse graphs, the two most common and useful ones are:

1. Depth-first search - traverse deep and backtrack on dead-ends
2. Breadth-first search - from start traverse closer nodes first

DFS is usually implemented with recursion, BFS with a queue
Graph Traversal Code

Example

```cpp
void DFS(Node* cur) {
    cur->explored = true;
    for(Node* next : cur->arc)
        if(!next->explored)
            DFS(next);
}

void BFS(Node* start) {
    deque<Node*> que(1, start);
    start->explored = true;
    while(que.size() > 0) {
        Node* cur = que.front();
        que.pop_front();
        for(Node* next : cur->arc)
            if(!next->explored) {
                next->explored = true;
                que.push_back(next);
            }
    }
}
```
Dijkstra Algorithm

- Suppose we have edges with differing non-negative lengths and we want to find a shortest path from A to every other vertex.
- If all edges had equal length, we could use BFS.
Dijkstra Algorithm

Idea

- Keep track of explored nodes $E$, and frontier nodes $F$ (nodes we could directly reach from explored nodes).
- Denote $d(u, v)$ as the minimum distance between vertices $u$ and $v$. For each explored vertex $u$, the distance $d(\text{start}, u)$ has already been calculated.
- For each frontier node $v$, denote its potential $p(v)$ as $p(v) = \min\{d(\text{start}, u) + d(u, v) | u \in E\}$
- Pick the frontier node $v$ with minimal $p(v)$, mark that $d(\text{start}, v) = p(v)$, mark the node as explored and update the sets and potentials.
- Repeat until every node is explored.
Proof of Correctness

- Let’s use proof by induction
- Suppose that at some point in the algorithm among every $u \in E$, the $d(\text{start}, u)$ has been calculated correctly
- Let’s look at node $v$ with minimal $p(v)$. Suppose that $d(\text{start}, v) < p(v)$, however then there must exist some vertex $u$ such that $d(\text{start}, v) = d(\text{start}, u) + d(u, v)$ and $u \in F$, however then $p(u) < p(v)$ contradicting the minimality of $p(v)$.
- Therefore, since $p(v)$ corresponds to the distance of an actual path, $d(\text{start}, v) = p(v)$ and after marking $v$ as explored, the calculated distances are still correct
- Therefore the algorithm calculates the correct minimal distance from the starting node to every other node
Dijkstra Tips

- It’s more convenient to start by adding the starting point to frontier and marking its potential as 0.
- To make the picking of minimum potentials faster, use a heap.
- With heap, the time complexity will be $O(M \log N)$ where $M$ is the number of edges, and $N$ is the number of vertices.
Example

```cpp
void Dijkstra(Node* start) {
    // Note: priority_queue is a max heap. Use the inverses of potentials
    priority_queue<pair<int, Node*> > front;
    front.push({0, start});
    while(front.size() > 0) {
        int dist; Node* cur;
        tie(dist, cur) = front.top();
        front.pop();
        if(cur->explored)
            continue;
        cur->explored = true;
        cur->distance = -dist;
        for(Arc* curArc : cur->arc)
            front.push({dist - curArc->length, curArc->destination});
    }
}
```