Dijkstra's Algorithm

One limitation of a Breadth First Search in finding shortest distances is that each "move" or "edge" counts as a distance of 1. In real life maps though, it's possible that there's a road connecting location A to location B and it's length is say 3 miles, while the road connecting location B to location C is only 1 mile long. Formally, a weighted directed edge in a graph is an ordered triplet (u, v, w) indicating that it's possible to travel from vertex u to vertex v with a cost of w. (We typically call this cost the weight of the edge, which is my the variable w is traditionally used.)

Let's review the regular Breadth First Search and then highlight (in yellow), steps that would have to change if the connections between vertices also had weights attached to them. In most real life applications, all of these weights are non-negative, so for the purpose of this lecture, we'll assume that we're dealing with a graph that only has non-negative edge weights.

1. Create a distance array, marking the distance from the source vertex to every other vertex. Initialize this to all -1s to indicate that no path has been found. Set distance[source] = 0.
2. Create a queue (of vertices, which in practice is integers) adding the source vertex to the queue.
3. While the queue is not empty, do the following:
    a. Dequeue the front item of the queue, call this current.
    b. Loop through each neighboring item, next of current and do the following:
        i. If the item has not been visited yet (dist[next] == -1)
            1. Set dist[next] = dist[cur] + 1. (Need 1 step from cur…)
            2. Add next to the queue as a new place to explore from.
4. The state of the distance array at this point stores all shortest distances from source. If a stored distance is -1, that location isn't reachable from the source.

For reasons that will soon be mentioned, a regular queue doesn't suffice to solve the shortest distance problem correctly once we assign different weights to the edges.

It's easier to see that we can't just add 1 to dist[cur] to obtain dist[next]. It should be pretty clear that what we should do instead is:

dist[next] = dist[cur] + weight[cur][next]

where weight[cur][next] represents the weight of the edge starting at vertex cur and ending at vertex next.

The issue with this update is two-fold:

1. With different edge weights it's possible that this update would **increase** the value stored in dist[next].

2. When the very first update is made to dist[next], it's not guaranteed that this is the shortest distance.

Recall in the Breadth First Search, if we explore a vertex, u, that is distance k away from the source and find that u is connected to v and that we haven't reached v yet (which means that v is at least k+1 steps away from the source), then, with confidence, we can ascertain that the distance to v must be k+1, since we found a valid path that is k+1 steps and already KNOW that no valid path exists that is k steps long.

The problem with different edge weights is that we could have A → B → C, where A→B has weight 1 and B → C has weight 1,000. But what if A → D → C is such that A → D is 3 and D → C is 2. Clearly this second option is better, assuming our source is vertex A. The issue here is that initially, we would say dist[C] = 1001 because we first reached C from B.

Thus, unlike Breadth First Search, if we are to find a shortest distance in a weighted graph, we must accept that we might update a distance array entry multiple times. The first time we discover a path to a vertex doesn't necessarily represent the shortest path to that vertex. So, now the question is: when can we be sure that our distance estimate is correct?

(Edsger) Dijkstra proved that if we assume all edge weights are non-negative, the point in time we can prove that we have the shortest distance to a vertex is if its estimate is the shortest estimate of all the remaining estimates that haven't been proven to be minimal. **In particular, when we remove vertices from the queue, we need to remove the vertex which has the shortest current estimate left in the queue, not the first vertex that was placed in the queue.**

Luckily, there is a name for a queue which dequeues the smallest item (instead of the first one put in). It's called a Priority Queue and is built into C++ (priority_queue). You can add something to the priority_queue (push), access the top of it (top) and remove the top (pop). Not so coincidentally, these are the exact same method names used for the queue class in C++.

Now that we've identified the need to use a new data structure as well as delaying our decision that we have an optimal distance, we can sketch out the high level algorithm called Dijkstra's Algorithm:

```
1. Create a distance array, marking the distance from the source vertex to every other vertex.
   Initialize this to all -1s to indicate that no path has been found. Set distance[source] = 0.
2. Create a priority queue of estimates to vertices. In C++ we can implement this as a pair
   with the first value storing the distance estimate and the second value storing the index of
   the vertex.
3. Create a boolean vector, done, which marks whether or not the shortest distance to that
   vertex has been found. Initialize this array to all false.
4. While the priority queue is not empty and we don't have all shortest distances,
   a. Dequeue the front item of the priority queue, call this current.
   b. If we already have the shortest distance to current (done[current] is true), continue.
   c. Mark that we now have the shortest distance to current (done[current] = true)
   d. Loop through each edge from current to other locations (call these next)
      i. If dist[next] is -1 OR if dist[cur] + weight[cur][next] < dist[next]
         1. Set dist[next] = dist[cur] + weight[cur][next]
         2. Add (dist[next], next) to the priority queue.
```

The state of the distance array when step 4 completes is the correct shortest distances from the source vertex.

Dijkstra Trace Through

Traditional Dijkstra's Code

In this section we'll look at traditional Dijkstra code on a graph. Kattis has a question that just asks to compute the shortest distances from a source vertex to all other vertices in a weighted graph. Here is a link to the question:

https://open.kattis.com/problems/shortestpath1

Here is a solution to the problem. Some comments have been removed from the posted file online:

```
// Justin Almazan
// 6/25/2025
// Illustration of how to read in a graph as an adjacency list and run Dijkstra
// on it in C++. Edited BFS (to show similarities) from BFS camp lecture

using namespace std;
#include <iostream>
#include <vector>
#include <queue>

using pii = pair<int, int>;
const int INF = 2e9;

int numV;
vector< vector<pii> > graph;

int* dijkstra(int v);

int main() {

    int numE, numQ, s, v1, v2, w;
    cin >> numV >> numE >> numQ >> s;

    // Allow for multiple graphs
    while (numV + numE + numQ + s > 0) {

        graph.clear();
        graph.resize(numV);

        // Read in the edges - add one way.
        for (int i=0; i<numE; i++) {
            cin >> v1 >> v2 >> w;
            graph[v1].push_back({w, v2});
        }

        // Run Dijkstra from start vertex.
        int* dist = dijkstra(s);
        // Answer queries
        while (numQ--) {
            cin >> v1;
            if (dist[v1] != INF) cout << dist[v1] << endl;
            else cout << "Impossible" << endl;
        }

        cout << endl;
        delete [] dist;
        cin >> numV >> numE >> numQ >> s;
    }

    return 0;
}

int* dijkstra(int v) {

    // Set up distances.
    int* dist = new int[numV];
    for (int i=0; i<numV; i++) dist[i] = INF;
    dist[v] = 0;

    // Add source vertex to priority queue.
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    pq.push({0, v});

    // Run Dijkstra.
    while (pq.size() > 0) {

        // Get next.
        pii cur = pq.top(); pq.pop();

        // If we already found a better distance, skip!
        if (cur.first > dist[cur.second])
            continue;

        // Check all neighbors for new best paths.
        for    (vector<pii>::iterator    x    =    graph[cur.second].begin();
x<graph[cur.second].end(); x++) {

            // Found new best path
            if (dist[x->second] > dist[cur.second] + x->first) {

                // Update
                dist[x->second] = dist[cur.second] + x->first;

                // Insert into priority queue
                pq.push({dist[x->second], x->second});
            }
        }
    }

    // Return shortest distances from source to all vertices
    return dist;
}
```

A couple notes about the function in particular:

1. Since the priority queue has duplicate estimates for vertices, we need to skip over clearly suboptimal estimates when they get pulled from the priority queue. The reason we don't remove these earlier is that the priority queue data structures doesn't support removing arbitrary elements from it efficiently.

2. In this implementation, we don't use a done array, so it's possible that the main loop will continue running even after we have all the shortest distances. However, it's guaranteed that the total number of objects ever in the priority queue is not more than the total number of edges in the graph (since we enqueue at most one item per edge). Since it can be shown that each priority queue operation occurs in O(lg V) time, the total run time of Dijkstra's is O(ElgV). Thus, Dijkstra's should be able to run fast enough with up to about 10^6 edges and 10^5 vertices.

Kattis Problem: Big Truck
Here is a link to the problem:

Kattis Problem: Texas Summers
Here is a link to the problem:

Kattis Problem: Flowery Trails
Here is a link to the problem: