0-1 BFS

Table of Contents

It is well-known, that you can find the shortest paths between a single source and all other vertices in $O(|E|)$ using Breadth First Search in an **unweighted graph**, i.e. the distance is the minimal number of edges that you need to traverse from the source to another vertex. We can interpret such a graph also as a weighted graph, where every edge has the weight 1. If not all edges in graph have the same weight, that we need a more general algorithm, like Dijkstra which runs in $O(|V|^2 + |E|)$ or $O(|E| \log |V|)$ time.

However if the weights are more constrained, we can often do better. In this article we demonstrate how we can use BFS to solve the SSSP (single-source shortest path) problem in $O(|E|)$, if the weights of each edge is either 0 or 1.

Algorithm

We can develop the algorithm by closely study Dijkstra's algorithm and think about the consequences that our special graph implies. The general form of Dijkstra's algorithm is (here a `set` is used for the priority queue):

```
d.assign(n, INF);
d[s] = 0;
set<pair<int, int>> q;
q.insert({0, s});
while (!q.empty()) {
    int v = q.begin()->second;
    q.erase(q.begin());

    for (auto edge : adj[v]) {
        int u = edge.first;
        int w = edge.second;

        if (d[v] + w < d[u]) {
            q.erase({d[u], u});
            d[u] = d[v] + w;
            q.insert({d[u], u});
        }
    }
}
```

We can notice that the difference between the distances between the source `s` and two other vertices in the queue differs by at most one. Especially, we know that $d[v] \le d[u] \le d[v] + 1$ for each $u \in Q$. The reason for this is, that we only add vertices with equal distance or with distance plus one to the queue each iteration. Assuming there exists a u in the queue with

$d[u] - d[v] > 1$, then u must have been insert in the queue via a different vertex t with $d[t] \ge d[u] - 1 > d[v]$. However this is impossible, since Dijkstra's algorithm iterates over the vertices in increasing order.

This means, that the order of the queue looks like this:

$$Q = \underbrace{v}_{d[v]}, \dots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]+1} \dots \underbrace{n}_{d[v]+1}$$

This structure is so simple, that we don't need an actually priority queue, i.e. a balanced binary tree, is an overkill. We can simply use a normal queue, and append new vertices at the beginning if the corresponding edge has weight 0, i.e. if $d[u] = d[v]$, or at the end if the edge has weight 1, i.e. if $d[u] = d[v] + 1$. This way the queue still remains sorted at all time.

```
vector<int> d(n, INF);
d[s] = 0;
deque<int> q;
q.push_front(s);
while (!q.empty()) {
    int v = q.front();
    q.pop_front();
    for (auto edge : adj[v]) {
        int u = edge.first;
        int w = edge.second;
        if (d[v] + w < d[u]) {
            d[u] = d[v] + w;
            if (w == 1)
```

```
                    q.push_back(u);
                else
                    q.push_front(u);
            }
        }
    }
```

Dial's algorithm

We can extend this even further if we allow the weights of the edges to be even bigger. If every edge in the graph has a weight $\le k$, than the distances of vertices in the queue will differ by at most k from the distance of v to the source. So we can keep $k + 1$ buckets for the vertices in the queue, and whenever the bucket corresponding to the smallest distance gets empty, we make a cyclic shift to get the bucket with the next higher distance. This extension is called **Dial's algorithm**.

Practice problems

- CodeChef - Chef and Reversing