Dijkstra on sparse graphs

Table of Contents

For the statement of the problem, the algorithm with implementation and proof can be found on the article Dijkstra's algorithm.

Algorithm

We recall in the derivation of the complexity of Dijkstra's algorithm we used two factors: the time of finding the unmarked vertex with the smallest distance $d[v]$, and the time of the relaxation, i.e. the time of changing the values $d[\text{to}]$.

In the simplest implementation these operations require $O(n)$ and $O(1)$ time. Therefore, since we perform the first operation $O(n)$ times, and the second one $O(m)$ times, we obtained the complexity $O(n^2 + m)$.

It is clear, that this complexity is optimal for a dense graph, i.e. when $m \approx n^2$. However in sparse graphs, when m is much smaller than the maximal number of edges n^2, the complexity gets less optimal because of the first term. Thus it is necessary to improve the execution time of the first operation (and of course without greatly affecting the second operation by much).

To accomplish that we can use a variation of multiple auxiliary data structures. The most efficient is the **Fibonacci heap**, which allows the first operation to run in $O(\log n)$, and the second operation in $O(1)$. Therefore we will get the complexity $O(n \log n + m)$ for Dijkstra's algorithm, which is also the theoretical minimum for the shortest path search problem. Therefore this algorithm works optimal, and Fibonacci heaps are the optimal data structure. There doesn't exist any data structure, that can perform both operations in $O(1)$, because this would also allow to sort a list of random numbers in linear time, which is impossible. Interestingly there exists an algorithm by Thorup that finds the shortest path in $O(m)$ time, however only works for integer weights, and uses a completely different idea. So this doesn't lead to any contradictions. Fibonacci heaps provide the optimal complexity for this task. However they are quite complex to implement, and also have a quite large hidden constant.

As a compromise you can use data structures, that perform both types of operations (extracting a minimum and updating an item) in $O(\log n)$. Then the complexity

of Dijkstra's algorithm is $O(n \log m + m \log n) = O(m \log n)$.

C++ provides two such data structures: `set` and `priority_queue`. The first is based on red-black trees, and the second one on heaps. Therefore `priority_queue` has a smaller constant hidden constant, but also has a drawback: it doesn't support the operation of removing an element. Because of this we need to do a "workaround", that actually leads to a slightly worse factor $\log m$ instead of $\log n$ (although in terms of complexity they are identical).

Implementation

set

Let us start with the container `set`. Since we need to store vertices ordered by their values $d[]$, it is convenient to store actual pairs: the distance and the index of the vertex. As a result in a `set` pairs are automatically sorted by their distances.

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<i
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
```

```
    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 to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                q.erase({d[to], to});
                d[to] = d[v] + len;
                p[to] = v;
                q.insert({d[to], to});
            }
        }
    }
}
```

We don't need the array $u[]$ from the normal Dijkstra's algorithm implementation any more. We will use the `set` to store that information, and also find the vertex with the shortest distance with it. It kinda acts like a queue. The main loops executes until there are no more vertices in the set/queue. A vertex with the smallest distance gets extracted, and for each successful relaxation we first remove the old pair, and then after the relaxation add the new pair into the queue.

priority_queue

The main difference to the implementation with `set` is that we cannot remove elements from the `priority_queue` (although heaps can support that operation in theory). Therefore we have to cheat a little bit. We simply don't delete the old pair from the queue. As a result a vertex can appear multiple times with different distance in the queue at the same time. Among these pairs we are only interested in the pairs where the first element is equal to the corresponding value in $d[]$, all the other pairs are old. Therefore we need to make a small modification: at the beginning of each iteration, after extracting the next pair, we check if it an important pair or if it is already an old and handled pair. This check is important, otherwise the complexity can increase up to $O(nm)$.

By default a `priority_queue` sorts elements in descending order. To make it sort the elements in ascending order, we can either store the negated distances in it, or pass it a different sorting function. We will do the second option.

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<i
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
```

```
    d[s] = 0;
    using pii = pair<int, int>;
    priority_queue<pii, vector<pii>, greater<p
    q.push({0, s});
    while (!q.empty()) {
        int v = q.top().second;
        int d_v = q.top().first;
        q.pop();
        if (d_v != d[v])
            continue;

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

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
```

In practice the `priority_queue` version is a little bit faster than the version with `set`.

Getting rid of pairs

You can improve the performance a little bit more if you don't store pairs in the containers, but only the vertex

indices. In this case we must overload the comparison operator: it must compare two vertices using the distances stored in $d[]$.

As a result of the relaxation, the distance of some vertices will change. However the data structure will not resort itself automatically. If fact changing distances of vertices in the queue, might destroy the data structure. As before, we need to remove the vertex before we relax it, and then insert it again afterwards.

Since we only can remove from `set`, this optimization is only applicable for the `set` method, and doesn't work with `priority_queue` implementation. In practice this significantly increases the performance, especially when larger data types are used to store distances, like `long long` or `double`.