Maximum flow - Push-relabel algorithm

Table of Contents

The push-relabel algorithm (or also known as preflow-push algorithm) is an algorithm for computing the maximum flow of a flow network. The exact definition of the problem that we want to solve can be found in the article Maximum flow - Ford-Fulkerson and Edmonds-Karp.

In this article we will consider solving the problem by pushing a preflow through the network, which will run in $O(V^4)$, or more precisely in $O(V^2E)$, time. The

algorithm was designed by Andrew Goldberg and Robert Tarjan in 1985.

Definitions

During the algorithm we will have to handle a **preflow** - i.e. a function f that is similar to the flow function, but does not necessarily satisfies the flow conservation constraint. For it only the constraints

$$0 \le f(e) \le c(e)$$

and

$$\sum_{(v,u) \in E} f((v,u)) \ge \sum_{(u,v) \in E} f((u,v))$$

have to hold.

So it is possible for some vertex to receive more flow than it distributes. We say that this vertex has some excess flow, and define the amount of it with the **excess** function $x(u) = \sum_{(v,u) \in E} f((v,u)) - \sum_{(u,v) \in E} f((u,v))$.

In the same way as with the flow function, we can define the residual capacities and the residual graph with the

preflow function.

The algorithm will start off with an initial preflow (some vertices having excess), and during the execution the preflow will be handled and modified. Giving away some details already, the algorithm will pick a vertex with excess, and push the excess to neighboring vertices. It will repeat this until all vertices, except the source and the sink, are free from excess. It is easy to see, that a preflow without excess is a valid flow. This makes the algorithm terminate with an actual flow.

There are still two problem, we have to deal with. First, how do we guarantee that this actually terminates? And secondly, how do we guarantee that this will actually give us a maximum flow, and not just any random flow?

To solve these problems we need the help of another function, namely the **labeling** functions h, often also called **height** function, which assigns each vertex an integer. We call a labeling is valid, if $h(s) = |V|$, $h(t) = 0$, and $h(u) \leq h(v) + 1$ if there is an edge (u, v) in the residual graph - i.e. the edge (u, v) has a positive capacity in the residual graph. In other words, if it is possible to increase the flow from u to v, then the height

of v can be at most one smaller than the height of u, but it can be equal or even higher.

It is important to note, that if there exists a valid labeling function, then there doesn't exist an augmenting path from s to t in the residual graph. Because such a path will have a length of at most $|V| - 1$ edges, and each edge can decrease the height only by at most by one, which is impossible if the first height is $h(s) = |V|$ and the last height is $h(t) = 0$.

Using this labeling function we can state the strategy of the push-relabel algorithm: We start with a valid preflow and a valid labeling function. In each step we push some excess between vertices, and update the labels of vertices. We have to make sure, that after each step the preflow and the labeling are still valid. If then the algorithm determines, the preflow is a valid flow. And because we also have a valid labeling, there doesn't exists a path between s and t in the residual graph, which means that the flow is actually a maximum flow.

If we compare the Ford-Fulkerson algorithm with the push-relabel algorithm it seems like the algorithms are the duals of each other. The Ford-Fulkerson algorithm keeps a valid flow at all time and improves it until there

doesn't exists an augmenting path any more, while in the push-relabel algorithm there doesn't exists an augmenting path at any time, and we will improve the preflow until it is a valid flow.

Algorithm

First we have to initialize the graph with a valid preflow and labeling function.

Using the empty preflow - like it is done in the Ford-Fulkerson algorithm - is not possible, because then there will be an augmenting path and this implies that there doesn't exists a valid labeling. Therefore we will initialize each edges outgoing from s with its maximal capacity: $f((s, u)) = c((s, u))$. And all other edges with zero. In this case there exists a valid labeling, namely $h(s) = |V|$ for the source vertex and $h(u) = 0$ for all other.

Now let's describe the two operations in more detail.

With the **push** operation we try to push as much excess flow from one vertex u to a neighboring vertex v. We have one rule: we are only allowed to push flow from u to v if $h(u) = h(v) + 1$. In layman's terms, the excess

flow has to flow downwards, but not too steeply. Of course we only can push $\min(x(u), c((u, v)) - f((u, v)))$ flow.

If a vertex has excess, but it is not possible to push the excess to any adjacent vertex, then we need to increase the height of this vertex. We call this operation **`relabel`**. We will increase it by as much as it is possible, while still maintaining validity of the labeling.

To recap, the algorithm in a nutshell is: We initialize a valid preflow and a valid labeling. While we can perform push or relabel operations, we perform them. Afterwards the preflow is actually a flow and we return it.

Complexity

It is easy to show, that the maximal label of a vertex is $2|V| - 1$. At this point all remaining excess can and will be pushed back to the source. This gives at most $O(V^2)$ relabel operations.

It can also be showed, that there will be at most $O(VE)$ saturating pushes (a push where the total capacity of the edge is used) and at most $O(V^2E)$ non-saturating

pushes (a push where the capacity of an edge is not fully used) performed. If we pick a data structure that allows us to find the next vertex with excess in $O(1)$ time, then the total complexity of the algorithm is $O(V^2E)$.

Implementation

```
const int inf = 1000000000;

int n;
vector<vector<int>> capacity, flow;
vector<int> height, excess, seen;
queue<int> excess_vertices;

void push(int u, int v)
{
    int d = min(excess[u], capacity[u][v]
    flow[u][v] += d;
    flow[v][u] -= d;
    excess[u] -= d;
    excess[v] += d;
    if (d && excess[v] == d)
        excess_vertices.push(v);
}
```

```
void relabel(int u)
{
    int d = inf;
    for (int i = 0; i < n; i++) {
        if (capacity[u][i] - flow[u][i] >
            d = min(d, height[i]);
    }
    if (d < inf)
        height[u] = d + 1;
}

void discharge(int u)
{
    while (excess[u] > 0) {
        if (seen[u] < n) {
            int v = seen[u];
            if (capacity[u][v] - flow[u][v
                push(u, v);
            else
                seen[u]++;
        } else {
            relabel(u);
            seen[u] = 0;
        }
    }
}
```

```
int max_flow()
{
    height.assign(n, 0);
    height[0] = n;
    flow.assign(n, vector<int>(n, 0));
    excess.assign(n, 0);
    excess[0] = inf;
    for (int i = 1; i < n; i++)
        push(0, i);
    seen.assign(n, 0);

    while (!excess_vertices.empty()) {
        int u = excess_vertices.front();
        excess_vertices.pop();
        if (u != 0 && u != n - 1)
            discharge(u);
    }

    int max_flow = 0;
    for (int i = 0; i < n; i++)
        max_flow += flow[0][i];
    return max_flow;
}
```

Here we use the queue `excess_vertices` to store all vertices that currently have excess. In that way we can pick the next vertex for a push or a relabel operation in constant time.

And to make sure that we don't spend too much time finding the adjacent vertex to whom we can push, we use a data structure called **current-arc**. Basically we will iterate over the edges in a circular order and always store the last edge that we used. This way, for a certain labeling value, we will switch the current edge only $O(n)$ time. And since the relabeling already takes $O(n)$ time, we don't make the complexity worse.

27:4048/1836