Listing 1: Push Relabel

```
//  push relabel
//
//  Created by Felix Zieger and Janos Meny
//  This is free software.

#include <fstream>
#include <sstream>
#include <stdexcept>
#include <limits>
#include <iostream>
#include <ctime>
#include <algorithm>
#include <vector>

using FlowUnit = int;

class Network {
public:
    class Node;
    class Edge;
    Network(char const* filename);

    class Edge {
    public:
        Edge(int tail_, int head_, FlowUnit cap_, bool original_, int index_)
            : tail(tail_)
            , head(head_)
            , cap(cap_)
            , original(original_)
            , index(index_)
        {
        }
        FlowUnit cap;
        int sister;
        bool original;
        int index;
        int head;
        int tail;
    };

    class Node {
    public:
        Node(int height_, int exc_, Node* bPrev_)
            : height(height_)
            , exc(exc_)
            , bNext(bPrev_)
        {
        }
        Node* bNext; /* pointer to the next active node with same distance label
            */
        std::vector<Edge>::iterator current;
        int height;
        int exc;
    };

    std::vector<Node*> buckets; /* i'th entry stores a pointer to the first
       active node with distance label i  */
    std::vector<std::vector<Network::Edge> > adj; /* array of edges, sorted by
       tail */
```

```
std::vector<Network::Node> nodes; /* pretty self explanatory */

int aMax; /* maximum active node label */
int n; /*number of nodes*/
Node* sentinelNode; /*marker for the end of nodes lists*/

//v must be the first element in the bucket;
void aRemove(int b, Node* v)
{
    buckets[b] = v->bNext;
}

void aAdd(int b, Node* v)
{
    v->bNext = buckets[b];
    buckets[b] = v;
    if (v->height > aMax)
        aMax = v->height;
}

void push(Edge& e)
{
    FlowUnit gamma = std::min(nodes[e.tail].exc, e.cap);
    e.cap -= gamma;
    adj[e.head][e.sister].cap += gamma;
    if (nodes[e.head].exc == 0 and e.head != 1 and e.head != 0) {
        aAdd(nodes[e.head].height , &nodes[e.head]);
    }
    nodes[e.head].exc += gamma;
    nodes[e.tail].exc -= gamma;
}

void relabel(int v)
{
    int min = 2 * n;
    for (auto e = adj[v].begin(); e != adj[v].end(); e++) {
        if (e -> cap > 0) {
            if(nodes[e -> head].height+1<min){
                min = nodes[e -> head].height + 1;
                nodes[v].current = e;
            }
        }
    }
    nodes[v].height = min;
}

void discharge(int v)
{
    do {
        for (auto e = nodes[v].current; e !=  adj[v].end(); e++) {
            if (e -> cap > 0 and nodes[e -> head].height + 1 == nodes[v].
               height) {
                push(*e);
                if (nodes[v].exc == 0)
                    nodes[v].current = adj[v].begin();
                    break;
            }
        }
        if (nodes[v].exc > 0) {
            relabel(v);
        }
```

```
            else {
                break;
            }
        } while (true);
    }

    void initialize()
    {
        for (auto& e : adj[0]) {
            if (e.cap > 0) {
                FlowUnit gamma = e.cap;
                nodes[e.head].exc += gamma;
                adj[e.head][e.sister].cap += gamma;
                e.cap -= gamma;
            }
        }
        aMax = 0;

        for (int i = 0; i < n; i++) {
            nodes[i].current = adj[i].begin();
            if (i == 0) {
                nodes[i].height = n;
            }
            else if (i == 1) {
                nodes[i].height = 0;
            }
            else if (nodes[i].exc > 0) {
                aAdd(1, &nodes[i]);
            }
        }
    }
    //sorting the flow-array by index
    static bool pairCompare(const std::pair<int, int>& a, const std::pair<int,
       int>& b)
    {
        return a.first < b.first;
    }

    void print()
    {
        std::cout << nodes[1].exc << std::endl;
        std::vector<std::pair<int, int> > flow;

        for (int v = 0; v < n; v++) {
            for (auto& e : adj[v]) {
                if (e.original and adj[e.head][e.sister].cap > 0) {
                    flow.push_back(std::pair<int, int>(e.index, adj[e.head][e.
                        sister].cap));
                }
            }
        }
        std::sort(flow.begin(), flow.end(), pairCompare);
        for (auto pair : flow) {
            //std::cout << pair.first << " " << pair.second << std::endl;
        }
    }
};

void push_relabel(Network& g_f)
{
    g_f.initialize();
    while (g_f.aMax >= 0) {
        if (g_f.buckets[g_f.aMax] == g_f.sentinelNode) {
            g_f.aMax--;
        }
        else {
            Network::Node* v = g_f.buckets[g_f.aMax];
            g_f.aRemove(g_f.aMax, v);
            g_f.discharge(v - &g_f.nodes[0]);
        }
    }
}

int main(int argc, char* argv[])
{
    clock_t begin = std::clock();
    if (argc > 1) {
        Network g_f(argv[1]);
        push_relabel(g_f);
        g_f.print();
    }
    clock_t end = std::clock();
    double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
    std::cout << elapsed_secs << std::endl;
}

Network::Network(char const* filename)
{
    std::ifstream file(filename); // open file
    if (not file) {
        throw std::runtime_error("Cannot␣open␣file.");
    }

    int num = 0;
    std::string line;
    std::getline(file, line); // get first line of file
    std::stringstream ss(line); // convert line to a stringstream
    ss >> num; // for which we can use >>
    if (not ss) {
        throw std::runtime_error("Invalid␣file␣format.");
    }
    n = num;
    sentinelNode = new Node(0, 0, NULL);
    nodes.assign(num, Node(1, 0, sentinelNode));
    buckets.assign(2 * num, sentinelNode);
    adj.resize(num);

    int index = 0;
    while (std::getline(file, line)) {
        std::stringstream ss(line);
        int head, tail;
        ss >> tail >> head;
        if (not ss) {
            throw std::runtime_error("Invalid␣file␣format.");
        }
        FlowUnit u = 1;
        ss >> u;
        if (tail != head) {

            adj[tail].push_back(Edge(tail, head, u, true, index));
            adj[head].push_back(Edge(head, tail, 0, false, index + num));
            adj[tail].back().sister = adj[head].size() - 1;
```

```cpp
            adj[head].back().sister = adj[tail].size() - 1;
            index++;
        }
        else {
            throw std::runtime_error("Invalid file format: loops not allowed.");
        }
    }
}
```