МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЁТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Потоки в сети

Студент гр. 8303 _____ Парфентьев Л.М.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Изучение алгоритма Форда-Фалкерсона поиска максимального потока в сети.

Задание

Вариант 6

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Сеть (ориентированный взвешенный граф) представляется в виде триплета из имён вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:

N – количество ориентированных рёбер графа

v_0 – исток

v_n – сток

v_i v_j ω_{ij} – ребро графа

v_i v_j ω_{ij} – ребро графа

…

Выходные данные:

$P_{\max}$ – величина максимального потока

v_i v_j ω_{ij} – ребро графа с фактической величиной протекающего потока

v_i v_j ω_{ij} – ребро графа с фактической величиной протекающего потока

…

В ответе выходные рёбра отсортируйте в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные рёбра, даже если поток в них равен 0).

Sample input:

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
```

Sample output:

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

Вариант 6

Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, соединяющей вершины, имена которых в алфавите ближе всего друг к другу. Если таких дуг несколько, то выбрать ту, имя конца которой в алфавите ближайшее к началу алфавита.

Описание алгоритма

Алгоритм Форда-Фалкерсона заключается в следующем:

1. Пытаемся найти путь в графе. Если путь найти не удаётся, алгоритм завершает работу.

2. Ищем максимальный поток через найденный путь; дял этого надо найти дугу с наименьшей пропускной способностью.

3. Уменьшаем пропускную способность каждой дуги пути на величину потока через путь, и увеличиваем на ту же величину пропускную способность противоположной дуги.

Если у дуги нет противоположной, считаем, что она есть, но её пропускная способность изначально равна нулю.

Путь можно брать любой, но в данном случае используется конкретное правило: на каждом шаге из всех дуг, ведущих из уже посещённой вершины в ещё не посещённую выбирается дуга с наименьшей разностью номеров соединяемых ею вершин (либо, если таких несколько, то с наименьшим номером новой вершины). Если вершины обозначаются буквами, в качестве номера вершины берётся код этой буквы.

Описание функций и структур данных

Для хранения рёбер и узлов в графе используются типы `Edge` и `Node` соответственно. Тем не менее, для обозначения узлов используются коды их символов, а для обозначения рёбер – их номера в векторе исходящих рёбер узла.

У класса `Edge` есть следующие поля:

`int dest` узел, в который ведёт ребро.

`int max_flux` начальная (максимальная) пропускная способность ребра (отрицательна для обратных рёбер).

`int current_flux` текущий поток через ребро.

`int rev_idx` номер обратного ребра в узле `dest`.

При добавлении в граф нового ребра из v_1 в v_2 с пропускной способностью ω на самом деле создаются 2 ребра: прямое с `current_flux` $= 0$, и обратное (из v_2 в v_1) с `max_flux` $= -\omega$ и `current_flux` $= \omega$. Изначально обратное ребро будет иметь текущую пропускную способность 0.

Функция `Edge::avail() const` вычисляет текущую пропускную способность ребра. Пропускная способность всегда неотрицательна. Поскольку для обратных рёбер `max_flux` < 0, `max_flux` берётся по модулю.

Функция `Edge::real() const` проверяет, что ребро прямое, т.е. настоящее. Это необходимо для того, чтобы пропускать обратные рёбра при выводе ответа.

У класса `Node` есть следующие поля и функции:

`std::vector<Edge> edges` вектор исходящих из данного узла рёбер.

`Edge &edge(int e)` **(и `const`)** выбор ребра по номеру.

`int edges_count() const` возвращает количество рёбер, исходящих из узла.

Для представления путей используется вектор номеров узлов. По такому вектору и номеру начальной вершины можно восстановить весь маршрут, а в данном случае все пути идут из одной и той же начальной вершины (истока).

Для удобства для типа пути объявлено имя `Path`.

Основной класс, в котором реализован алгоритм – `Graph`:

`std::vector<Node> nodes` вектор узлов графа.

`int start, end` номера начального и конечного узлов графа.

`int base_char` наименьший номер узла. Узел с этим номером будет находиться в векторе по индексу 0. При считывании графа этот номер может уменьшаться, при этом в начало вектора будут добавляться новые узлы.

Отделять номера узлов от их индексов нужно, т.к. неизвестен диапазон символов, которыми будут обозначаться узлы. В примере они обозначаются буквами в нижнем регистре, но могут обозначаться, например, цифрами.

`Node &node(int n)` (и `const`) выбор узла по его номеру (не индексу).

`int node_char(int idx) const` получение номера узла по его индексу.

`int node_index(int n) const` получение индекса узла по его номеру. Эта функция нужна, чтобы иметь возможность хранить временные массивы, где каждый элемент по соответствует узлу с тем же индексом.

`int nodes_count() const` возвращает количество узлов в графе.

`Edge &edge(int n, int e)` (и `const`) выбор ребра `e` из узла `n`.

`Edge &revedge(int n, int e)` (и `const`) выбор обратного ребра для ребра из узла `n` с номером `e`.

`void add_node(int n)` добавление нового узла с номером `n`. Именно добавление происходит только если номер этого узла меньше или больше всех, что были добавлены ранее.

`void add_edge(int n1, int n2, int w)` добавление ребра из узла `n1` в узел `n2`, с пропускной способностью `w`, и обратного ему. Прямое ребро добавляется в узел `n1`, а обратное – в `n2`.

`void mod_edge(int n, int e, int dw)` увеличение величины текущего потока через ребро `e` из узла `n` на `dw` (т.е. уменьшение текущей пропускной способности через это ребро на $-$`dw`). Текущий поток через обратное ребро изменяется на $-$`dw`.

`int path_flux(const Path &p) const` возвращает максимальный поток через путь `p`.

`void apply_flux(const Path &p, int f)` изменяет текущий поток через все рёбра пути `p` на `f`.

`int get_max_flux()` возвращает максимальный поток через граф, при этом находя фактический поток через каждое ребро графа.

Поиск пути в графе выполняется в функции `find_path`. Для неё определён вспомогательный класс `EdgeCmpRef`, при помощи которого выбирается следующее ребро. Рёбра хранятся в очереди с приоритетами.

Сама функция `find_path` просто прописывает, откуда мы приходим в каждый узел в первый раз. Это обозначается номером ребра, обратного тому, через которое мы в это вершину пришли. Функция `recover_path` восстанавливает путь.

Сложность алгоритма

На каждом шаге мы ищем путь и ещё несколько раз его проходим. Данная реализация выполняет поиск пути за $O(|E| \log |E|)$ (т.к. для хранения набора доступных рёбер используется очередь с приоритетами), пройти путь можно за $O(|E|)$. Таким образом, можно сказать, что каждый шаг выполняется за $O(|E| \log |E|)$.

В худшем случае, на каждом шаге мы находим путь с потоком 1; тогда число шагов, которое нам понадобится, равно максимальному потоку через граф f. Получим сложность по времени $O(f|E| \log |E|)$.

Для хранения графа требуется $O(|V| + |E|)$ памяти. Требуется дополнительная память для поиска пути (очередь вершин, два массива дополнительной информации об узлах, а также сам путь) , но это тоже $O(|V|+|E|)$. Таким образом, сложность по памяти $O(|V| + |E|)$.

Тестирование

На рисунках слева входные данные, а справа вывод – фактический поток через граф.

- Простой тест.

  Ввод:

```
5
a
d
a c 1
a b 1
b d 1
c d 1
b c 1
```

  Вывод:

```
2
a b 1
a c 1
b c 0
b d 1
c d 1
```

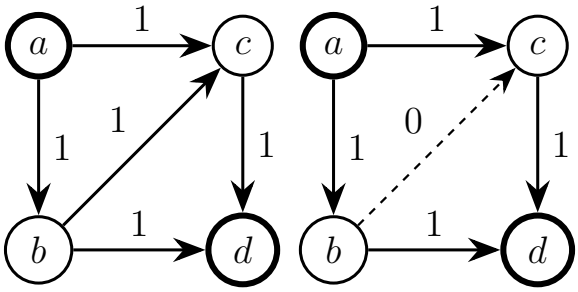

Рис. 1: Ввод/вывод 1

- Тест из задания.

  Ввод:

```
7
a
f
```

```
a b 7
b d 6
d e 3
e c 2
a c 6
c f 9
d f 4
```

Вывод:

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

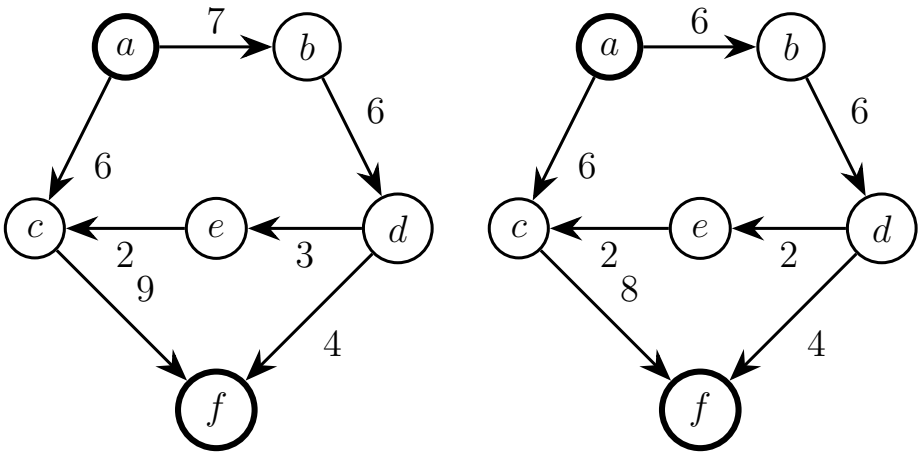

Рис. 2: Ввод/вывод 2

- Тест 3

  Ввод:

```
11
a
h
a b 3
b e 1
a c 1
c e 2
```

```
a d 2
d e 4
e g 3
e f 2
f h 3
g h 1
d f 1
```

Вывод:

```
4
a b 1
a c 1
a d 2
b e 1
c e 1
d e 1
d f 1
e f 2
e g 1
f h 3
g h 1
```

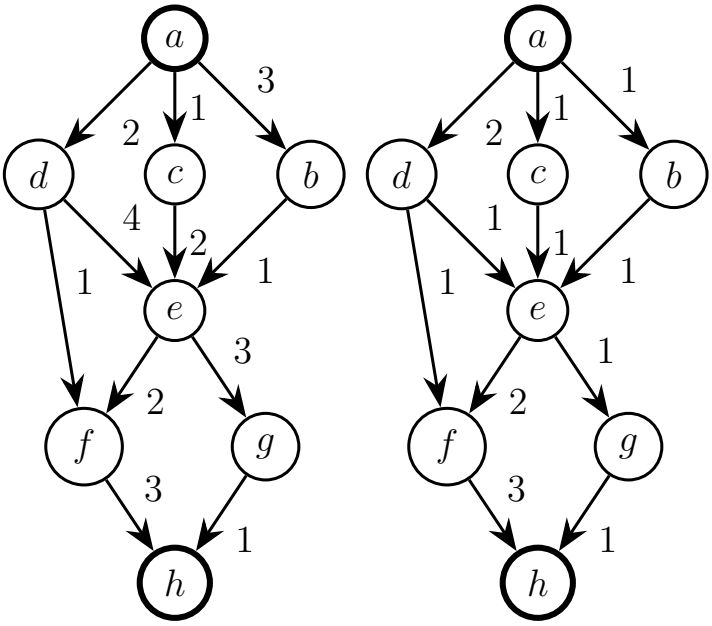

Рис. 3: Ввод/вывод 3

- Тест 4

  Ввод:

```
10
a
f
a b 7
a c 5
c d 7
b d 2
d e 6
b e 3
d f 8
e f 8
b c 4
a d 4
```

Вывод:

```
16
a b 7
a c 5
a d 4
b c 2
b d 2
b e 3
c d 7
d e 5
d f 8
e f 8
```

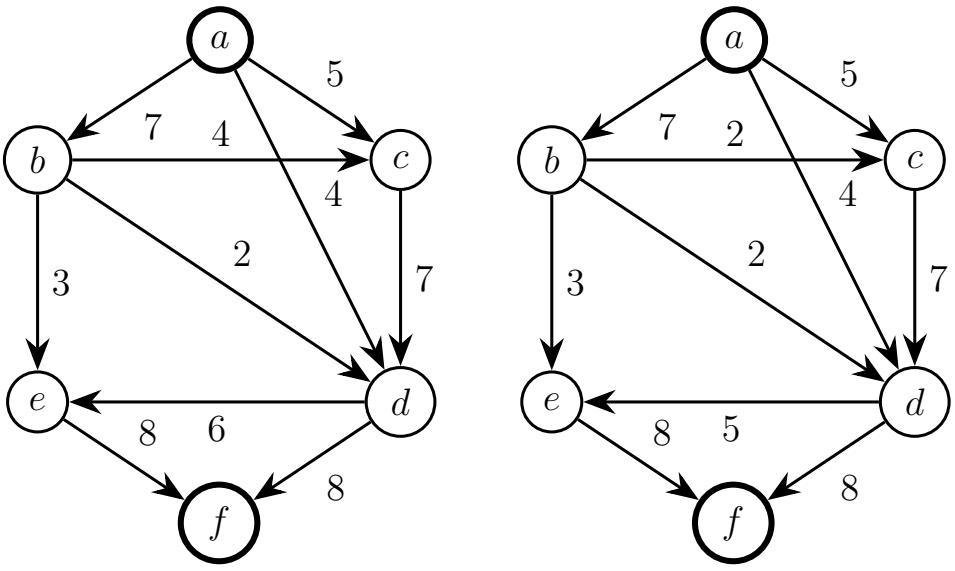

Рис. 4: Ввод/вывод 4

Выводы

В ходе выполнения лабораторной работы исследован алгоритм Форда-Фалкерсона для поиска максимального потока в графе.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab3.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <functional>

#include <limits.h>

// Note: define ENABLE_DEBUG to enable debugging output to stderr


struct Edge {
    int dest;                   // ‘node_char’s, not indices
    int max_flux, current_flux;
    int rev_idx;                // index of the corresponding dest
                                // node’s edge

    int avail() const { return std::abs(max_flux) - current_flux; }

    // We don’t want to write out auxiliary reverse edges which have
    // negative max_flux
    bool real() const { return max_flux > 0; }
};

struct Node {
    std::vector<Edge> edges;

    Edge &edge(int e) { return edges[e]; }
    const Edge &edge(int e) const { return edges[e]; }

    int edges_count() const { return edges.size(); }
};

// Edge indices, assuming we start from g.start and finish at g.end
using Path = std::vector<int>;

std::ostream & write_node(std::ostream &os, int n);

struct Graph {
    std::vector<Node> nodes;
    int start, end;             // ‘node_char’s

    // We don’t assume any particular range of characters. Instead, we
    // use characters we read as identifiers but only keep nodes in the
    // character range actually used. ‘base_char’ is the lowest
```

```
    // character used as a node identifier so far.
    int base_char = -1;

    Graph() :nodes{}, start{-1}, end{-1} {}

    Node &node(int n) { return nodes[n - base_char]; }
    const Node &node(int n) const { return nodes[n - base_char]; }

    // node_char: external id of a node
    int node_char(int idx) const { return idx + base_char; }
    // node_index: zero-based index of node
    int node_index(int n) const { return n - base_char; }

    int nodes_count() const { return nodes.size(); }

    Edge &edge(int n, int e) { return node(n).edge(e); }
    const Edge &edge(int n, int e) const { return node(n).edge(e); }
    Edge &revedge(int n, int e)
    {
        Edge &ed = edge(n, e);
        return edge(ed.dest, ed.rev_idx);
    }
    const Edge &revedge(int n, int e) const
    {
        const Edge &ed = edge(n, e);
        return edge(ed.dest, ed.rev_idx);
    }

    void add_node(int n);

    void add_edge(int n1, int n2, int w);
    void mod_edge(int n, int e, int dw);

    int path_flux(const Path &p) const;
    void apply_flux(const Path &p, int f);

    int get_max_flux();
};

void
Graph::add_node(int n)
{
    if (base_char < 0) {
        // First node
        base_char = n;
    } else if (n < base_char) {
        // Node below lowest char seen before: prepend nodes
        nodes.insert(nodes.begin(), base_char - n, Node{});
        base_char = n;
    } else {
        int idx = n - base_char,
            cur = nodes_count();
```

```
        // If new char is above every one seen before, append nodes
        if (idx >= cur)
            nodes.insert(nodes.end(), idx - cur + 1, Node{});
    }
}

void
Graph::add_edge(int n1, int n2, int w)
{
    int e1idx = node(n1).edges_count(),
        e2idx = node(n2).edges_count();

    // Add 2 interconnected (via dest and rev_idx) edges at once.
    node(n1).edges.push_back(Edge{n2, w, 0, e2idx});
    node(n2).edges.push_back(Edge{n1, -w, w, e1idx});

#ifdef ENABLE_DEBUG
    write_node(write_node(std::cerr << "add edge: ", n1), n2);
    write_node(write_node(std::cerr << '/', n2), n1);
    std::cerr << " weight: " << w << std::endl;
#endif
}

void
Graph::mod_edge(int n, int e, int dw)
{
    Edge
        &e1 = edge(n, e),
        &e2 = revedge(n, e);

    e1.current_flux += dw;
    e2.current_flux -= dw;

#ifdef ENABLE_DEBUG
    int d = e1.dest;
    std::cerr << "modify (by " << dw << ") edges: ";
    write_node(write_node(std::cerr, n), d)
        << " (new:" << e1.current_flux << " avail:"
        << e1.avail() << "), ";
    write_node(write_node(std::cerr, d), n)
        << " (new:" << e2.current_flux << " avail:"
        << e2.avail() << ")" << std::endl;
#endif
}

int
Graph::path_flux(const Path &p) const
{
    int f = INT_MAX;

    // Traverse path, find min available flux
```

```
    int n = start;
    auto iter = p.begin();
    for (; n != end;
         n = edge(n, *iter++).dest)
        f = std::min(f, edge(n, *iter).avail());

    return f;
}

// 'apply': 'consume' that much flux from each edge in the path
void
Graph::apply_flux(const Path &p, int f)
{
    int n = start;
    auto iter = p.begin();
    for (; n != end;
         n = edge(n, *iter++).dest)
        mod_edge(n, *iter, f);
}

std::ostream &
debug_write_path(const Graph &g,
                 std::ostream &os,
                 const Path &p)
{
    int n = g.start;
    auto iter = p.begin();
    for (; n != g.end;
         n = g.edge(n, *iter++).dest)
        write_node(os, n);
    write_node(os, n);

    return os;
}

Path find_path(const Graph &g);

int
Graph::get_max_flux()
{
    int total = 0;

    for (;;) {
        Path p = find_path(*this);
        if (p.empty())
            break;

        int f = path_flux(p);
        apply_flux(p, f);
        total += f;

#ifdef ENABLE_DEBUG
```

```
        debug_write_path(*this, std::cerr << "found path: ", p);
        std::cerr << ", flux: " << f << std::endl;
        std::cerr << "current total: " << total << std::endl;
#endif
    }

    return total;
}




Path
recover_path(const Graph &g,
             const std::vector<int> &revs)
{
    Path p {};

    // Traverse the path from end to start following recorded reverse
    // edges
    for (int n = g.end; n != g.start;) {
        const Edge &e = g.edge(n, revs[g.node_index(n)]);
        p.push_back(e.rev_idx);
        n = e.dest;
    }

    std::reverse(p.begin(), p.end());

    return p;
}

struct EdgeCmpRef {
    int src;
    int idx;

    const Edge &edge(const Graph &g) const { return g.edge(src, idx); }

    int dst(const Graph &g) const { return edge(g).dest; }
    int rev_idx(const Graph &g) const { return edge(g).rev_idx; }

    // std::less<int> / std::greater<int>
    template<typename C>
    bool compare(const EdgeCmpRef &o,
                 const Graph &g,
                 C cmp) const
    {
        int dst1 = dst(g),
            dst2 = o.dst(g);
        int diff1 = std::abs(dst1 - src),
            diff2 = std::abs(dst2 - o.src);

        if (diff1 != diff2)
```

```
            return cmp(diff1, diff2);
        return cmp(dst1, dst2);
    }
};

Path
find_path(const Graph &g)
{
    auto edgecmpref_less =
        [&g](const EdgeCmpRef &a, const EdgeCmpRef &b)
        {
            // std::greater to reverse the std::priority_queue order
            return a.compare(b, g, std::greater<int>{});
        };

    // We'll always have the best edge on top
    std::priority_queue<EdgeCmpRef,
                        std::vector<EdgeCmpRef>,
                        decltype(edgecmpref_less)> q {edgecmpref_less};

    std::vector<bool> visited (g.nodes_count(), false);

    // rev[g.node_index(n)]: which edge of node n we should use to
    // return to the node from which we came to 'n' the first time
    std::vector<int> rev (g.nodes_count(), -1);

    int n = g.start;
    while (n != g.end) {
        visited[g.node_index(n)] = true;

        int ec = g.node(n).edges_count();
        for (int ei = 0; ei < ec; ei++) {
            const Edge &e = g.edge(n, ei);
            if (!visited[g.node_index(e.dest)]
                && e.avail() > 0) {
                q.push(EdgeCmpRef{n, ei});

#ifdef ENABLE_DEBUG
                write_node(std::cerr << " pushing edge: ", n) << " -> ";
                write_node(std::cerr, g.edge(n, ei).dest) << std::endl;
#endif
            }
        }

        // No more edges but we haven't seen g.end => no path
        if (q.empty())
            return Path{};

        // Pick the next edge
        // n <- next node
        // r <- reverse edge
        int r;
```

```
        do {
            const EdgeCmpRef &er = q.top();

            r = er.rev_idx(g);
            n = er.dst(g);

            q.pop();

#ifdef ENABLE_DEBUG
            write_node(std::cerr << " looking at edge: ", er.src);
            write_node(std::cerr << " -> ", n) << std::endl;
#endif
        } while (visited[g.node_index(n)]);

#ifdef ENABLE_DEBUG
        std::cerr << " ok\n";
#endif

        rev[g.node_index(n)] = r;
    }

    return recover_path(g, rev);
}




// Read a node character and make sure it's valid in the graph
int
read_node(std::istream &is,
          Graph &g)
{
    char c;
    is >> c;

    g.add_node(c);

    return c;
}

Graph
read_graph(std::istream &is)
{
    Graph g {};

    int count;
    is >> count;

    g.start = read_node(is, g);
    g.end = read_node(is, g);

    for (int i = 0; i < count; i++) {
```

```
        int n1 = read_node(is, g),
            n2 = read_node(is, g),
            weight;
        is >> weight;

        g.add_edge(n1, n2, weight);
    }

    return g;
}



std::ostream &
write_node(std::ostream &os, int n)
{
    return os << static_cast<char>(n);
}

std::ostream &
write_edge(std::ostream &os,
           const Graph &g, int n, int e)
{
    // Format: "{from} {to} {actual_flux}"
    const Edge &edge = g.edge(n, e);
    return write_node(write_node(os, n) << " ", edge.dest)
        << " " << edge.current_flux << std::endl;
}

// Write edges sorted by destination node character
std::ostream &
write_node_edges(std::ostream &os,
                 const Graph &g, int n)
{
    struct EdgeRef {
        int dest;
        int idx;

        bool operator<(const EdgeRef &er) const
        {
            return dest < er.dest;
        }
    };
    std::vector<EdgeRef> ers {};

    int c = g.node(n).edges_count();
    for (int i = 0; i < c; i++) {
        const Edge &edge = g.edge(n, i);
        if (edge.real())
            ers.push_back(EdgeRef{edge.dest, i});
    }
```

```
    std::sort(ers.begin(), ers.end());

    for (const EdgeRef &er : ers)
        write_edge(os, g, n, er.idx);

    return os;
}

std::ostream &
write_flux(std::ostream &os,
           const Graph &g)
{
    // Note: we iterate over node indices, so we use g.node_char
    int c = g.nodes_count();
    for (int i = 0; i < c; i++)
        write_node_edges(os, g, g.node_char(i));
    return os;
}




int
main(void)
{
    Graph g = read_graph(std::cin);
    int flux = g.get_max_flux();
    write_flux(std::cout << flux << std::endl, g);

    return 0;
}
```