

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

ОТЧЕТ
по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Потoki в сети

Студент гр. 8382

Щеглов А.С.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучение работы алгоритма Форда-Фалкерсона для нахождения максимального потока в сети.

Задание.

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

$v_i v_j \omega_{ij}$ - ребро графа

$v_i v_j \omega_{ij}$ - ребро графа

...

Выходные данные:

$v_i v_j \omega_{ij}$ - ребро графа с фактической величиной протекающего потока

$v_i v_j \omega_{ij}$ - 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).

Вариант дополнительного задания.

Вар. 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 из истока в сток. Если путь найти не удаётся, алгоритм завершает работу.
2. Ищем максимальный поток через найденный путь; для этого надо найти ребро с наименьшей пропускной способностью.
3. Уменьшаем пропускную способность каждого ребра пути на величину потока через путь, и увеличиваем на ту же величину пропускную способность противоположного ребра.

Если у ребра нет противоположного, считаем, что оно есть, но его пропускная способность изначально равна нулю.

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

Алгоритм завершает работу когда не удастся найти очередной путь в графе. Он всегда находит решение на графе с неотрицательными целочисленными весами.

Описание функций и структур данных

Для хранения рёбер и узлов в графе используются типы *Edge* и *Vertex* соответственно. Тем не менее, для обозначения узлов используются коды их символов, а для обозначения рёбер – их ID (тип *edge_id_t*). Предполагается, что ID можно перебирать, и что следующий ID можно получить через

оператор инкремента. Это верно для индексов, и допускает использование итераторов из STL (как минимум, из категории `ForwardIterator`), например, итераторов `std::forward_list`. В данной реализации ID являются индексами в векторах исходящих рёбер в узлах.

У класса *Edge* есть следующие поля:

int dest узел, в который ведёт ребро.

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

int current_flux текущий поток через ребро.

int rev ID обратного ребра в узле *dest*.

При добавлении в граф нового ребра из *v1* в *v2* с пропускной способностью ω на самом деле создаются 2 ребра: прямое с *current_flux* = 0, и обратное (из *v2* в *v1*) с *max_capacity* = $-\omega$ и *current_flux* = ω .

Изначально обратное ребро будет иметь текущую пропускную способность 0. Функция *Edge::remaining_capacity() const* возвращает текущую пропускную способность ребра. Пропускная способность всегда неотрицательна. Поскольку для обратных рёбер *max_capacity* < 0, *max_flux* берётся по модулю.

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

У класса *Vertex* есть следующие поля и функции:

std::vector<Edge> edges вектор исходящих из данного узла рёбер (приватное поле).

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

edge_id_t add_edge() добавляет исходящее ребро в узел и возвращает его ID.

edge_id_t begin_id() const ID, с которого осуществляется перебор исходящих рёбер.

edge_id_t end_id() const ID, на котором нужно завершить перебор, т.е. ID, идущий сразу за последним.

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<Vertex> vertexes 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.

Vertex &vertex(int v) (и *const*) выбор узла по его номеру (не индексу).

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

int vertex_index(int v) const получение индекса узла по его номеру.

Эта функция нужна, чтобы иметь возможность хранить временные массивы, где каждый элемент по соответствию узлу с тем же индексом.

int vertexes_count() const возвращает количество узлов в графе.

Edge &edge(int v, edge_id_t e) (и *const*) выбор ребра *e* из узла *v*.

Edge &revedge(int v, edge_id_t e) (и *const*) для ребра *e* из узла *v* возвращает обратное.

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

void add_edge(int v1, int v2, int cap) добавление ребра из *v1* в *v2* с максимальной пропускной способностью *cap*, а также обратного ему. Прямое ребро добавляется в узел *v1*, а обратное – в *v2*.

void mod_edge(int v, edge_id_t e, int dcap) увеличение значения текущего потока через ребро *e* из узла *v* на *dcap* (т.е. уменьшение текущей пропускной

способности через это ребро на $-dcap$). Текущий поток через обратное ребро изменяется на $-dcap$.

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.

Path find_path() const выполняет поиск пути в графе. Рёбра хранятся в очереди с приоритетами, для их выбора в нужном порядке определён вспомогательный класс *ComparedEdgeRef*.

Path recover_path(const std::vector<edge_id_t> &revs) const эта функция восстанавливает путь в графе. Используется, как вспомогательная функция в *find_path*.

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 $O(|V| + |E|)$.

данными

```

11
a
x
a b 5
add edge: ab/ba max capacity: 5
a c 5
add edge: ac/ca max capacity: 5
a d 5
add edge: ad/da max capacity: 5
a z 10
add edge: az/za max capacity: 10
b c 5
add edge: bc/cb max capacity: 5
c d 8
add edge: cd/dc max capacity: 8
d z 15
add edge: dz/zd max capacity: 15
z x 50
add edge: zx/xz max capacity: 50
f y 12
add edge: fy/yf max capacity: 12
c f 12
add edge: cf/fc max capacity: 12
y z 11
add edge: yz/zy max capacity: 11
looking at edge: a -> b
looking at edge: a -> c
looking at edge: a -> d
looking at edge: a -> c
looking at edge: c -> d
looking at edge: d -> f
looking at edge: d -> y
looking at edge: d -> z
looking at edge: d -> x
modify (by 5) edges: ab (new:5 remaining capacity:0), ba (new:0 remaining capacity:5)
modify (by 5) edges: bc (new:5 remaining capacity:0), cb (new:0 remaining capacity:5)
modify (by 5) edges: cf (new:5 remaining capacity:7), fc (new:7 remaining capacity:5)
modify (by 5) edges: fy (new:5 remaining capacity:7), yf (new:7 remaining capacity:5)
modify (by 5) edges: yz (new:5 remaining capacity:6), zy (new:6 remaining capacity:5)
modify (by 5) edges: zx (new:5 remaining capacity:45), xz (new:45 remaining capacity:5)
found path: abcfyzx, flux: 5
current total: 5

```

```

looking at edge: a -> c
looking at edge: c -> b
looking at edge: a -> d
looking at edge: c -> d
looking at edge: d -> f
looking at edge: d -> y
looking at edge: d -> z
looking at edge: d -> x
modify (by 5) edges: ac (new:5 remaining capacity:0), ca (new:0 remaining capacity:5)
modify (by 5) edges: cf (new:10 remaining capacity:2), fc (new:2 remaining capacity:10)
modify (by 5) edges: fy (new:10 remaining capacity:2), yf (new:2 remaining capacity:10)
modify (by 5) edges: yz (new:10 remaining capacity:1), zy (new:1 remaining capacity:10)
modify (by 5) edges: zx (new:10 remaining capacity:40), xz (new:40 remaining capacity:10)
found path: acfyzx, flux: 5
current total: 10
looking at edge: a -> d
looking at edge: a -> z
looking at edge: z -> y
looking at edge: y -> x
modify (by 5) edges: ad (new:5 remaining capacity:0), da (new:0 remaining capacity:5)
modify (by 5) edges: dz (new:5 remaining capacity:10), zd (new:10 remaining capacity:5)
modify (by 5) edges: zx (new:15 remaining capacity:35), xz (new:35 remaining capacity:15)
found path: adzx, flux: 5
current total: 15
looking at edge: a -> z
looking at edge: z -> y
looking at edge: y -> x
modify (by 10) edges: az (new:10 remaining capacity:0), za (new:0 remaining capacity:10)
modify (by 10) edges: zx (new:25 remaining capacity:25), xz (new:25 remaining capacity:25)
found path: azx, flux: 10
current total: 25
25
a b 5
a c 5
a d 5
a z 10
b c 5
c d 0
c f 10
d z 5
f y 10
y z 10
z x 25

```

Тестирование без промежуточных результатов


```
6
k
k
k c 10
c d 10
c b 1
b c 1
k b 10
b d 10
0
b c 0
b d 0
c b 0
c d 0
k b 0
k c 0
```

```
10
a f
a b 16
a c 13
c b 4
b c 10
b d 12
c e 14
d c 9
d f 20
e d 7
e f 4
23
a b 16
a c 7
b c 4
b d 12
c b 0
c e 11
d c 0
d f 19
e d 7
e f 4
```

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

```
8
a h
a b 1
a v 2
a c 23
c g 5
g h 10
c h 5
b h 4
v h 2
13
a b 1
a c 10
a v 2
b h 1
c g 5
c h 5
g h 5
v h 2
```

Выводы.

В ходе лабораторной работы была изучена работа алгоритма поиска максимального потока в сети - метод Форда-Фалкерсона, способы хранения графа и остаточной сети и сложности по времени и памяти.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД.

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

#include <limits.h>

#define ENABLE_DEBUG

using edge_id_t = int;

struct Edge {
    int dest;           // 'vertex_char's, not indices
    int max_capacity, current_flux;
    edge_id_t rev;      // id of the corresponding dest's edge

    int remaining_capacity() const
    {
        return std::abs(max_capacity) - current_flux;
    }

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

class Vertex {
    std::vector<Edge> edges;

public:
    Edge& edge(edge_id_t e)
    {
        return edges[e];
    }
    const Edge& edge(edge_id_t e) const
    {
        return edges[e];
    }

    edge_id_t add_edge()
    {
        edges.push_back(Edge{});
        return edges.size() - 1;
    }

    edge_id_t begin_id() const
    {
        return 0;
    }
    edge_id_t end_id() const
    {
        return edges.size();
    }
};
```

```

};

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

std::ostream& write_vertex(std::ostream& os, int v);

struct Graph {
    std::vector<Vertex> vertexes;
    int start, end;          // 'vertex_char's

    // We don't assume any particular range of characters. Instead, we
    // use characters we read as identifiers but only keep vertexes in
    // the character range actually used. 'base_char' is the lowest
    // character used as a vertex identifier so far.
    int base_char = -1;

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

    Vertex& vertex(int v)
    {
        return vertexes[v - base_char];
    }
    const Vertex& vertex(int v) const
    {
        return vertexes[v - base_char];
    }

    // vertex_char: external id of a vertex
    int vertex_char(int idx) const
    {
        return idx + base_char;
    }
    // vertex_index: zero-based index of vertex
    int vertex_index(int v) const
    {
        return v - base_char;
    }

    int vertexes_count() const
    {
        return vertexes.size();
    }

    Edge& edge(int v, edge_id_t e)
    {
        return vertex(v).edge(e);
    }
    const Edge& edge(int v, edge_id_t e) const
    {
        return vertex(v).edge(e);
    }
    Edge& revedge(int v, edge_id_t e)
    {
        Edge& ed = edge(v, e);
        return edge(ed.dest, ed.rev);
    }
    const Edge& revedge(int v, edge_id_t e) const
    {

```

```

        const Edge& ed = edge(v, e);
        return edge(ed.dest, ed.rev);
    }

    void add_vertex(int v);

    void add_edge(int v1, int v2, int cap);
    void mod_edge(int v, edge_id_t e, int dcap);

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

    int get_max_flux();

    Path recover_path(const std::vector<edge_id_t>& revs) const;
    Path find_path() const;
};

void Graph::add_vertex(int v)
{
    if (base_char < 0) {
        // First vertex
        base_char = v;
    }
    else if (v < base_char) {
        // Vertex below lowest char seen before: prepend vertexes
        vertexes.insert(vertexes.begin(), base_char - v, Vertex{});
        base_char = v;
    }
    else {
        int idx = vertex_index(v),
            cur = vertexes_count();

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

void Graph::add_edge(int v1, int v2, int cap)
{
    edge_id_t
        e1 = vertex(v1).add_edge(),
        e2 = vertex(v2).add_edge();

    // Add 3 interconnected (via dest and rev) edges at once.
    edge(v1, e1) = Edge{ v2, cap, 0, e2 };
    edge(v2, e2) = Edge{ v1, -cap, cap, e1 };

#ifdef ENABLE_DEBUG
    write_vertex(write_vertex(std::cerr << "add edge: ", v1), v2);
    write_vertex(write_vertex(std::cerr << '/', v2), v1);
    std::cerr << " max capacity: " << cap << std::endl;
#endif
}

void Graph::mod_edge(int v, edge_id_t e, int dcap)
{
    Edge

```

```

        & e1 = edge(v, e),
        & e2 = revedge(v, e);

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

#ifdef ENABLE_DEBUG
    int d = e1.dest;
    std::cerr << "modify (by " << dcap << ") edges: ";
    write_vertex(write_vertex(std::cerr, v), d)
        << " (new:" << e1.current_flux << " remaining capacity:"
        << e1.remaining_capacity() << ")", ";
    write_vertex(write_vertex(std::cerr, d), v)
        << " (new:" << e2.current_flux << " remaining capacity:"
        << e2.remaining_capacity() << ")" << std::endl;
#endif
}

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

    // Traverse path, find min remaining capacity
    int v = start;
    auto iter = p.begin();
    for (; v != end; v = edge(v, *iter++).dest)
        f = std::min(f, edge(v, *iter).remaining_capacity());

    return f;
}

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

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

    return os;
}

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

    for (;;) {
        Path p = find_path();
        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 Graph::recover_path(const std::vector<edge_id_t>& revs) const
{
    Path p{};

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

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

    return p;
}

Path Graph::find_path() const
{
    // Specifies a particular edge in the graph. Comparison (operator<)
    // is defined to make the best edge the largest.
    struct ComparedEdgeRef {
        // required to calculate distance between dest and source
        int src;

        // if we have an edge ptr, we don't need a ref to the graph
        const Edge* edge;

        int dst() const
        {
            return edge->dest;
        }
        edge_id_t rev() const
        {
            return edge->rev;
        }
    };

    bool operator<(const ComparedEdgeRef& o) const
    {
        int dst1 = dst(),
            dst2 = o.dst();
        int diff1 = std::abs(dst1 - src),

```

```

        diff2 = std::abs(dst2 - o.src);

        // Note: using '>' (greater) here to keep the edge with the
        // smallest distance or destination vertex id the largest
        // one.
        if (diff1 != diff2)
            return diff1 > diff2;

        // here as well
        return dst1 > dst2;
    }
};

// We'll always have the best edge on top
std::priority_queue<ComparedEdgeRef> q{};

std::vector<bool> visited(vertexes_count(), false);

// rev[vertex_index(v)]: which edge of vertex v we should use to
// return to the vertex from which we came to 'v' the first time
std::vector<int> rev(vertexes_count(), -1);

int v = start;
while (v != end) {
    visited[vertex_index(v)] = true;

    const Vertex& vv = vertex(v);
    for (edge_id_t ei = vv.begin_id(); ei != vv.end_id(); ++ei) {
        const Edge& e = vv.edge(ei);

        if (!visited[vertex_index(e.dest)]
            && e.remaining_capacity() > 0) {

            q.push(ComparedEdgeRef{ v, &e });

        }
    }

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

    // Pick the next edge
    // v <- next vertex
    // r <- reverse edge
    edge_id_t r;
    do {
        const ComparedEdgeRef& er = q.top();

        r = er.rev();
        v = er.dst();

        q.pop();

    } while (v != end);

#ifdef ENABLE_DEBUG
    write_vertex(std::cerr << " looking at edge: ", er.src);
    write_vertex(std::cerr << " -> ", v) << std::endl;
#endif
}

```



```

    } while (visited[vertex_index(v)]);

    rev[vertex_index(v)] = r;
}

return recover_path(rev);
}

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

    g.add_vertex(c);

    return c;
}

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

    int count;
    is >> count;

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

    for (int i = 0; i < count; i++) {
        int v1 = read_vertex(is, g),
            v2 = read_vertex(is, g);

        int max_capacity;
        is >> max_capacity;

        g.add_edge(v1, v2, max_capacity);
    }

    return g;
}

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

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

```

```

}

// Write edges sorted by destination vertex character
std::ostream& write_vertex_edges(std::ostream& os, const Graph& g, int v)
{
    struct EdgeRef {
        int dest;
        edge_id_t ei;

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

    const Vertex& vv = g.vertex(v);
    for (edge_id_t ei = vv.begin_id(); ei != vv.end_id(); ++ei) {
        const Edge& edge = vv.edge(ei);
        if (edge.is_real())
            ers.push_back(EdgeRef{ edge.dest, ei });
    }

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

    for (const EdgeRef& er : ers)
        write_edge(os, g, v, er.ei);

    return os;
}

std::ostream& write_flux(std::ostream& os, const Graph& g)
{
    // Note: we iterate over vertex indices, so we use g.vertex_char
    int c = g.vertexes_count();
    for (int i = 0; i < c; i++)
        write_vertex_edges(os, g, g.vertex_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;
}

```