

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

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

Студент гр. 9382

Субботин М. О.

Преподаватель

Фирсов М. А.

Санкт-Петербург

2021

Цель работы.

Познакомиться с одним из часто используемых на практике алгоритмом, поиска потоков в сети. Получить навыки решения задач на этот алгоритм.

Задание.

Вар. 1. Поиск в ширину. Поочерёдная обработка вершин текущего фронта, перебор вершин в алфавитном порядке.

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

$v_i \quad v_j \quad w_{ij}$ - ребро графа

$v_i \quad v_j \quad w_{ij}$ - ребро графа

...

Выходные данные:

P_{max} — величина максимального потока

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

$v_i \quad v_j \quad w_{ij}$ - 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
```

Описание алгоритма.

На каждой итерации алгоритма Форда-Фалкерсона происходит поиск пути от начальной вершины до конечной, в данной реализации это поиск в ширину. Путь есть, если через ребра этого пути можно пустить поток. После нахождения пути, определяется максимально возможный поток, который можно проложить на этом пути, т.е. минимальная вместимость по всем ребрам в выбранном пути. Затем идет пересчёт емкостей как в прямом пути, так и в обратном. Емкости для пути в обратном порядке нужны затем, чтобы направить поток в противоположном направлении к изначальному направлению ребра. Затем общий поток инкрементируется на величину текущего. Алгоритм заканчивает работу, когда не останется свободных путей от начальной к конечной вершине.

Сложность алгоритма.

В худшем случае алгоритм Форда-Фалкерсона будет проходить F раз, где F – величина максимального потока в сети, т.к. в худшем случае на каждой итерации алгоритма поток будет увеличиваться на 1 (если емкость ребер – целые числа). Поиск в ширину занимает время $O(V + E)$, где V – количество вершин, E – количество ребёр в “остаточном” графе. Можно упростить как $O(E)$. Так что сложность по времени работы алгоритма будет $O(E * F)$.

Сложность алгоритма по памяти $O(V + E)$, где V – количество вершин, E – количество ребер в “остаточном” графе, т.к. алгоритм хранит лишь граф и путь от начальной вершины до конечной.

Описание функций и структур данных.

```
class FordFulkerson {
public:
    bool bfs();
    void doFulkerson();
    std::vector<std::pair<char, int>> getAdjacentVertices(char u);
    void readGraph();

private:
    char s;
    char t;
    std::map<char, char> parent;
    std::map<char, std::map<char, int>> graph;
    std::map<char, std::map<char, int>> residualGraph;
} – структура данных для работы алгоритма поиска максимального пути в графе.
```

`std::map<char, std::map<char, int>> graph` – структура данных для хранения графа. Ключ – вершина графа, значение – `map` в которой ключ – смежная вершина, значение – емкость ребра. Ребро направлено от вершины с первого ключа к вершине во втором ключе.

`std::map<char, std::map<char, int>> residualGraph` – структура данных для хранения “остаточного” графа, в котором вычисляются потоки. Структура представления этого графа аналогична предыдущему.

`std::map<char, char> parent` – структура данных для представления пути в графе. Ключом служит текущая вершина, а значение – вершина от которой мы пришли к текущей вершине.

`std::vector<std::pair<char, int>> getAdjacentVertices(char u)` – функция, возвращающая смежные вершины к `u` и емкости ребер, образованные между возвращаемыми вершинами и вершины `u`.

Аргументы:

`char u` – вершина, для которой ищутся смежные вершины

Возвращаемое значение:

`std::vector<std::pair<char, int>>` - вектор пар смежных вершин и емкостей ребер.

`bool bfs()` – функция поиска пути в графе

Возвращаемое значение:

`bool` – есть свободный путь в графе или нет.

`void doFulkerson()` – функция, выполняющая основной алгоритм поиска максимального потока в графе.

Тестирование.

№	Входные данные	Выходные данные	Результат
1	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	Правильно
2	4 a d a c 1 a b 1 c b 1 b c 1	0 a b 0 a c 0 b c 0 c b 0	Правильно
3	11 a	4 a b 1	Правильно

	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	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	
4	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 12 a c 11 b c 0 b d 12 c b 0 c e 11 d c 0 d f 19 e d 7 e f 4	Правильно
5	5 a d a b 1000	2000 a b 1000 a c 1000 b c 0	Правильно

	a c 1000 b c 1 b d 1000 c d 1000	b d 1000 c d 1000	
6	14 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2 b a 7 c a 6 d b 6 f c 9 e d 3 f d 4 c e 2	12 a b 6 a c 6 b a 0 b d 6 c a 0 c e 0 c f 8 d b 0 d e 2 d f 4 e c 2 e d 0 f c 0 f d 0	Правильно

Выводы.

Был исследован часто используемый на практике алгоритм - поиск максимального потока в графе. Также были получены навыки решения задач на этот алгоритм.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <limits>
#include <map>
#include <queue>
#include <vector>

#define DEBUG

class FordFulkerson {
public:
    bool bfs();
    void doFulkerson();
    std::vector<std::pair<char, int>> getAdjacentVertices(char u);
    void readGraph();

private:
    char s;
    char t;
    std::map<char, char> parent;
    std::map<char, std::map<char, int>> graph;
    std::map<char, std::map<char, int>> residualGraph;
};

std::vector<std::pair<char, int>> FordFulkerson::getAdjacentVertices(char u) {
    std::vector<std::pair<char, int>> edges;
    for (auto edge : residualGraph[u]) {
        edges.emplace_back(std::make_pair(edge.first, edge.second));
    }
    return edges;
}

bool FordFulkerson::bfs() {
#ifdef DEBUG
    std::cout << std::endl
                << "Ищем путь от s к t с помощью алгоритма поиска в ширину: " <<
std::endl;
#endif
}
```



```

std::queue<char> verticesToExplore;
verticesToExplore.push(s);
std::map<char, bool> visited;
visited[s] = true;

//пока есть вершины, которые хотим посетить
while (!verticesToExplore.empty()) {
    char u = verticesToExplore.front();
    verticesToExplore.pop();
    std::vector<std::pair<char, int>> edges = getAdjacentVertices(u);
#ifdef DEBUG
        std::cout << "Достаем вершину " << u << " из очереди и проходимся по ее
соседам: " << std::endl;
        for (auto edge : edges) {
            std::cout << edge.first << " ";
        }
        std::cout << std::endl;
#endif
        //смотрим смежные вершины с текущей
        for (auto edge : edges) {
#ifdef DEBUG
            std::cout << "Соседняя вершина " << edge.first << " "
                << "ребро с ней имеет capacity "
                << edge.second << ", ";
            visited[edge.first] ? std::cout << "вершина посещалась ранее" :
std::cout << "вершина не посещалась ранее";
            std::cout << std::endl;
#endif

            //если capacity > 0 и вершина еще не посещалась
            if (edge.second > 0 && !visited[edge.first]) {
                //записываем ее в очередь, чтобы потом пройти
                verticesToExplore.push(edge.first);
                //записываем в map для восстановления пути
                parent[edge.first] = u;
                //помечаем посещенной
                visited[edge.first] = true;
#ifdef DEBUG
                    std::cout << "Записываем вершину в очередь и в путь" << std::endl;
#endif
            }
        }
    }
}

```

```

        //если же достигли конечной вершины -- путь найден, заканчиваем
bfs
        if (edge.first == t) {
#ifdef DEBUG
            std::cout << "Достигли конечной вершины, путь найден,
завершаем поиск." << std::endl
                << std::endl;
#endif
            return true;
        }
    }
    return false;
}

void FordFulkerson::doFulkerson() {
    char u, v;
    residualGraph = graph;

    int max_flow = 0;

    //пока есть путь от s к t
    while (bfs()) {
        int path_flow = std::numeric_limits<int>::max();

#ifdef DEBUG
        std::cout << "Ищем минимальный capacity среди ребер из пути: " <<
std::endl;
#endif
        //проходимся по найденному пути и ищем ребро с минимальным capacity
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
#ifdef DEBUG
            std::cout << '(' << u << ',' << v << ',' << residualGraph[u][v] <<
')' << ' ';
#endif
            path_flow = std::min(path_flow, residualGraph[u][v]);
        }
#ifdef DEBUG
    
```

```

std::cout << std::endl
    << "Минимальный capacity: " << path_flow << std::endl;
std::cout << "Проходимся по пути, изменяя текущие capacity ребер: " <<
std::endl;
#endif

//проходимся по пути снова и отнимаем от каждого capacity минимальный
//также для ребер в обратном направлении увеличиваем значение на тот же
минимальный capacity
for (v = t; v != s; v = parent[v]) {
    u = parent[v];
    residualGraph[u][v] -= path_flow;
    residualGraph[v][u] += path_flow;
}

#ifdef DEBUG

for (v = t; v != s; v = parent[v]) {
    u = parent[v];
    std::cout << '(' << u << ',' << v << ',' << residualGraph[u][v] <<
')' << ' ';
}
std::cout << std::endl;

std::cout << "И также меняем значение заполненности ребер для пути в
обратном направлении: " << std::endl;
for (v = t; v != s; v = parent[v]) {
    u = parent[v];
    std::cout << '(' << v << ',' << u << ',' << residualGraph[v][u] <<
')' << ' ';
}
std::cout << std::endl;

#endif

//инкрементируем общий поток
max_flow += path_flow;
#ifdef DEBUG
std::cout << "Текущий общий поток: " << max_flow << std::endl;
#endif
}

```

```

#ifdef DEBUG
    std::cout << std::endl
        << "Доступных путей больше нет, завершаем алгоритм." << std::endl
        << std::endl;
#endif

    std::cout << max_flow << std::endl;
    for (auto const &vertex : graph) {
        for (auto const neighbor : graph[vertex.first]) {
            int flow = (neighbor.second -
residualGraph[vertex.first][neighbor.first] < 0) ? 0 : neighbor.second -
residualGraph[vertex.first][neighbor.first];
            std::cout << vertex.first << " " << neighbor.first << " " << flow <<
std::endl;
        }
    }
}

void FordFulkerson::readGraph() {
    int N;
    std::cin >> N;
    std::cin >> s >> t;
    for (int i = 0; i < N; i++) {
        char u, v;
        int capacity;
        std::cin >> u >> v >> capacity;
        graph[u][v] = capacity;
    }
}

int main() {
    FordFulkerson ford;
    ford.readGraph();
    ford.doFulkerson();
    return 0;
}

```