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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Потоки в сети

Студентка гр. 8382 _____ Рочева А.К.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Разработать программу, находящую максимальный поток к сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона. Проанализировать алгоритм.

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 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).

Индивидуализация.

Вариант 4.

Поиск в глубину. Итеративная реализация.

Описание алгоритма.

Для хранения остаточной сети создается вектор из структур Vertex. Сеть заполняется при считывании данных, т.е изначально остаточная сеть совпадает с исходной сетью, все потоки обнуляются.

Затем вызывается функция *findMaxFlow()*, в которой происходит поиск максимального потока. Для этого в остаточной сети поиском в глубину (в функции *findPath()*) находится путь из истока в сток (если путь не будет найден, то в пути будет только исток). Пусть представляется вектором из символов (*std::vector<char> path*).

Далее остаточная сеть модифицируется в функции *modifyNet()* с учетом найденного пути. Сначала на этом пути находится ребро с минимальной пропускной способностью (в функции *minPathCapacity()*). Для этого каждый раз просматриваются соседи вершины пути. Если пропускная способность нового ребра оказывается меньше минимальной (ранее вычисленной), то она становится новой минимальной пропускной способностью. Затем тоже самое выполняется для следующей вершины пути. После того, как будет просмотрена последняя вершина, функция возвращает минимальное значение пропускной способности. В итоге для всех ребер на найденном пути вычисляется новая пропускная способность - для каждого ребра поток увеличивается на минимальное значение пропускной способности, а для противоположного ему ребра поток уменьшается на это же значение.

Модификация остаточной сети происходит каждый раз после нового найденного пути.

После последней модификации происходит подсчет значения максимального потока. Для этого складываются значения потоков через соседей истока.

Описание структур данных.

1. Структура *Vertex* представляет вершину графа. В поле *name* хранится название вершины, в поле *visited* – булевое значение о том, была ли посещена вершина (для поиска в глубину), в *parent* – имя предыдущей вершины Информация о соседних вершинах, а так же о пропускной способности и потоке инцидентных к этим вершинам ребер хранится в векторе *std::vector<std::pair<char, std::pair<int, int>>> neighbors*. Он состоит из пар значений <имя соседней вершины, <пропускная спобоность инцидентного к ней ребра, поток через инцидентное ребро>>.
2. *vector <Vertex> residualNet* – контейнер vector из stl, содержащий структуры Vertex для представления остаточной сети. Сеть представлена как массив (вектор) смежности.
3. *vector <char> path* – контейнер vector из stl для представления найденного пути. Хранит названия посещенных вершин от начальной до конечной.
4. *vector <int> resultFLows* – контейнер vector из stl для хранения значений потоков через ребра, инцидентные истоку и его соседям. Нужен для ясности, на работу алгоритма не влияет.

Описание функций.

1. *void findMaxFlow(std::vector<Vertex> & residualNet, char sourceVertex, char sinkVertex)* – функция поиска максимального потока в сети. Аргументы: std*::vector<Vertex> & residualNet* – ссылка на остаточную

сеть, *char sourceVertex, char sinkVertex* – названия вершины-истока и вершины-стока.

2. *bool cmpNet(Vertex const &a, Vertex const &b)* – функция-компаратор для сортировки сети (по именам вершин в алфавитном порядке). Аргументы: *Vertex const &a, Vertex const &b* – константные ссылки на вершины сети. Функция возвращает *true*, если имя вершины *a* меньше имени вершины *b* (значит буква имени вершины *a* стоит в алфавите раньше). Иначе возвращает *false.*
3. *bool cmpVert(std::pair<char, std::pair<int, int>> const &a, std::pair<char, std::pair<int, int>> const &b)* – функция-компаратор для сортировки соседей вершины (по именам вершин в алфавитном порядке) . Аргументы: *std::pair<char, std::pair<int, int>> const &a, std::pair<char, std::pair<int, int>> const &b* – константные ссылки на соседние вершины (пары “имя соседа - <величина пропускной способности инцидентного ему ребра, величина потока этого же ребра>”). Функция возвращает *true*, если имя вершины *a* меньше имени вершины *b* (значит буква имени вершины *a* стоит в алфавите раньше). Иначе возвращает *false.*
4. *void modifyNet(std::vector<char> &path, std::vector<Vertex>& residualNet)* – функция модификации остаточной сети. В ней пересчитываются потоки через ребра сети. Аргументы: *std::vector<char> &path* – ссылка на вектор, содержащий путь от истока к стоку, *std::vector<Vertex>& residualNet* – ссылка на остаточную сеть.
5. *int minPathCapacity(std::vector<char> path, std::vector<Vertex> &residualNet)* – функция поиска на найденном пути ребра с минимальной пропускной способностью. Аргументы: *std::vector<char> path* – вектор, содержащий путь, *std::vector<Vertex>& residualNet* – ссылка на остаточную сеть. Функция возвращает значение минимальной пропускной способности.

6. *std::vector<char> findPath(std::vector<Vertex> net, char startVertex, char finishVertex)* – функция поиска пути от истока к стоку итеративным поиском в глубину. Аргументы: *std::vector<Vertex> net* – сеть, в которой ведется поиск пути, *char startVertex, char finishVertex* – стартовая и конечная вершины (исток и сток). Функция возвращает найденный путь (если путь не был найден, то возвращает вектор из одного значения – имени стартовой вершины).
7. *char returnNextVertex(Vertex &vertex, std::vector<Vertex> &net)* – функция выбора следующей вершины для просмотра сети поиском в глубину. Аргументы: *Vertex &vertex* – ссылка на вершину, из соседей которой будет выбираться следующая, *std::vector<Vertex> &net* – ссылка на сеть, в которой ведется поиск в глубину. Функция возвращает имя выбранной вершины (или символ '\0', если вершина не была найдена).
8. *Vertex *returnVertex(char name, std::vector<Vertex> &net)* – функция возврата указателя на вершину графа по заданному имени. Аргументы: *char name* – имя вершины, указатель на которую нужно вернуть, *std::vector<Vertex> &net* – ссылка на сеть, вершину которой нужно получить. Возвращает указатель на найденную вершину или *nullptr*, если вершина не была найдена.

Сложность алгоритма.

Сложность по операциям: поиск пути происходит за O(|E|+|V|) (при выборе следующей вершины мы просматриваем соседей текущей. Всего для всех вершин таких просмотров будет не более |E|. Вершин, в которые мы входим, не более |V|). На каждом шаге находится новый путь, всего будет не более f таких шагов, где f — максимальная пропускная способность графа (каждый раз алгоритм увеличивает поток по крайней мере на единицу, т. к. пропускные способности всех ребер — целые числа). Поэтому общая сложность алгоритма — O((|E|+|V|)*f).

Сложность по памяти: O(|V|+|E|) (храним только граф, а в пути хранятся не структуры Vertex, а только их имена.).

Тестирование

(без вывода промежуточных данных):

Входные данные	Выходные данные
9 a d a b 8 b c 10 c d 10 h c 10 e f 8 g h 11 b e 8 a g 10 f d 8	18 a b 8 a g 10 b c 0 b e 8 c d 10 e f 8 f d 8 g h 10 h c 10
5 a d a e 5 a b 6 e c 5 b c 6 c d 10	10 a b 6 a e 4 b c 6 c d 10 e c 4
7 a f a f 20 a b 10 b a 5 b h 8 h f 5 h g 4 g f 3	28 a b 8 a f 20 b a 0 b h 8 g f 3 h f 5 h g 3

(с выводом промежуточных данных):

Входные данные	Выходные данные
8 a g a b 5 a c 8	Graph: a: b 5 0 c 8 0 b: d 3 0

b d 3 c e 6 c f 2 d g 2 e g 5 f g 2	c: e 6 0 f 2 0 d: g 2 0 e: g 5 0 f: g 2 0 g: Update residual net using path [acfg] capacity (a, c) = 8 now it is minimum capacity. capacity (c, f) = 2 now it is minimum capacity. Minimum capacity for path = 2 Update residual net using path [aceg] capacity (a, c) = 6 now it is minimum capacity. capacity (e, g) = 5 now it is minimum capacity. Minimum capacity for path = 5 Update residual net using path [abdg] capacity (a, b) = 5 now it is minimum capacity. capacity (b, d) = 3 now it is minimum capacity. capacity (d, g) = 2 now it is minimum capacity. Minimum capacity for path = 2 max flow = 0 + max flow (b) + max flow (c) = 0 + 2 + 7 = 9 RESULT 9 a b 2 a c 7 b d 2 c e 5 c f 2 d g 2 e g 5 f g 2

Выводы.

В ходе выполнения работы была разработана программа, выполняющая поиск максимального потока в сети (код в приложении А).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ФАЙЛА MAIN.CPP

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

struct Vertex {
    std::vector<std::pair<char, std::pair<int, int>>> neighbors; //
вершины, в которые можно попасть из данной:
                                                                  // <имя
вершины, <С, поток>>
    char name;
    bool visited;
    char parent;
};

// возвращает вершину сети по заданному имени
Vertex *returnVertex(char name, std::vector<Vertex> &net){
    for (auto & vertex : net)
        if (vertex.name == name)
            return &vertex;
    return nullptr;
}

// возвращает следующую вершину, которую нужно рассматривать в поиске в
глубину
char returnNextVertex(Vertex &vertex, std::vector<Vertex> &net){
    char nextVertex = '\0';
    for (auto & vert : vertex.neighbors){
        // если вершина еще не посещена и емкость ребра больше нуля
        if(!(returnVertex(vert.first, net)->visited) && vert.second.first
> 0)
            nextVertex = vert.first;
    }
    return nextVertex;
}

// ищем путь поиском в глубину
std::vector<char> findPath(std::vector<Vertex> net, char startVertex, char
finishVertex){
    char currentVertex = startVertex;
    std::vector<char> path;
    path.push_back(startVertex);

    char nextVertex = returnNextVertex(*returnVertex(currentVertex, net),
net);
```

```
    if (nextVertex == '\0')
        return path; // собственно возвратится стартовая вершина, т.к.
сеть состоит только из нее

    // итеративный поиск в глубину. Закончится, когда дойдем до конца или
если текущая вершина равна стартовой
                                                // и при этом переход в
следующую вершину не может быть осуществлен
    while(currentVertex != finishVertex && !(currentVertex == startVertex
&& nextVertex == '\0')){

        returnVertex(currentVertex, net)->visited = true;
        if(nextVertex != '\0') {
            returnVertex(nextVertex, net)->parent = currentVertex;
            currentVertex = nextVertex;
            path.push_back(currentVertex);
        }
        else { // нет свободной вершины, в которую можно перейти
            path.pop_back();
            currentVertex = returnVertex(currentVertex, net)->parent;
        }

        if ((returnVertex(currentVertex, net)->neighbors).empty()) // у
новой текущей вершины нет соседей
            continue;

        nextVertex = returnNextVertex(*returnVertex(currentVertex, net),
net);
    }

    return path;
}

// ищем ребро с минимальной пропускной способностью в пути и возвращаем
это значение
int minPathCapacity(std::vector<char> path, std::vector<Vertex>
&residualNet){
    int minCapacity = -1;
    while(path.size() > 1){

        // рассматриваем соседей вершины, которая сейчас находится в
начале пути
        for (auto & vertex : (returnVertex(path[0], residualNet)-
>neighbors))
            // если нашли соседа, который является второй вершиной в пути,
            // и при этом емкость через ребро к этой вершине меньше
известной минимальной, то меняем мин. емкость
```

```
            if (vertex.first == path[1] && (minCapacity == -1 ||
vertex.second.first < minCapacity)) {
                std::cout << "capacity (" << path[0] << ", " << path[1] <<
") = " << vertex.second.first << std::endl;
                std::cout << "now it is minimum capacity." << std::endl;
                minCapacity = vertex.second.first;
            }
        path.erase(path.begin()); // удаляем текущую верхнюю вершину пути
    }
    return minCapacity;
}

// модифицируем остаточную сеть,
void modifyNet(std::vector<char> &path, std::vector<Vertex>& residualNet){
    std::cout << "Update residual net using path [";
    for (auto & vert : path)
        std::cout << vert;
    std::cout << "]" << std::endl;
    int minCapacity = minPathCapacity(path, residualNet);
    std::cout << "Minimum capacity for path = " << minCapacity <<
std::endl;

    while(path.size() > 1){

        for (auto & vertex1 : returnVertex(path[0], residualNet)-
>neighbors){

            if (vertex1.first == path[1]) { // если сосед является
следующей вершиной пути
                vertex1.second.second += minCapacity; // добавляем поток
соседу
                vertex1.second.first -= vertex1.second.second; // и
уменьшаем его пропускную способность

                // пробегаем по соседам соседа в поисках обратного ребра
                for (auto & vertex2 : returnVertex(path[1], residualNet)-
>neighbors){
                    if (vertex2.first == path[0]){ // если нашли ребро к
родительской вершине
                        vertex2.second.second -= minCapacity; // то
уменьшаем поток между ними
                        vertex2.second.first -= vertex2.second.second; //
и емкость
                    }
                }
            }
        }
        path.erase(path.begin());
```

```
    }
}

// для соседей вершины
bool cmpVert(std::pair<char, std::pair<int, int>> const &a,
std::pair<char, std::pair<int, int>> const &b) {
    return a.first < b.first;
}

// для графа
bool cmpNet(Vertex const &a, Vertex const &b){
    return a.name < b.name;
}

void findMaxFlow(std::vector<Vertex> & residualNet, char sourceVertex,
char sinkVertex){

    std::vector<char> path = findPath(residualNet, sourceVertex,
sinkVertex);

    while (path.size() != 1) {
        modifyNet(path, residualNet);
        path = findPath(residualNet, sourceVertex, sinkVertex);
    }

    int maxFlow = 0;

    std::vector<int> resultFlows;
    // макимальный поток = сумма потоков соседей истока
    std::cout << "max flow = 0";
    for (auto & vertex : returnVertex(sourceVertex, residualNet)-
>neighbors){
        std::cout << " + max flow (" << vertex.first << ")";
        resultFlows.push_back(vertex.second.second);
        maxFlow += vertex.second.second;
    }
    std::cout << " = 0";
    for (auto & flow : resultFlows)
        std::cout << " + " << flow;
    std::cout << " = " << maxFlow << std::endl;

    std::cout << "RESULT" << std::endl;
    std::cout << maxFlow << std::endl;
    std::sort(residualNet.begin(), residualNet.end(), cmpNet);
    for(auto & vertex : residualNet) {
        std::sort(vertex.neighbors.begin(), vertex.neighbors.end(),
cmpVert);
        for(auto neighbor : vertex.neighbors)
```

```
            std::cout << vertex.name << " " << neighbor.first << " " <<
std::max(0, neighbor.second.second) << std::endl;
    }
}

int main(){
    std::vector<Vertex> residualNet; // остаточная сеть
    int n;
    char sourceVertex, sinkVertex; // исток и сток
    std::cin >> n;
    std::cin >> sourceVertex;
    std::cin >> sinkVertex;
    char from, to;
    int capacity;
    for(int i = 0; i < n; i++) {
        std::cin >> from >> to >> capacity;
        Vertex *vertex = returnVertex(from, residualNet);
        if (vertex == nullptr){
            vertex = new Vertex;
            vertex->name = from;
            vertex->visited = false;
            vertex->neighbors.push_back({to, {capacity, 0}});
            residualNet.push_back(*vertex);
        } else {
            vertex->neighbors.push_back({to, {capacity, 0}}); // 0 -
обнуляем все потоки сначала
        }
        if (!returnVertex(to, residualNet)){
            auto vertex1 = new Vertex;
            vertex1->name = to;
            vertex1->visited = false;
            residualNet.push_back(*vertex1);
        }
    }

    std::cout << "Graph:" << std::endl;
    for (auto & edge : residualNet){
        std::cout << edge.name << ":";
        for (auto & ver : edge.neighbors)
            std::cout << "\t" << ver.first << " " << ver.second.first << "
" << ver.second.second << std::endl;
        std::cout << std::endl;
    }

    findMaxFlow(residualNet, sourceVertex, sinkVertex);

    return 0;
}
```