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: Максимальный поток

Студент гр. 9382 _____ Юрьев С.Ю.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить принципы поиска максимального потока в сети, а также фактической величины потока, протекающей через каждое ребро, реализовать соответствующую программу.

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.

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
vn - сток
vi *vj* *ωij* - ребро графа
vi *vj* *ωij* - ребро графа
...

Выходные данные:
Pmax - величина максимального потока
vi *vj* *ωij* - ребро графа с фактической величиной протекающего потока
vi *vj* *ωij* - 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
```

Теоретические сведения.

Сеть – ориентированный взвешенный граф, имеющий один исток и один сток.
Исток – вершина, из которой рёбра только выходят*.
Сток – вершина, в которую рёбра только входят*.
Поток – абстрактное понятие, показывающее движение по графу.
Величина потока – числовая характеристика движения по графу (сколько всего выходит из истока = сколько всего входит в сток).
Пропускная способность – свойство ребра, показывающее, какая максимальная величина потока может пройти через это ребро.
Максимальный поток (максимальная величина потока) – максимальная величина, которая может быть выпущена из истока, которая может пройти через все рёбра графа, не вызывая переполнения ни в одном ребре.
Фактическая величина потока в ребре – значение, показывающее, сколько величины потока проходит через это ребро.

Описание алгоритма.

Пока в сети можно найти путь, происходит поиск пути, подсчет его пропускной способности, пересчет остаточных пропускных способностей. Когда не останется путей из истока в сток, поиск прекращается, считается максимальный поток в графе, и выводится результат. Максимальный поток считается по значениям дуг, исходящих из истока.

При построении пути выбор ребра осуществляется по принципу, указанному в индивидуализации:
выбирается ребро, соединяющее вершины, имена которых находятся ближе всего друг к другу, в случае, когда таких ребер несколько, выбирается то, имя конца которого ближе к началу алфавита. Вершины, в которые попадали дуги при поиске пути, записываются, потом при выборе нового ребра просматриваются все смежные вершины для каждой из них.

При расчете пропускной способности пути, рассматриваются все ребра, участвующие в пути, и выбирается наименьшая длина. Это значение и будет пропускной способностью рассматриваемого пути.

При пересчете пропускных способностей, рассматриваются все ребра, участвующие в пути. Из длины каждого ребра вычитается пропускная способность пути, а к длине обратного ребра добавляется это же значение.

Описание функций и структур данных.

class Point — класс для хранения информации о врешине.

Содержит поля:

m_name — имя вершины

m_isVisited — флаг, который поднимается, если при поиске пути было

выбрано ребро с этой вершиной

m_cameFrom — указывает ребро, по которому добрались в эту вершину

m_neighbours — карта для хранения смежных вершин, где ключ — имя вершины, а значение — величина потока.

bool cmp(Point *a, Point *b) — компаратор для сортировки вершин в лексикографическом порядке.

Принимает на вход указатели на сравниваемые вершины;

Возвращает true или false, в зависимости от результата сравнения вершин.

void doReadingAndInitialization(int numberOfEdges, std::vector<Point *> &graph, std::vector<std::pair<char, char>> &edges) — делает ввод и инициализацию списка ребер и списка вершин.

Принимает на вход:

int numberOfEdges — число ребер

std::vector<Point *> &graph — список вершин

std::vector<std::pair<char, char>> &edges — список введенных ребер графа

Ничего не возвращает.

bool isEdgeInEdgesList(std::vector<std::pair<char, char>> edges, char ver1, char ver2) - проверяет, было ли дано такое ребро дано изначально.

Принимает на вход:

std::vector<std::pair<char, char>> edges — список введенных ребер графа

char ver1 — имя вершины (начало ребра)

char ver2 — имя вершины (конец ребра)

Возвращает true или false, в зависимости от того, есть такое ребро, или нет.

void writeAnswer(std::vector<Point *> graph, char from, std::vector<std::pair<char, char>> &edges) — подсчитывает и выводит конечный ответ в нужной форме.

Принимает на вход:

std::vector<Point *> graph -

char from -

std::vector<std::pair<char, char>> &edges -

Ничего не возвращает.

std::pair<Point *, Point *> choosePoint(std::vector<Point *> graph, std::vector<char> vertices) — выбирает следующее ребро.

Принимает на вход:

std::vector<Point *> graph — список всех вершин

std::vector<char> vertices — вершины, которые уже были посещены при поиске текущего пути.

Возвращает пару вершин на концах выбранного ребра.

Point* findPointInGraph(char name, std::vector<Point *> graph) - находит

вершину в графе, если она есть.

Принимает на вход:

char name — имя искомой вершины

std::vector<Point *> graph — список вершин

Возвращает указатель на вершину, если она существует, и nullptr в противном случае.

bool findFlow(std::vector<Point *> &graph, char start, char end) - находит очередной путь в графе.

Принимает на вход:

std::vector<Point *> &graph — список вершин

char start — имя источника

char end — имя стока

Возвращает true или false в зависимости от того, был ли найден какой-нибудь путь из источника в сток.

int findMinWayCapacity(char end, std::vector<Point *> graph) - находит пропускную способность указанного пути.

Принимает на вход:

char end — имя стока

std::vector<Point *> graph — список вершин

Возвращает значение пропускной способности найденного ранее пути.

void recountResidualCapacities(int min, char end, std::vector<Point *> &graph) - делает пересчет остаточных пропускных способностей.

Принимает на вход:

int min — пропускная способность найденного пути

char end — имя стока

std::vector<Point *> &graph — список вершин

Ничего не возвращает.

void clearUnwantedMarks(std::vector<Point *> &graph) — делает очистку меток, проставленных во время поиска пути

Принимает на вход:

std::vector<Point *> &graph — список вершин.

Ничего не возвращает.

void findMaxFlow(std::vector<Point *> &graph, char start, char end) — находит максимальный поток в сети и фактическую величину потока, протекающего через каждое ребро.

Принимает на вход:

std::vector<Point *> &graph — список вершин

char start — имя источника

char end — имя стока

Ничего не возвращает.

void freeMemory(std::vector<Point *> &graph) — освобождает выделенную под хранение вершин память.

Принимает на вход:

std::vector<Point *> &graph -

Ничего не возвращает.

void doTheTask() - вызывает другие функции в нужном порядке для выполнения поставленной задачи.

Ничего не принимает на вход.

Ничего не возвращает.

Оценка сложности.

Обозначения: V - количество вершин, F - максимальный поток, E - количество ребер.

По памяти O(V+E), так как хранится информация как о вершинах, так и о ребрах.

По времени O(F*V), так как максимум нужно будет искать путь F раз и рассматривать все ребра.

Тестирование.

Ввод	Вывод

5 a d a b 2 a c 4 b c 5 a d 3 c d 1	4 a b 1 a c 0 a d 3 b c 1 c d 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

7 a e a b 5 a d 4 b c 5 d c 7 c d 7 d e 4 c e 8	9 a b 5 a d 4 b c 5 c d 0 c e 5 d c 0 d e 4
3 a b a b 2 a c 3 c b 1	3 a b 2 a c 1 c b 1
10 b f a c 5 d e 7 b g 7 g f 4 c b 5 a d 3 d b 4	14 a c 2 a d 0 b f 8 b g 6 c b 0 c f 2 d b 0 d e 0 g a 2

g a 6 b f 8 c f 2	g f 4

Выводы.

В ходе выполнения работы была написана программа, реализующая поиск максимального потока в сети и вычисление фактического потока, протекающего через каждое ребро.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

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

#define COMMENTS
#define PATH

class Point        // класс вершины
{
public:
   char m_name;           // имя вершины
   bool m_isVisited;      // флаг "посещения" (для поиска пути)
   std::pair<int, Point *> m_cameFrom;              // указывает, как мы добрались до этой
вершины
   std::map<char, std::pair<int, int>> m_neighbours;   // хранит информацию о смежных
вершинах

   Point(): m_isVisited{false}, m_cameFrom{0, nullptr}
   {}
};


// компаратор для сортировки вершин в графе
bool cmp(Point *a, Point *b);

// ввод и инициализация списка ребер и списка вершин
void doReadingAndInitialization(int numberOfEdges, std::vector<Point *> &graph,
std::vector<std::pair<char, char>> &edges);

// проверяет, было ли дано такое ребро
bool isEdgeInEdgesList(std::vector<std::pair<char, char>> edges, char ver1, char ver2);

// выводит ответ
void writeAnswer(std::vector<Point *> graph, char from, std::vector<std::pair<char, char>>
&edges);

// выбирает очередную вершину
std::pair<Point *, Point *> choosePoint(std::vector<Point *> graph, std::vector<char> vertices);

// находит вершину в графе, если она есть (в ином случае вернет nullptr)
Point* findPointInGraph(char name, std::vector<Point *> graph);

// находит очередной путь в графе
bool findFlow(std::vector<Point *> &graph, char start, char end);
```

```
// находит пропускную способность данного пути
int findMinWayCapacity(char end, std::vector<Point *> graph);

// делает пересчет остаточных пропускных способностей
void recountResidualCapacities(int min, char end, std::vector<Point *> &graph);

// очистка меток, проставленных во время поиска пути (откат значений полей вершин)
void clearUnwantedMarks(std::vector<Point *> &graph);

// функция поиска максимального потока
void findMaxFlow(std::vector<Point *> &graph, char start, char end);

// очищает выделенную под список вершин память
void freeMemory(std::vector<Point *> &graph);

// функция-менеджер (чтобы main() был пустым)
void doTheTask();




int main()
{
   doTheTask();

   return 0;
}

bool cmp(Point *a, Point *b)
{
   return a->m_name < b->m_name;
}

void doReadingAndInitialization(int numberOfEdges, std::vector<Point *> &graph,
std::vector<std::pair<char, char>> &edges)
{
   char from, to;
   int len;
   Point *ver;

   for (int i = 0; i < numberOfEdges; i++)
   {
      std::cin >> from >> to >> len;  // считали ребро
      edges.push_back({from, to});    // и добавили его в список ребер

      ver = findPointInGraph(from, graph);

      if (ver != nullptr) // если вершина уже есть в графе
      {
```

```
            ver->m_neighbours[to] = {len, 0}; // добавим смежную вершину к списку смежных
        }
        else    // иначе создадим и добавим вершину в список
        {
            ver = new Point;
            ver->m_name = from;
            ver->m_neighbours[to] = {len, 0};
            graph.push_back(ver);
        }

        if (!isEdgeInEdgesList(edges, to, from)) // если обратное ребро не было уже задано
        {
            ver = findPointInGraph(to, graph);      // снова ищем вершину в графе, но уже для
обратного ребра

            if (ver == nullptr) // если вершины еще нет, то добавим ее
            {
                ver = new Point;
                ver->m_name = to;
                ver->m_neighbours[from] = {0, 0};
                graph.push_back(ver);
            }
            else    // если же вершина уже есть в графе
            {
                ver->m_neighbours[from] = {0, 0}; // добавим смежную вершину к списку
смежных
            }
        }
    }
}

bool isEdgeInEdgesList(std::vector<std::pair<char, char>> edges, char ver1, char ver2)
{
    for (auto i : edges)
    {
        if (i.first == ver1 && i.second == ver2)
        {
            return 1;
        }
    }
    return 0;
}

void writeAnswer(std::vector<Point *> graph, char from, std::vector<std::pair<char, char>>
&edges)
{
    auto start = findPointInGraph(from, graph);
    int max = 0;

    for (auto i : start->m_neighbours)
```

```
    {
        max += i.second.second;
    }
    std::cout << max << '\n';

    for (auto ver : graph)
    {
        for (auto neib : ver->m_neighbours)
        {
            if (isEdgeInEdgesList(edges, ver->m_name, neib.first))
            {
                if (neib.second.second > 0)
                {
                    std::cout << ver->m_name << " " << neib.first << " " << neib.second.second << '\n';
                }
                else
                {
                    std::cout << ver->m_name << " " << neib.first << " 0\n";
                }
            }
        }
    }
}

std::pair<Point *, Point *> choosePoint(std::vector<Point *> graph, std::vector<char> vertices)
{
    int min = 26, check;
    Point *minV;
    Point *prev;
    Point *ver;

#ifdef COMMENTS
    std::cout << "\t\t\tВыбираем новую дугу\n";
#endif
    for (auto name : vertices)
    {
        ver = findPointInGraph(name, graph);
        for (auto neib : ver->m_neighbours)
        {
            if (findPointInGraph(neib.first, graph)->m_isVisited == 1) // если в вершину уже
"заходили", то пропускаем ее
            {
                continue;
            }

            check = abs(neib.first - name); // расстояние между именами вершин
(индивидуализация)
            if ((check < min || check == min && neib.first < minV->m_name) && neib.second.first
> 0)
            {
```

```
                prev = ver;
                min = check;
                minV = findPointInGraph(neib.first, graph);
#ifdef COMMENTS
                std::cout << "\t\t\t\tНовая дуга выбрана: [" << prev->m_name << "," << neib.first <<
"]\n";
#endif
            }
        }
    }

    if (min == 26)
    {
        return {nullptr, nullptr};
    }
    return {minV, prev};
}

Point* findPointInGraph(char name, std::vector<Point *> graph)
{
    for (auto i : graph)
    {
        if (i->m_name == name)
        {
            return i;
        }
    }
    return nullptr;
}

bool findFlow(std::vector<Point *> &graph, char start, char end)
{
    std::vector<char> vertices; // для хранения пройденных вершин
    vertices.push_back(start);  // внесли стартовую точку

    auto ver = findPointInGraph(start, graph);
    ver->m_isVisited = true;

#ifdef COMMENTS
    std::cout << "\t\tПоиск пути:\n";
#endif
    while (ver->m_name != end)
    {
#ifdef COMMENTS
        std::cout << "\t\t\tУже использованные вершины:\n\t\t\t[";
        for (auto q : vertices)
        {
            std::cout << q << " ";
        }
        std::cout << "]\n";
```

```
#endif
        auto newVers = choosePoint(graph, vertices); // выбор следующего
        auto newV = newVers.first;
        if (newV == nullptr)
        {
            return 0; // если потоков больше нет
        }
#ifdef COMMENTS
        std::cout << "\t\t\tВыбранная вершина: [" << newV->m_name << "]\n";
#endif
        auto prev = newVers.second;
        vertices.push_back(newV->m_name);
        newV->m_isVisited = 1;
        newV->m_cameFrom = {prev->m_neighbours[newV->m_name].first, prev};
        ver = newV;
    }
    return 1; // если дуга нашлась, то данные о вершинах в графе будут изменены, и можно
будет восстановить путь
}

int findMinWayCapacity(char end, std::vector<Point *> graph)
{
    auto ver = findPointInGraph(end, graph);
    int min = ver->m_cameFrom.first;

    while (ver->m_cameFrom.second != nullptr) // пока не дойдем до начала
    {
        if (min > ver->m_cameFrom.first)
        {
            min = ver->m_cameFrom.first;
        }
        ver = ver->m_cameFrom.second;
    }
    return min;
}

void recountResidualCapacities(int min, char end, std::vector<Point *> &graph)
{
    auto ver = findPointInGraph(end, graph)->m_cameFrom.second;
    char prev = end;
#ifdef PATH
    std::cout << "\t\tПропускная способность данного пути: " << min << "\n";
    std::vector<char> path;
#endif
#ifdef COMMENTS
    std::cout << "\t\tНачало пересчета пропускных способностей:\n";
#endif
    while (ver->m_cameFrom.second != nullptr) //пока не дойдем до начала
    {
#ifdef PATH
```

```
        path.push_back(ver->m_name);
#endif
#ifdef COMMENTS
        std::cout << "\t\t\tИз пропускной способности ребра [" << ver->m_name << ", " << prev
<< "] было вычтено " << min << "\n";
        std::cout << "\t\t\tК пропускной способности ребра [" << prev << ", " << ver->m_name
<< "] было прибавлено " << min << "\n\n";
#endif
        ver->m_neighbours[prev].first -= min; //вычитаем из использованного ребра
        ver->m_neighbours[prev].second += min;
        auto prevVer = findPointInGraph(prev, graph);
        prevVer->m_neighbours[ver->m_name].first += min; //добавляем к обратному ребру
        prevVer->m_neighbours[ver->m_name].second -= min;
        prev = ver->m_name;
        ver = ver->m_cameFrom.second;
    }
#ifdef COMMENTS
    std::cout << "\t\t\tИз пропускной способности ребра [" << ver->m_name << ", " << prev <<
"] было вычтено " << min << "\n";
    std::cout << "\t\tПересчет пропускных способностей окончен\n";
#endif
#ifdef PATH
    path.push_back(ver->m_name);
    std::cout << "\t\tПуть:\n\t\t";
    for (char i = path.size() - 1; i >= 0; i--)
    {
        std::cout << path[i];
    }
    std::cout << end;
#endif
    ver->m_neighbours[prev].first -= min;
    ver->m_neighbours[prev].second += min;
#ifdef COMMENTS
    std::cout << "\n\n";
#endif
}

void clearUnwantedMarks(std::vector<Point *> &graph)
{
    for (auto ver : graph)
    {
        ver->m_cameFrom = {0, nullptr};
        ver->m_isVisited = false;
    }
}

void findMaxFlow(std::vector<Point *> &graph, char start, char end)
{
    std::vector<char> vertices; // для хранения пройденных вершин
    int min;
```

```
#ifdef COMMENTS
   std::cout << "Начат поиск макс. потока\n";
#endif
   while (findFlow(graph, start, end)) // пока можем - находим пути в графе
   {
      // находим пропускную способность найденного пути
      min = findMinWayCapacity(end, graph);
#ifdef COMMENTS
      std::cout << "\tПропускная способность посчитана\n";
#endif
      // пересчитываем остаточные пропускные способности
      recountResidualCapacities(min, end, graph);
#ifdef COMMENTS
      std::cout << "\tПересчет пропускных сбособностей выполнен\n";
#endif
      // очищаем метки, поставленные во время поиска пути
      clearUnwantedMarks(graph);
#ifdef COMMENTS
      std::cout << "\tОчистка меток выполнена\n----------------------------------------\n";
#endif
   }

#ifdef COMMENTS
   std::cout << "Конец! Макс. поток найден\n";
#endif
}

void freeMemory(std::vector<Point *> &graph)
{
   for(int k = 0; k < graph.size(); k++)
   {
      delete graph.at(k);
   }
}

void doTheTask()
{
   int numberOfEdges;
   char start, end;

   std::cin >> numberOfEdges >> start >> end;              // считали кол-во ребер, исток и
сток

   std::vector<Point*> graph;
   std::vector<std::pair<char, char>> edges;

   doReadingAndInitialization(numberOfEdges, graph, edges);    // считали введенные ребра и
инициализировали список ребер и список вершин(граф)

   std::sort(graph.begin(), graph.end(), cmp);              // отсортировали список вершин
```

```
    findMaxFlow(graph, start, end);                    // нашли максимальный поток

    writeAnswer(graph, start, edges);                  // вывели ответ

    freeMemory(graph);                                 // очистили выделенную для хранения
вершин память
}
```