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: Поиск с возвратом

Студент гр. 8303 _____ Кибардин А.Б.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучение алгоритма Форда-Фалкерсона для поиска максимальной пропускной способности сети.

Формулировка задачи.

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

Индивидуализация.

Вариант 4. Поиск в глубину. Итеративная реализация.

Описание алгоритма.

Алгоритм проверяет с помощью поиска в глубину, есть ли путь через заданный граф. Если путь найден, то в данном пути ищется ребро с наименьшей пропускной способностью. При нахождении значения этого потока, всем прямым ребрам в пути прибавляется этот поток. Если в пути встречается обратное ребро, то из пропускной способности этого ребра вычитается найденная наименьшая пропускная способность. Величина найденного потока прибавляется к текущей пропускной способности всего графа. Далее путь откатывается назад до вершины, из которой выходит ребро с наименьшей пропускной способностью и запускается новый поиск в глубину. Алгоритм работает до тех пор, пока есть путь из истока в сток.

Сложности алгоритма.

Сложность алгоритма по памяти O(V+E), где V – количество вершин в графе, E – количество ребер в графе

Временная сложность алгоритма O(Ef), где – E количество ребер, f-максимальный поток в графе.

Описание структур данных.

```
class Edge
{
```

```
public:
   std::vector<char> backEdge;
   std::map<char, std::pair<int, int>> forwardEdge;
};
```

Класс Edge является контейнером ребер вершин.

backEdge – контейнер для хранения имен вершин, из которых идет поток в данную вершину.

forwardEdge – контейнер для хранения прямых ребер (названия следующей вершины и величин максимального и текущего потоков через рассматриваемое ребро)

```
class Graph
{
   char start, end
   std::map<char,  Edge> graph
   std::vector<char> way
   std::vector<char> backNeighborList;
}
```

Класс Graph описывает структуру графа и основные методы для работы с ним.

start и end – исток и сток графа

graph – контейнер описывающий структуру графа.

way – контейнер для хранения текущего пути в графе

backNeighborList – контейнер для хранения множества вершин, которые добавлялись в путь при обратном ходе

Описание методов класса Graph.

Graph() – конструктор класса, внутри которого задается исток и сток графа, а так же множество ребер и их максимальные пропускные способности.

bool dfs() – метод поиска в глубину. Возвращает true, если путь от истока в сток найден.

int searchFlow() – метод поиска максимального потока в графе с помощью алгоритма Форда-Фалкерсона. Возвращает значение максимального потока в графе

void print() – метод печати максимальной пропускной способности, ребер и потоков, проходящих через них.

void printPatch() – метод печати текущего пути в графе.

Тестирование.

```
12
a
e
a b 5
a c 4
a d 2
b c 1
b g 5
b f 3
c g 4
c f 1
d c 2
d f 1
g e 5
f e 8
-------------------
searching way
forward available ways:
a->b 5/0
backward available way
next top: b
forward available ways:
b->c 1/0
backward available way
next top: c
forward available ways:
c->f 1/0
backward available way
next top: f
forward available ways:
f->e 8/0
backward available way
next top: e
```

```
founded way: abcfe
start search bottle neck
current flow: 1
flow of a->b = 1
flow of b->c = 1
flow of c->f = 1
flow of f->e = 1
go back
pop e
pop f
pop c
-------------------
searching way
forward available ways:
b->f 3/0
backward available way
next top: f
forward available ways:
f->e 8/1
backward available way
f->c 1
next top: c
forward available ways:
c->g 4/0
backward available way
next top: g
forward available ways:
g->e 5/0
backward available way
next top: e
founded way: abfcge
start search bottle neck
bottle neck: 1
current flow: 2
```

```
flow of a->b = 2
flow of b->f = 1
flow of c->f = 0
flow of c->g = 1
flow of g->e = 1
go back
pop e
pop g
pop c
-------------------
searching way
forward available ways:
f->e 8/1
backward available way
next top: e
founded way: abfe
start search bottle neck
current flow: 4
flow of a->b = 4
flow of b->f = 3
flow of f->e = 3
go back
pop e
pop f
-------------------
searching way
forward available ways:
b->g 5/0
backward available way
next top: g
forward available ways:
g->e 5/1
backward available way
```

```
next top: c
forward available ways:
c->f 1/0
backward available way
next top: f
forward available ways:
f->e 8/3
backward available way
next top: e
founded way: abgcfe
start search bottle neck
current flow: 5
flow of a->b = 5
flow of b->g = 1
flow of c->g = 0
flow of c->f = 1
flow of f->e = 4
go back
pop e
pop f
pop c
-------------------
searching way
forward available ways:
g->e 5/1
backward available way
next top: e
founded way: abge
start search bottle neck
current flow: 5
flow of a->b = 5
flow of b->g = 1
flow of g->e = 1
go back
```

```
pop e
pop g
pop b
-------------------
searching way
forward available ways:
a->c 4/0
backward available way
next top: c
forward available ways:
c->g 4/0
backward available way
c->b 1
next top: b
forward available ways:
b->g 5/1
backward available way
next top: g
forward available ways:
g->e 5/1
backward available way
next top: e
founded way: acbge
start search bottle neck
bottle neck: 1
current flow: 6
flow of a->c = 1
flow of b->c = 0
flow of b->g = 2
flow of g->e = 2
go back
pop e
pop g
pop b
```

```
------------------
searching way
forward available ways:
c->g 4/0
backward available way
next top: g
forward available ways:
g->e 5/2
backward available way
g->b 2
next top: b
forward available ways:
backward available way
no way from: b
go back to prev top

------------------
forward available ways:
g->e 5/2
backward available way
next top: e
founded way: acge
start search bottle neck
current flow: 9
flow of a->c = 4
flow of c->g = 3
flow of g->e = 5
go back
pop e
pop g
pop c
------------------
searching way
forward available ways:
```

```
searching way
forward available ways:
a->d 2/0
backward available way
next top: d
forward available ways:
d->c 2/0
backward available way
next top: c
forward available ways:
c->g 4/3
backward available way
next top: g
forward available ways:
backward available way
g->b 2
next top: b
forward available ways:
backward available way
no way from: b
go back to prev top

--------------------
forward available ways:
backward available way
no way from: g
go back to prev top

--------------------
forward available ways:
backward available way
no way from: c
go back to prev top
```

```
------------------
forward available ways:
d->f 1/0
backward available way
next top: f
forward available ways:
f->e 8/4
backward available way
next top: e
founded way: adfe
start search bottle neck
current flow: 10
flow of a->d = 1

flow of d->f = 1
flow of f->e = 5
go back
pop e
pop f
------------------
searching way
forward available ways:
d->c 2/0
backward available way
next top: c
forward available ways:
c->g 4/3
backward available way
next top: g
forward available ways:
backward available way
g->b 2
next top: b
forward available ways:
```

```
backward available way
no way from: b
go back to prev top

-------------------
forward available ways:
backward available way
no way from: g
go back to prev top

-------------------
forward available ways:
backward available way
no way from: c
go back to prev top

-------------------
forward available ways:
backward available way
no way from: d
go back to prev top

-------------------
forward available ways:
backward available way
no way from: a
go back to prev top

-------------------
-------------------
10
a b 5
a c 4
a d 1
b c 0
b f 3
b g 2
c b 0
c f 1
c g 3
d c 0
d f 1
f c 0
f e 5
g c 0
g e 5

Process finished with exit code 0
```

Вывод.

В ходе выполнения лабораторной работы были получены знания для работы с алгоритмом Форда-Фалкерсона для поиска максимального потока в графе. Были реализованы классы ребер и графа. Так же в классе графа реализованы методы поиска в глубину и алгоритма Форда-Фалкерсона

Приложение А. Исходный код.

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

#define debug

class Edge
{
public:
   std::vector<char> backEdge;                  // -контейнер для хранения обратных путей
   std::map<char, std::pair<int, int>> forwardEdge;  // -контейнер для хранения множества прямых путей и
их пропускных способностей
};
class Graph
{
   char start, end;                             // -исток и сток графа
   std::map<char,  Edge> graph;                 // -контейнер графа
   std::vector<char> way;                       // -найденный путь
   std::vector<char> backNeighborList;          // -контейнер для хранения множества вершин, которые
добавлялись в
public:                                         // путь при обратном проходе
   Graph()
   {
      int edgeNumber;
      std::cin >> edgeNumber;
      std::cin >> start;
      std::cin >> end;
      for(int i = 0; i < edgeNumber; i++)
      {
         int distance;
         char from, where;
         std::cin >> from >> where >> distance;
         graph[from].forwardEdge[where].first = distance;
         graph[from].forwardEdge[where].second = 0;

      }
   }

   bool dfs()                                   // -метод поиска в глубину
   {
      std::vector<char> closed;
#ifdef debug
      std::cout << "--------------------" << std::endl;
      std::cout << "searching way" << std::endl;
#endif
      while (!way.empty()) {
         char currentTop = way.back();
         if(currentTop == end)                  // -если на очередной итерации был добавлен сток, то путь найден
            return true;
         bool flag = false;
         bool backNeighborFounded = false;
                                                // ищем прямые пути
#ifdef debug
         std::cout << "forward available ways:"<< std::endl;
#endif
         for (const auto &iter : graph[currentTop].forwardEdge) {
            bool inWay = false, inClosed = false;
            for (auto way_iter: way) {          // -проверка на наличии рассматриваемой вершины в пути
               if (way_iter == iter.first) {
                  inWay = true;
```

```
                break;
            }
        }
        for(auto closed_iter: closed)        // -проверка на наличии рассматриваемой вершины в множетсве
            if(closed_iter == iter.first)    // просмотренных вершин
            {
                inClosed = true;
                break;
            }
        if(inClosed || inWay)
            continue;

        if (iter.second.first - iter.second.second) {
#ifdef debug
            std::cout <<currentTop << "->" <<iter.first  << " " <<iter.second.first << "/"  << iter.second.second
<<std::endl;
#endif
            currentTop = iter.first;
            flag = true;
            break;
        }
    }

    // -рассматриваем множество обратных ребер, которые не входят в путь
#ifdef debug
    std::cout <<"backward available way"<<std::endl;
#endif
    for (auto backNeighbor: graph[way.back()].backEdge) {  // -аналогично прямым путям, проверяем
        bool inWay = false, inClosed = false;          // есть ли данное ребро в пути и в множестве
        for (auto way_iter: way) {                     // просмотренных вершин
            if (way_iter == backNeighbor) {
                inWay = true;
                break;
            }
        }
        for(auto closed_iter: closed)
            if(closed_iter == backNeighbor)
            {
                inClosed = true;
                break;
            }
        if(inClosed || inWay)
            continue;
        if (graph[backNeighbor].forwardEdge[way.back()].second) {
#ifdef debug
            std::cout <<way.back() << "->" <<backNeighbor  << " " <<
graph[backNeighbor].forwardEdge[way.back()].second <<std::endl;
#endif
            currentTop = backNeighbor;
            flag = true;
            backNeighborFounded = true;
            break;
        }
    }

    if (flag) {                                  // -Если путь найден, то заносим его в контейнер и
                                                 // Так же производим проверку, что эта найденная
                                                 // связана с предыдущей обратным ребром
#ifdef debug
        std::cout <<"next top: "  << currentTop <<std::endl;
#endif
        way.push_back(currentTop);
```

```
            if (backNeighborFounded)
            {
                bool inList = false;
                for(auto list_iter : backNeighborList)
                    if(list_iter == currentTop)
                    {
                        inList = true;
                        break;
                    }
                if(!inList)
                    backNeighborList.push_back(currentTop);
            }
        } else {
            closed.push_back(way.back());                   // -Если путь не найден, то помечаем вершину как
                                                            // просмотренную и удаляем ее из пути
#ifdef debug
            std::cout << "no way from: " << way.back() << std::endl;
            std::cout << "go back to prev top\n" << std::endl;
            std::cout << "--------------------" << std::endl;
#endif
            way.pop_back();
        }

    }
    return false;

}

int searchFlow()
{
    int currentFlow = 0;
    way.push_back(start);

    //пока есть путь до стока и ребра исходящие из истока
    //находим ребро
    //добавляем его в путь. Пока не дошли до стока, ищем новое наибольшее ребро, которое не входит в
путь.
    // Дойдя до стока, ищем наименьшее значение потока в текущем пути.
    // Откатываемся назад, до вершины, которя находится перед ребром с наименьшей пропускной
способностью
    while(dfs())
    {

#ifdef debug
        std::cout << "founded ";
        printPatch();
        std::cout << "start search bottle neck " << std::endl;
#endif
        int minFlow = INT32_MAX;
        char minTop = end;

        for(auto way_iter = way.begin(); way_iter != way.end() - 1; way_iter++) // -поиск минимального потока
        {
            bool inList = false;
            for(auto list_iter : backNeighborList)
                if(list_iter == *(way_iter+1) )
                {
                    inList = true;
                    break;
                }
            if(!inList)
            {
```

```
            for(const auto& next_iter : graph[*way_iter].forwardEdge)
            {

                if(next_iter.first == *(way_iter + 1) && minFlow > next_iter.second.first - next_iter.second.second)
                    minFlow = next_iter.second.first - next_iter.second.second;
            }
        }
        else
        {
            for(const auto& next_iter : graph[*(way_iter + 1)].forwardEdge)
            {

                if(next_iter.first == *way_iter && minFlow > next_iter.second.second)
                {
                    minFlow = next_iter.second.second;
#ifdef debug
                    std::cout << "bottle neck: " << minFlow << std::endl;
#endif
                }
            }
        }


    }
    currentFlow += minFlow;
#ifdef debug
    std::cout << "current flow: " << currentFlow << std::endl;
#endif
    bool flag = true;
    bool backTop = true;
    for(auto way_iter = way.begin(); way_iter != way.end() - 1; way_iter++) // -заполняем путь потоком
    {
        bool inList = false;
        for(auto list_iter : backNeighborList)
            if(list_iter == *(way_iter+1) )
            {
                inList = true;
                break;
            }
        if(!inList)                                     //-если путь прямой, то прибавляем поток
        {
            graph[*way_iter].forwardEdge[*(way_iter + 1)].second += minFlow;
#ifdef debug
            std::cout << "flow of " << *way_iter<< "->" << *(way_iter+1)  << " = " <<
graph[*way_iter].forwardEdge[*(way_iter + 1)].second << std::endl;
#endif
            bool hasThatNeighbor = false;
            for(auto& backNeighbor: graph[*(way_iter + 1)].backEdge)       //-проверка на наличие
            {                                           //вершины в множестве обратных путей
                if(*way_iter == backNeighbor)                       //следующей вершины
                {
                    hasThatNeighbor = true;
                    break;
                }
            }
            if(!hasThatNeighbor)
                graph[*(way_iter + 1)].backEdge.push_back(*way_iter);

            if(flag && backTop && graph[*way_iter].forwardEdge[*(way_iter + 1)].second ==
graph[*way_iter].forwardEdge[*(way_iter + 1)].first)
            {
                minTop = *way_iter;
```

```
                flag = false;
            }
        } else                                  //-из обратного ребра вычитаем
        {                                       //минимальный поток
            graph[*(way_iter + 1)].forwardEdge[*way_iter].second -= minFlow;
            minTop = *way_iter;
            backTop = false;
#ifdef debug
            std::cout << "flow of " << *(way_iter+1)<< "->" << *way_iter << " = " <<
graph[*way_iter].forwardEdge[*(way_iter + 1)].second << std::endl;
#endif
        }
    }
#ifdef debug
    std::cout << "go back" << std::endl;
#endif
    while(way.back() != minTop)                 //-откатываемся назад до вершины,
    {                                       // из которой выходит наименьший поток
#ifdef debug
        std::cout << "pop " << way.back() << std::endl;
#endif
        if(!backNeighborList.empty() && way.back() == backNeighborList.back())
            backNeighborList.pop_back();
        way.pop_back();
    }

  }
  return currentFlow;
}

void print()                // -метод печати максимального потока и ребер графа
{
  for(const auto& iter: graph) {
    for (const auto& next_iter: iter.second.forwardEdge)
      std::cout << iter.first << " " << next_iter.first << " " << next_iter.second.second << std::endl;

  }
}

void printPatch()           // -метод печати пути
{
  std::cout << "way: ";
  for(auto iter:way)
    std::cout << iter;
  std::cout << std::endl;
}

};

int main() {
  Graph graph;
  std::cout << graph.searchFlow() << std::endl;
  graph.print();
  return 0;
}
```