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

ОТЧЕТ

по практической работе №3
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Поток в сети

Студент гр. 8382 _____ Вербин К.М.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучение алгоритма Форда-Фалкерсона для поиска максимальной пропускной способности 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 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

...

Пример входных данных

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Соответствующие выходные данные

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Вариант 3

Поиск в глубину. Рекурсивная реализация.

Описание структур данных

Класс ориентированного ребра графа(class Path):

Класс ориентированного ребра графа состоит из следующих свойств:

char nameFrom – имя вершины откуда исходит ребро;

char nameOut – имя вершины куда входит ребро;

int bandwidth – пропускная способность;int flow – поток;

Класс ориентированного ребра графа содержит следующие методы:

- Path(char nameFrom, char nameOut, int bandwidth) – конструктор, принимающий на вход имя вершины куда идём, имя вершины куда идём и пропускную способность.
- void setFlow(int flow) – функция для переопределения значения потока;
- char getNameFrom() – функция, которая возвращает значение имени вершины откуда исходит ребро;
- char getNameOut() – функция, которая возвращает значение имени вершины куда входит ребро;
- int getBandwidth() – функция, которая возвращает значение пропускной способности;
- int getFlow() – функция, которая возвращает значение потока;

Функция void findMin(std::vector& local, int* maxFlow, int depth), принимает вектор, в котором хранятся ребра найденного пути, переменную, которая считает максимальный поток и переменную, которая отвечает за демонстрацию рекурсии.

Функция bool comp2(Path* a, Path* b) принимает 2 переменные класса ребер. Данная функция является компаратором для сортировки ребер по наименьшему значению разности между пропускной способностью и потоком.

Функция bool isVisitedPath(std::vector& local2, char element, int depth) принимает на вход вектор просмотренных ребер, переменную типа char и переменную, которая отвечает за демонстрацию рекурсии. Функция 7 выполняет проверку на то, что вершину, куда входит ребро, ещё не посещали

Описание алгоритма.

Решение поставленной задачи осуществляется с помощью рекурсивной функции bool findPath(std::vector& paths, std::vector& local, std::vector& local2, char myPoint, char endPoint, int depth), которая принимает на вход: исходный вектор, в котором записаны все ребра с величиной пропускаемого потока;

вектор, в который записывается найденный путь; вектор, в который записываются посещенные ребра; вершина, с которой функция будет начинать поиск; вершина, до которой функция ищет путь; и переменная, необходимая для визуального показания рекурсии алгоритма.

Функция начинает работу прежде всего с проверки, является ли текущая вершина искомой, если нет, то происходит отбор всех ребер, которые исходят из текущей вершины. Нужные ребра записываются во временный граф ребер (localPaths). Далее, если таких ребер несколько, то они сортируются по наименьшему значению разности между пропускной способности и потоком.

Следующий шаг. Начинается в цикле проход по отсортированным путям. Происходит проверка, что ребро не переполнено, т.е. значение потока меньше значения пропускной способности. Затем проверяется, посещалась ли уже вершина, куда входит текущее ребро. Если обе проверки пройдены, то это ребро записывается в вектор просмотренных ребер. После чего, рекурсивно вызывается функция поиска пути, но теперь в качестве текущей вершины будет вершина, в которую входило данное ребро.

Также рассмотрим следующий случай. Если на проверке уже отобранных и отсортированных ребер окажется так, что ребро переполнено, то цикл for продолжится, просто перейдя к следующему ребру, также выполнив все проверки. Но если среди всех доступных для вершины ребер не будет того, которое не переполнено, то функция сделает откат назад, к предыдущей рассматриваемой вершине.

Как только на очередном рекурсивном вызове будет достигнута искомая вершина(сток), то функция возвратит true и рекурсия вернется на шаг назад, затем в вектор ответа (local) будет записано ребро и функция снова вернет true. Получается, что все ребра пути будут записаны в вектор ответа, но в обратном порядке.

Затем, после того как функция поиска пути вернула true, происходит вызов функции поиска минимальной разности между величиной пропускной

способности и величиной потока. Далее найденный минимум прибавляется ко всем потокам найденного пути и также к значению максимального потока.

После выхода из функции происходит очищение вектора ответа(в котором записан текущий путь до стока) и вектора пройденных ребер. И цикл в main снова повторяется, снова запускается функция поиска путей с текущей начальной вершиной. Данный цикл завершается, когда функция поиска путей не сможет найти ни один путь до искомой вершины.

После завершения цикла, происходит сортировка всех ребер в лексикографическом порядке с помощью функции sort() и написанного к ней компаратора.

В конце main происходит вывод ответа, полностью соответствующий всем требованиям. А именно, сначала выводится найденное значение максимального потока, а уже за ним все ребра графа с фактической величиной протекающего потока.

Сложность алгоритма по памяти

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

Временная сложность алгоритма

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

Тестирование

Input	Output
8 a h a b 5 a c 4	3 a b 0 a c 2 a d 1 b g 0

a d 1 b g 1 c e 2 c f 3 d e 6 e h 4 f h 4	c e 2 c f 0 d e 1 e h 3
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	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

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

КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>      // std::setw

class Path {
private:
   char nameFrom;
   char nameOut;
   int bandwidth;//bandwidth
   int flow = 0;//flow

public:
   Path(char nameFrom, char nameOut, int bandwidth) :
nameFrom(nameFrom), nameOut(nameOut), bandwidth(bandwidth) {}

   void setFlow(int flow) {
      Path::flow = flow;
   }

   char getNameFrom() const {
      return nameFrom;
   }

   char getNameOut() const {
      return nameOut;
   }

   int getBandwidth() const {
      return bandwidth;
   }

   int getFlow() const {
      return flow;
   }


};

bool comp(Path a, Path b);
```

```
        void findMin(std::vector<Path*>& local, int* maxFlow, long int depth)
{//function for selecting the minimum difference between throughput and flow
            std::cout << std::setw(depth + 1) << ' ' << "The function of selecting the
minimum difference between bandwidth and flow, as well as changing
information, is started." << std::endl;
            int Min = local.front()->getBandwidth();   //front() Returns a reference to
the first element in the vector container
            for (Path* path : local) {
               if (Min > (path->getBandwidth() - path->getFlow())) {
                  Min = path->getBandwidth() - path->getFlow();
               }
            }
            std::cout << std::setw(depth + 1) << ' ' << "Minimum =  " << Min <<
std::endl;
            std::cout << std::setw(depth + 1) << ' ' << "Adding the found minimum to
the path threads." << std::endl;
            for (Path* path : local) {
               path->setFlow(path->getFlow() + Min);//now add the found minimum
to the stream
            }

            *maxFlow = *maxFlow + Min;//increase the value of the maximum flow
            std::cout << std::setw(depth + 1) << ' ' << "The maximum flow is now
equal to:  " << *maxFlow << std::endl;
        }

        bool comp(Path a, Path b) {
            if (a.getNameFrom() != b.getNameFrom()) {

               return a.getNameFrom() < b.getNameFrom();
            }
            else {
               return a.getNameOut() < b.getNameOut();
            }
        }

        bool comp2(Path* a, Path* b) {//the comparator is used to choose where to
go. For example, there are 2 paths, one 10/40 and the other 15/30. It is necessary to
go to the second, since 30-15 < 40-10

            return (a->getBandwidth() - a->getFlow()) < (b->getBandwidth() - b-
>getFlow());
        }
```

```
        bool isVisitedPath(std::vector<Path*>& local, char element, long int depth)
{
            std::cout << std::setw(depth + 1) << ' ' << "A function is launched to
check whether this vertex has already been visited." << std::endl;
            for (Path* path : local) {
                if (element == path->getNameFrom()) {
                    std::cout << std::setw(depth + 1) << ' ' << "The check function shuts
down. Top  " << element << "  is already in sight. This means that we will not go
along this edge." << std::endl;
                    return false;
                }
            }
            std::cout << std::setw(depth + 1) << ' ' << "The check function shuts
down. Top  " << element << "  hasn't been viewed yet." << std::endl;
            return true;
        }

        bool findPath(std::vector<Path>& paths, std::vector<Path*>& local,
std::vector<Path*>& local2, char myPoint, char* endPoint, long int depth) {
            depth++;

            std::cout << std::setw(depth + 1) << ' ' << "The path search function
starts." << std::endl;

            std::cout << std::setw(depth + 1) << ' ' << "Checking is in progress: the
vertex is not the one being searched for." << std::endl;
            if (myPoint == *endPoint) {
                std::cout << std::setw(depth + 1) << ' ' << "The vertex is the one you
are looking for. This means that the function returns true." << std::endl;
                return true;
            }

            std::vector<Path*> localPaths;

            std::cout << std::setw(depth + 1) << ' ' << "Selecting edges that originate
from the current vertex." << std::endl;
            for (auto& path : paths) {//selecting which edges come from the current
vertex
                if (path.getNameFrom() == myPoint) {
                    std::cout << std::setw(depth + 1) << ' ' << "Since the edge:  " <<
path.getNameFrom() << ' ' << path.getNameOut() << "  originates from the current
vertex  " << myPoint << ". Then we add this edge to the temporary path vector."
<< std::endl;

                    localPaths.emplace_back(&path);
```

```
            }
        }

        std::cout << std::setw(depth + 1) << ' ' << "Sorting edges by the smallest
difference between throughput and flow." << std::endl;
        std::sort(localPaths.begin(), localPaths.end(), comp2);//choosing where to
go first

        for (Path* path : localPaths) {
            std::cout << std::setw(depth + 1) << ' ' << "Checking that the edge  " <<
path->getNameFrom() << ' ' << path->getNameOut() << "  is not overfilled." <<
std::endl;
            if (path->getFlow() < path->getBandwidth()) {//checking that the path
flow is less than the bandwidth
                std::cout << std::setw(depth + 1) << ' ' << "Checking that the vertex
where the edge enters has not been visited yet." << std::endl;
                if (isVisitedPath(local2, path->getNameOut(), depth)) {//if we
haven't visited the top yet
                    std::cout << std::setw(depth + 1) << ' ' << "Writing an edge  " <<
path->getNameFrom() << ' ' << path->getNameOut() << "  in the vector of viewed
edges." << std::endl;
                    local2.emplace_back(path);
                    std::cout << std::setw(depth + 1) << ' ' << "Recursively calling the
function, but now the current vertex will be:  " << path->getNameOut() <<
std::endl;
                    if (findPath(paths, local, local2, path->getNameOut(), endPoint,
depth)) {//ðåêóðñèâíî âûçûâàåì ôóíêöèþ ñ íåïðîñìîòðåííîé âåðøèíîé
                        std::cout << std::setw(depth + 1) << ' ' << "Since the function
returned true, it means that the vertex was found. Writing an edge  " << path-
>getNameFrom() << ' ' << path->getNameOut() << "  in the response vector." <<
std::endl;
                        depth--;
                        local.emplace_back(path);
                        return true;
                    }
                    else {
                        //äåëàåì îòêàò íàçàä
                        std::cout << std::setw(depth + 1) << ' ' << "Making a rollback."
<< std::endl;
                        local2.pop_back();
                    }
                }
            }
            std::cout << std::setw(depth + 1) << ' ' << "The edge is full or there are
no more possible paths." << std::endl;
```

```
            }
            std::cout << std::setw(depth + 1) << ' ' << "The Pathfinder function shuts
down." << std::endl;

            return false;

        }

        int main() {
            setlocale(LC_ALL, "Russian");
            char startPoint, endPoint;//source and drain
            char start, end;
            int weight;
            int count;
            int flag = 0;
            int flag2 = 0;
            long int depth = 0;
            std::vector<Path*> local;

            std::vector<Path*> local2;

            std::vector<Path> paths;

            int maxFlow = 0;
            std::cout << "Hello! Please enter the number of oriented edges of the
graph, the source, drain, and edges of the graph." << std::endl;
            std::cin >> count;
            std::cin >> startPoint;
            std::cin >> endPoint;
            if (count != 0)
            {
                flag2 = 1;
                while (count != 0) {
                    std::cin >> start >> end >> weight;
                    paths.emplace_back(Path(start, end, weight));
                    count--;
                }
            }
            else
                std::cout << "Zero number of oriented edges of the graph is
introduced!" << std::endl;

            if (flag2)
            {
                std::cout << "The path search function is called." << std::endl;
```

```
            while (findPath(paths, local, local2, startPoint, &endPoint, depth)) {//as soon as we find the path to the drain, we call the function to find the minimum and change the flow
                findMin(local, &maxFlow, depth);
                std::cout << std::setw(depth + 1) << ' ' << "Clearing the response vector and the vector of viewed edges." << std::endl;
                local.clear();
                local2.clear();
            }


            std::cout << std::setw(depth + 1) << ' ' << "Sorting the output edges in lexicographic order. " << std::endl;

            std::sort(paths.begin(), paths.end(),
                [](const Path& a, const Path& b)
                {
                    if (a.getNameFrom() != b.getNameFrom()) {

                        return a.getNameFrom() < b.getNameFrom();
                    }
                    else {
                        return a.getNameOut() < b.getNameOut();
                    }

                });

            //std::sort(paths.begin(), paths.end(), comp);//sort the vertices in lexicographical order
            std::cout << std::endl;
            std::cout << "Answer:  " << std::endl;
            std::cout << "The value of maximum flow =  " << maxFlow << std::endl;

            for (Path path : paths) {
                std::cout << "Edge of the graph with the actual value of the flowing stream:   " << path.getNameFrom() << " " << path.getNameOut() << " " << path.getFlow() << std::endl;
            }

            std::cout << "Goodbye!" << std::endl;

        }

        return 0;
```

}