Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №9 по курсу «Дискретный анализ»

Студент:	И. А. Мариничев
Преподаватель:	Н. С. Капралов
Группа:	М8О-208Б
Дата:	22.05.21
Оценка:	
Подпись:	

Москва, 2021

Лабораторная работа №9. Поиск максимального потока алгоритмом Форда-Фалкерсона

Задача: Разработать программу на языке C или C++, реализующую указанный алгоритм согласно заданию:

Задан взвешенный ориентированный граф, состоящий из **n** вершин и **m** ребер. Вершины пронумерованы целыми числами от **1** до **n**. Необходимо найти величину максимального потока в графе при помощи алгоритма Форда-Фалкерсона. Для достижения приемлемой производительности в алгоритме рекомендуется использовать поиск в ширину, а не в глубину. Истоком является вершина с номером **1**, стоком – вершина с номером **n**. Вес ребра равен его пропускной способности. Граф не содержит петель и кратных ребер.

Формат входных данных

В первой строке заданы $1 \le n \le 2000$ и $1 \le m \le 10000$. В следующих **m** строках записаны ребра. Каждая строка содержит три числа – номера вершин, соединенных ребром, и вес данного ребра. Вес ребра – целое число от 0 до 10^9.

Формат результата

Необходимо вывести одно число – искомую величину максимального потока. Если пути из истока в сток не существует, данная величина равна нулю.

1 Описание

Согласно [1], транспортная сеть (flow network) $G = (V, E)$ представляет собой ориентированный граф, в котором каждое ребро $(u,v) \in E$ имеет неотрицательную **пропускную способность** (capacity) $c(u,v) \geq 0$. Далее мы потребуем, чтобы в случае, если E содержит ребро (u,v), обратного ребра (v,u) не было. Если $(u,v) \notin E$, то для удобства определим $c(u,v) = 0$, а также запретим петли. В транспортной сети выделяются две вершины: **исток** (source) s и **сток** (sink) t. Для удобства предполагается, что каждая вершина лежит на неком пути от истока к стоку, т.е. для любой вершины $v \in V$ транспортная сеть содержит путь $s \leadsto v \leadsto t$. Таким образом, граф является связным и, поскольку каждая вершина, отличная от s, содержит как минимум одно входящее ребро, $|E| \geq |V|$ - 1.

Теперь мы готовы дать более формальное определение потоков. Пусть $G = (V, E)$ - транспортная сеть с функцией пропускной способности c. Пусть s является истоком, а t — стоком. Потоком (flow) в G является действительная функция $f : V$ x $V \rightarrow \mathbb{R}$, удовлетворяющая следующим двум условиям.

Ограничение пропускной способности. Для всех u, $v \in V$ должно выполняться

$$0 \leq f(u,v) \leq c(u,v).$$

Сохранение потока. Для всех $u \in V$ - $\{s, t\}$ должно выполняться

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v).$$

Когда $(u,v) \notin E$, потока из $f(u,v)$ быть не может, так что $f(u,v) = 0$. Неотрицательную величину $f(u,v)$ мы называем потоком из вершины u в вершину v. Величина (value) $|f|$ потока f определяется как

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s),$$

т.е. как суммарный поток, выходящий из истока, минус входящий в него. Обычно транспортная сеть не имеет ребер, входящих в исток, и поток в исток, задаваемый суммой $\sum_{v \in V} f(v,s)$, равен 0. При рассмотрении остаточных сетей потоки в исток станут важными. В задаче о максимальном потоке (maximum flow problem) дана некоторая транспортная сеть G с истоком s и стоком t, и необходимо найти поток максимальной величины.

Метод Форда-Фалкерсона.

Давайте определим еще одно понятие. Остаточная пропускная способность направленного ребра это пропускная способность минус поток. Отметим, что если через направленное ребро *(u,v)* есть поток, то обратное ребро имеет нулевую пропускную способность и мы можем определить его поток, как *f(v,u) = -f(u,v)*. Это также определяет остаточчную пропусную способность для всех обратных ребер. Из всех этих ребер мы можем создать остаточную сеть, это сеть с теми же вершинами и ребрами, но вместо пропускнух способностей используются остаточные.

Метод Форда-Фалкерсона работает следующим образом. Сначала мы устанавливаем поток каждого ребра равным нулю. Затем мы ищем увеличивающий путь от s до t. Увеличивающий путь - это простой путь в остаточном графе, то есть вдоль ребер, остаточная пропускная способность которых положительна. Если такой путь найден, то мы можем добавить увеличение потока по этим ребрам. Мы продолжаем искать увеличивающие пути и увеличивать поток. Когда больше не существует увеличивающего пути, поток становится максимальным.

Уточним подробнее, что означает увеличение потока по увеличивающему пути. Пусть C - наименьшая остаточная пропускная способность ребер пути. Затем мы увеличиваем поток следующим образом: обновляем *f(u, v) += C* и *f(v, u) -= C* для каждого ребра *(u, v)* в пути.

Следует отметить, что метод Форда-Фалкерсона не определяет метод нахождения увеличивающего пути. Возможные подходы - использование DFS или BFS, которые работают за O(E). Если все пропускные способности сети являются целыми числами, то для каждого увеличивающего пути поток сети увеличивается как минимум на 1. Следовательно, сложность всего алгоритма равна O(E * F), где F - максимальный поток сети. В случае рациональных пропускных способностей алгоритм также завершится, но сложность не ограничена. В случае иррациональных пропускных способностей алгоритм может никогда не завершиться и даже не сойтись к максимальному потоку.

Алгоритм Эдмондса-Карпа.

Алгоритм Эдмондса-Карпа - это просто реализация метода Форда-Фалкерсона, которая использует BFS для поиска увеличивающих путей. Алгоритм был впервые опубликован Ефимом Диницем в 1970 году, а затем независимо опубликован Джеком Эдмондсом и Ричардом Карпом в 1972 году.

Сложность может быть задана независимо от максимального потока. Алгоритм выполняется за время O $(V * E^2)$ даже для иррациональных пропускных способностей. Идея заключается в том, что каждый раз, когда мы находим увеличивающийся путь, одно из ребер становится насыщенным, и расстояние от ребра до s будет больше, если оно появится позже снова в увеличивающем пути. А длина простых путей ограничена V.

2 Исходный код

Теперь поговорим о реализации данного алгоритма на языке C++. Ребро представим в виде стркутуры **TEdge**, которая будет хранить пару из двух соединенных вершин и пропускную способность. Роль графа остаточной сети будет выполнять **vector<vector<int>> residualGraph(n + 1)**, в котором в ячейке (u, v) будем хранить остаточные пропускные способности, которые изначально равны пропускным способностям заданной сети. Также будем использовать список смежности **vector<vector<int>> adjacencyList(n + 1)** для того, чтобы упростить поиск увеличивающего пути. Кроме того, для обновления остаточных пропускных способностей будем сохранять текущие пути в вектор **vector<int> parent(t + 1)**. Также в программе приведена стандартная реалзация BFS с использованием **std:queue**.

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

using namespace std;

const int INF = 1e9 + 1;

struct TEdge {
    pair<int, int> vertices;
    int capacity;
};

// Breadth-first search
int BFS(vector<vector<int>> &residualNetwork, vector<int> &parent, vector<vector<int>>
     &adj, int s, int t) {
    // filling an array to remeber the path
  // "-1" indicates unvisited vertices
  // "-2" indicates source
  fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;

  // creates a queue of pairs (vertex, minimum residual capacity) for BFS
    queue<pair<int, int>> q;
    q.push({s, INF});

    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

    // for every vertex in Adjacency List for current vertex
        for (int next : adj[cur]) {
```

```cpp
            // if vertex is unvisited and still has capacity
            if (parent[next] == -1 && residualNetwork[cur][next]) {
                parent[next] = cur; // add vertex next to the path
                int minFlow = min(flow, residualNetwork[cur][next]);
                if (next == t)
                    return minFlow;
                q.push({next, minFlow});
            }
        }
    }

    return 0;
}

// Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method that uses BFS for finding augmenting paths
// s - source, t - sink
long long EdmondsKarp(vector<vector<int>> &residualNetwork, vector<vector<int>> &adj, int s, int t) {
    long long maxFlow = 0;
    vector<int> parent(t + 1); // array filled by BFS that stores path (for every element we store its parent)
    int flow;

    // while we are able to find augmenting paths
    // otherwise BFS will return 0 thus breaking loop
    while ((flow = BFS(residualNetwork, parent, adj, s, t))) {
        maxFlow += flow;

        // go from sink to source
        int cur = t;
        while (cur != s) {
            int prev = parent[cur]; // using path established by BFS
            residualNetwork[prev][cur] -= flow; // subtract flow capacity of an edge along the path
            residualNetwork[cur][prev] += flow; // add flow capacity along the reversed edge
            cur = prev;
        }
    }

    return maxFlow;
}

int main() {
    int n; // number of vertices
    int m; // number of lines
    cin >> n >> m;

```

```
    // creates Residual Capacity matrix nxn filled with zeros
    vector<vector<int>> residualGraph(n + 1);
    for (int i = 0; i < n + 1; ++i) {
        residualGraph[i].resize(n + 1);
    }

  // creates Adjacency List filled with zeros
  vector<vector<int>> adjacencyList(n + 1);
    for (int i = 0; i < n + 1; ++i) {
        adjacencyList[i].resize(n + 1);
    }

    TEdge edge;
    for (int i = 0; i < m; ++i) {
        cin >> edge.vertices.first >> edge.vertices.second >> edge.capacity;

    // fills Residual Capacity matrix with elements
      // where [u][v] element stores residual capacity of an edge from vertex u to
          vertex v
        residualGraph[edge.vertices.first][edge.vertices.second] = edge.capacity;

    // fills Adjacency List with elements
      // where every element has a list of connected vertices, assuming the graph is
          undirected
    adjacencyList[edge.vertices.first][edge.vertices.second] = edge.vertices.second;
    adjacencyList[edge.vertices.second][edge.vertices.first] = edge.vertices.first;
    }

  // writes maximal flow or 0 if it doesn't exist
  cout << EdmondsKarp(residualGraph, adjacencyList, 1, n) << endl;

    return 0;
}
```

3 Тест производительности

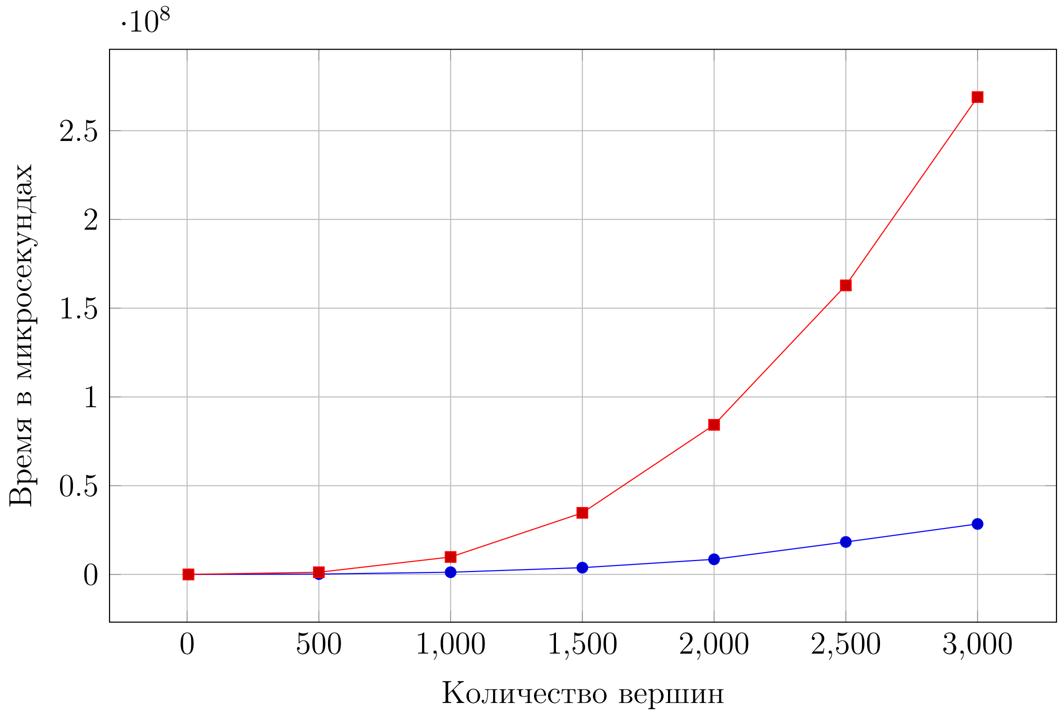
Теперь прейдем к оценке скорости и посмотрим на то, как же ведет себя программа на практке. Для этого сравним две реализации алгоритма Форда-Фалкерсона: при помощи DFS и при помощи BFS (алг. Эдмондса-Карпа). Проверим время работы программы на тестах с разным количеством вершин: 5, 500, 1000, 1500, 2000, 2500, 3000. Время выводится микросекундах. Для измерения времени использовалась бибилиотека **chrono**.

```
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_bfs
<../tests/test_5.t
22
Time: 222 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_dfs
<../tests/test_5.t
22
Time: 230 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_bfs
<../tests/test_500.t
4266
Time: 200896 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_dfs
<../tests/test_500.t
4266
Time: 1224881 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_bfs
<../tests/test_1000.t
8994
Time: 1235897 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_dfs
<../tests/test_1000.t
8994
Time: 9821751 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_bfs
<../tests/test_1500.t
13500
Time: 3811981 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_dfs
<../tests/test_1500.t
13500
Time: 34684747 microsec
```

```
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_bfs
<../tests/test_2000.t
17710
Time: 8518587 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_dfs
<../tests/test_2000.t
17710
Time: 84270917 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_bfs
<../tests/test_2500.t
22443
Time: 18292003 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_dfs
<../tests/test_2500.t
22443
Time: 162840459 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_bfs
<../tests/test_3000.t
26403
Time: 28432127 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab9/benchmark$ ./solution_dfs
<../tests/test_3000.t
26403
Time: 268933071 microsec
```

Представим полученные результаты в более наглядной форме, а именно в виде графика.

Синий: Алгоритм Форда-Фалкерсона, реализация с помощью BFS.
Красный: Алгоритм Форда-Фалкерсона, реализация с помощью DFS.



Как мы видим поиск в глубину значительно проигрывает по времени поиску в ширину. Дело в том, что поиск в ширину всегда находит кратчайший путь, на каждом шаге приближаясь к конечной точке, а поиск в глубину обходит вершины графа в случайном порядке и находит не самый оптимальный путь, совершая много лишних операций, т.к. в первую очередь он предназанчен для поиска лексикографически первого пути.
Однако асимитотическая сложность этих алгоритмов одинакова: O(E + V), где E - количество ребер графа, V - количество вершин графа.

Тесты создавались с помощью программы на языке Python:

```
from random import randint

NUMBER = [5, 500, 1000, 1500, 2000, 2500, 3000]

length_of_list = len(NUMBER)

for i in range(length_of_list):
    start = 1
    end = NUMBER[i]
    from_vertex = 1
    to_vertex = 1
```

```
    max_weight = 20

    with open(f'test_{NUMBER[i]}.t', "w+") as file:
        file.write(f'{end} {end * end}\n')
        for i in range(1, end * end):
            from_vertex = randint(start, end)
            to_vertex = randint(start, end)
            w = randint(1, max_weight)
            while from_vertex >= to_vertex:
                from_vertex = randint(start, end)
                to_vertex = randint(start, end)
            file.write(f'{from_vertex} {to_vertex} {w}\n')
```

4 Выводы

Выполнив девятую лабораторную работу по курсу «Дискретный анализ», я изучил метод Форда-Фалкерсона для решения задачи поиска максимального потока в транспортной сети. Реализовал данный алгоритм на языке C++ и сравнил две реализации этого метода с разным поиском увеличивающего пути в графе: при помощи DFS и при помощи BFS. Задача о максимальном потоке является одним из классических типов задач, к которому можно сводить похожие задачи, приеменяя к задаче метод, продемонстрованный в данной лабораторной работе.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Maximum flow - Ford-Fulkerson and Edmonds-Karp*
URL: `https://cp-algorithms.com/graph/edmonds_karp.html` (дата обращения: 23.05.2021).

[3] *Ford-Fulkerson Algorithm for Maximum Flow Problem*
URL: `https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/` (дата обращения: 22.05.2021).

[4] *Алгоритм Форда-Фалкерсона*
URL: `https://www.youtube.com/watch?v=u9NigdVHUr0` (дата обращения: 22.05.2021).