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:	А.Д. Волков
Преподаватель:	А. А. Кухтичев
Группа:	М8О-306Б
Дата:	12.12.2024
Оценка:	
Подпись:	

Москва, 2024

Лабораторная работа №9

Задача: 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.

1 Описание

Требуется реализовать алгоритм методом Форда-Фалкерсона, для поиска максимального потока в транспортной сети. Было уточнение, что нужно реализовать этот метод используя поиск в ширину. У такого алгоритма есть название - алгоритм Эдмондса-Карпа. Он использует все те же понятния, которые описаны в методе Форда-Фалкерсона, но для поиска увеличивающих путей, он использует обход графа в ширину, что дает ему большую производительность. Давайте определим метод Форда-Фалкерсона.

Метод Форда-Фалкерсона - метод решения задачи о максимальном потоке. Он называется методом, а не алгоритмом, потому что он допускает несколько реализаций с различным временем выполнения. Метод Форда-Фалкерсона базируется на трех важных идеях, а именоо остаточные сети, увеличивающие пути и разрезы. Остаточная сеть - это сеть, состоящая из ребер с пропускными способностями, указывающими, как могут меняться потоки через ребра G. Увеличивающим путем для заданных транспортной сети $G = (V, E)$ и потока f является просто путь из s в t в остаточной сети G_f. Разрезом (S, T) транспортной сети $G = (V, E)$ называется разбиение множества вершин V на множества S и $T = V - S$, такие, что $s \in S$, а $t \in T$. Метод Форда-Фалкерсона итеративно увеличивает значение потока. Вначале поток обнуляется: $f(u, v) = 0$ для всех $u, v \in V$. На каждой итерации величина потока в G увеличивается посредством поиска "увеличивающего пути"в связанной "остаточной сети"G_f. Зная ребра увеличиваюшего пути в G_f, мы можем легко идентифицировать конкретные ребра в G, для которых можно изменить поток таким образом, что его величина увеличится. Хотя каждая итерация метода Форда-Фалкерсона увеличивает величину потока, но поток через конкретное ребро может как возрастать, так и уменьшаться; уменьшение потока через некоторые ребра может быть необходими для того, чтобы позволить алгоритму переслать большой поток от истока к стоку. Мы многократно увеличиваем поток до тех пор, пока остаточная сеть не будет иметь ни одного увеличивающего пути.[1]

2 Исходный код

Сначала мы считываем все данные, и создаем экземпляр класса TGraph, в который записываются пропускные способности ребер. В самом классе при этом уже создается остаточная сеть, в которую записываются начальные пропускные способности. Затем мы применяем функцию EdmondsKarp, которая и будет высчитывать максимальный поток. Инициализируем максимальный поток нулем. Создаем вектор parents, по которому в дальнейшем мы будем восстанавливать увеличивающий путь. Затем запускаем цикл while по функции bfs, которая и будет нам искать увеличивающие пути. В функции bfs мы создаем вектор visited, по которому будем определять, бывали мы уже в вершине, или нет и очередь current_queue, которая нужна для реализации поиска в ширину. В первый элемент вектора visited мы записываем значение true, так как на данный момент мы находимся в истоке. А также исток мы записываем в очередь, так как на данный момент мы его и рассматриваем. Затем мы запускаем цикл while по очереди current_queue, в котором мы будем прокладывать путь до стока. Берем первый элемент из очереди и в цикле for рассматриваем все его связи с другими вершинами. Если мы еще не были в рассматриваемой вершине и пропускная способность ребра, соединяющего текущию вершину и рассматриваемую больше нуля (точнее существует ли оно вообще), то мы записываем его в очередь на дальнейшее рассмотрение, записываем в вектор parent откуда мы пришли в данную вершину, и помечаем вершину посещенной. Так, если мы доходим до стока, то мы помечаем его посещенным и возвращаем true. Если же мы не доходим до стока, то значит у нас не осталось путей в сток, и мы заканчиваем алгоритм. Пока мы выполняли поиск в ширину, у нас формировался вектор parent, по которому можно восстановить найденный увеличивающий путь. Для этого достаточно пройти от конца вектора, то есть от стока, до истока, по номерам вершин, которые записаны в векторе parent. Во время этого обхода, мы смотрим на все значения пропускных способностей и выбираем минимальное. Затем мы еще раз выполняем обход по parents, уже для того, чтобы обновить все ребра. Затем мы суммируем получившееся значение потока с текущим и чистим вектор parent, для следующей итерации. Таким образом мы и находим максимальный поток для транспортной сети.

```
#include <bits/stdc++.h>

class TGraph
{
public:
    int64_t vertices_quantity;
    // Weights matrix
    std::vector<std::vector<int64_t>> weights;
    // Vectors of linked vertices
    std::vector<std::vector<int64_t>> linked_vertices;
    // Constructor
    TGraph(int64_t vertices)
```

```
    {
        this->vertices_quantity = vertices;
        this->weights.resize(vertices, std::vector<int64_t>(vertices, 0)); // Initialize with 0
        this->linked_vertices.resize(vertices);
    }
    // Method for adding edges
    void AddEdge(int64_t vertice_a, int64_t vertice_b, int64_t weight)
    {
        this->weights[vertice_a - 1][vertice_b - 1] += weight;
        this->linked_vertices[vertice_a - 1].push_back(vertice_b - 1); // Adding link
        this->linked_vertices[vertice_b - 1].push_back(vertice_a - 1); // Adding reversed link for returning part of flow
    }
};

bool bfs(TGraph& graph, std::vector<int64_t>& parent)
{
    std::vector<bool> visited(graph.vertices_quantity, false);
    std::queue<int64_t> current_queue;
    current_queue.push(0); // Starting always from 0-vertice (source)
    visited[0] = true;

    while (!current_queue.empty())
    {
        int64_t current_vertice = current_queue.front();
        current_queue.pop();

        for (int64_t i = 0; i < graph.linked_vertices[current_vertice].size(); i++)
        {
            // If did not visit and not 0 capacity
            if (!visited[graph.linked_vertices[current_vertice][i]] && graph.weights[current_vertice][graph.linked_vertices[current_vertice][i]] > 0)
            {
                current_queue.push(graph.linked_vertices[current_vertice][i]);
                parent[graph.linked_vertices[current_vertice][i]] = current_vertice;
                visited[graph.linked_vertices[current_vertice][i]] = true;
            }
        }
    }
    // Did we visit target vertice
    return visited[graph.vertices_quantity - 1];
}

int64_t EdmondsKarp(TGraph& graph)
{
    int64_t max_flow = 0;
    // Parent vertices vector, for restoring path
    std::vector<int64_t> parent(graph.vertices_quantity, -1);
```

```cpp
    while (bfs(graph, parent))
    {
        int64_t flow_path = std::numeric_limits<int64_t>::max();
        // Calculating minimum capacity in found path
        for (int64_t current_vertice = graph.vertices_quantity - 1; current_vertice != 0; current_vertice = parent[current_vertice])
        {
            int64_t prev_vertice = parent[current_vertice];
            flow_path = std::min(flow_path, graph.weights[prev_vertice][current_vertice]);
        }
        // Refresh capacities and reversed edges
        for (int64_t current_vertice = graph.vertices_quantity - 1; current_vertice != 0; current_vertice = parent[current_vertice])
        {
            int64_t prev_vertice = parent[current_vertice];
            graph.weights[prev_vertice][current_vertice] -= flow_path;
            graph.weights[current_vertice][prev_vertice] += flow_path;
        }
        max_flow += flow_path;
        // Clearing vector of parents for next iteration of bfs
        parent = std::vector<int64_t>(graph.vertices_quantity, -1);
    }
    return max_flow;
}

int main()
{
    int64_t vertices_quantity, edges_quantity;
    std::cin >> vertices_quantity >> edges_quantity;
    TGraph graph(vertices_quantity);
    for (int64_t _ = 0; _ < edges_quantity; _++)
    {
        int64_t vertice_a, vertice_b, weight;
        std::cin >> vertice_a >> vertice_b >> weight;
        graph.AddEdge(vertice_a, vertice_b, weight);
    }
    int64_t max_flow = EdmondsKarp(graph);
    std::cout << max_flow << "\n";
}
```

main.cpp	
int64_t EdmondsKarp()	Функция для подсчета максимального потока сети
int64_t bfs()	Функция для обхода графа в ширину, для нахождения увеличивающего пути

TGraph()	Стандартный конструктор класса TGraph
void AddEdge()	Метод для добавления вершин в граф
int main()	Точка входа программы

```
lass TGraph
{
public:
    int64_t vertices_quantity;
    // Weights matrix
    std::vector<std::vector<int64_t>> weights;
    // Vectors of linked vertices
    std::vector<std::vector<int64_t>> linked_vertices;
    // Constructor
    TGraph(int64_t vertices);
    // Method for adding edges
    void AddEdge(int64_t vertice_a, int64_t vertice_b, int64_t weight);
};
```

3 Консоль

```
lexasy@lexasy$ cat test.txt
5 6
1 2 4
1 3 3
1 4 1
2 5 3
3 5 3
4 5 10
lexasy@lexasy$ make
g++ main.cpp -o solution
lexasy@lexasy$ ./solution <test.txt
7
```

4 Тест производительности

Тест производительности представляет из себя сравнение времени работы нашего алгоритма и алгоритма, который ищет увеличивающий путь с помощью обхода в глубину на большом тесте.

```
lexasy@lexasy$ make
g++ main.cpp -o solution
lexasy@lexasy$ make benchmark
g++ benchmark.cpp -o benchmark
lexasy@lexasy$ ./solution <tests/10.t | grep time
time: 7970ms
lexasy@lexasy$ ./benchmark <tests/10.t | grep time
time: 29004947ms
```

Как видно, разница на лицо. Это происходит из за того, что сложность алгоритма, использующего поиск в глубину зависит от величины максимального потока, которая может быть огромной, а именно, его сложность $O(E|f^*|)$, где E - количество ребер, а f^* - величина максимального потока. А сложность нашего алгоритма зависит только от количества ребер и вершин, а именно $O(VE^2)$, где V - количество вершин.

5 Выводы

Выполнив лабораторную работу №9, я узнал как искать максимальный поток в транспортной сети разными способами. Также я вспомнил материал с курса Дискретной Математики по поводу транспортных сетей. Это может мне помочь в будущем, при решении алгоритмических задач на графы.

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.))