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: Р. С. Лисин
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б-20

Дата:
Оценка:
Подпись:

Москва, 2022

Лабораторная работа №9

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$1 \leq n \leq 2000$ и $1 \leq m \leq 10000$. 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], алгоритм Форда-Фалкерсона — решает задачу нахождения максимального потока в транспортной сети. Задача о максимальном потоке для данной сети состоит в следующем: найти максимально возможную скорость производства (и потребления) вещества, при которой его еще можно доставить от истока к стоку при данных пропускных способностях труб.

Пусть у каждого ребра есть число f - функция потока (текущий поток через данное ребро) и c - пропускная способность (максимальная величина потока, которая может быть у данного ребра). Для нахождения максимального потока необходимо найти все возможные пути из источника в сток, удовлетворяющие условиям:

- Путь может проходить как по направлению ребра, так и против него.
- Путь может проходить по направлению ребра только тогда, когда функция потока этого ребра меньше его пропускной способности.
- Путь может проходить против направления ребра только тогда, когда функция потока этого ребра не равна нулю.

Когда мы проходим по направлению ребра, мы пускаем какой-то поток по этому ребру, функция потока увеличивается. Когда мы проходим против направления ребра, мы переносим часть потока с этого ребра на какое-то другое. Благодаря переходам против направлений ребер мы можем найти **максимальное** значение потока.

Когда мы нашли путь, удовлетворяющий условиям выше, нужно обновить значения функций потока ребер. Для этого нужно найти минимальное значение среди множества $\{c - f\}$ для всех ребер, ПО направлению которых мы идем, и множества $\{c\}$ для всех ребер, ПРОТИВ направления которых мы идем. Значения функции потока всех ребер, ПО (ПРОТИВ) направлению которых проходит путь, нужно увеличить (уменьшить) на найденную величину. Итоговый поток складывается из всех таких величин изменений потока.

Алгоритм останавливается, когда в графе нет пути из источника в сток, удовлетворяющего трем условиям выше.

Поиск путей можно выполнять при помощи алгоритма BFS или DFS (или любого другого алгоритма поиска пути в графе).

Сложность алгоритма: $O(flow * m)$, где $flow$ - величина максимального потока, m - количество ребер в графе.

2 Исходный код

Граф задается списком смежности, который реализован в виде структуры *std::vector <std::unordered_map <int, int»*. Благодаря такой структуре у нас будет возможность перемещаться по графу, быстро добавлять туда новые ребра, и при этом мы занимаем минимальный объем памяти.

Значение оставшейся пропускной способности заменяет значение функции потока и пропускной способности. После нахождения минимального прироста потока на каждом шаге значение оставшейся пропускной способности уменьшается на найденную величину, но для того, чтобы можно было переходить по ребрам в обратном направлении, мы увеличиваем оставшуюся пропускную способность для противоположного ребра. Это действие эквивалентно обновлению функции потока, описанного в п.1. Алгоритм нахождения максимального потока описан в функции *MaxFlow*.

Для поиска путей используется алгоритм поиска в ширину, реализованный в функции *BFS*. Функция возвращает истинное значение, если путь из одной вершины в другую существует, и ложное в противном случае. Поиск в ширину составляет массив *parents*, где *parents[i]* - предок вершины i, который нужен для прохода по найденному пути и определения прироста потока. Функция прекращает свою работу после первого найденного пути.

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <unordered_map>
5
6  using namespace std;
7
8  bool BFS(vector<unordered_map<int,int>>& graph, int start, int end, vector<int>&
       parents) {
9      parents.clear();
10     parents.resize(graph.size(), -1);
11     vector<bool> visited(graph.size());
12     queue<int> q;
13     q.push(start);
14     visited[start] = true;
15
16     while (!q.empty()) {
17         int cur_vertex = q.front();
18         q.pop();
19         for (auto edge : graph[cur_vertex]) {
20             int new_vertex = edge.first;
21             int new_capacity = edge.second;
22             if (!visited[new_vertex] && new_capacity) {
23                 visited[new_vertex] = true;
24                 q.push(new_vertex);
25                 parents[new_vertex] = cur_vertex;
```

```
                if (new_vertex == end) {
                    return true;
                }
            }
        }
    }
    return false;
}

long long MaxFlow(vector<unordered_map<int,int>>& graph, int source, int sink) {
    long long resFlow = 0;
    vector<int> parents;
    while (BFS(graph, source, sink, parents)) {
        int flow = 1000000001;
        for (int i = sink; i != source; i = parents[i]) {
            if (i == -1) {
                throw runtime_error("Can't find a parent for not source node.");
            }
            if (graph[parents[i]][i] < flow) {
                flow = graph[parents[i]][i];
            }
        }
        resFlow += flow;

        for (int i = sink; i != source; i = parents[i]) {
            graph[parents[i]][i] -= flow;
            graph[i][parents[i]] += flow;
        }
    }
    return resFlow;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<unordered_map<int,int>> graph(n);
    for (int i = 0; i < m; ++i) {
        int from, to, capacity;
        cin >> from >> to >> capacity;
        --from;
        --to;
        graph[from][to] = capacity;
    }
    cout << MaxFlow(graph, 0, n - 1) << "\n";
}
```

3 Консоль

```
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ g++ main.cpp
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./a.out
5 6
1 2 4
1 3 3
1 4 1
2 5 3
3 5 3
4 5 10
7
```

4 Тест производительности

В программе я использовал поиск в ширину для нахождения пути. Попробуем использовать поиск в глубину и сравним результаты. Для сравнения я подготовил четыре теста с полными графами, где количество вершин равно 10, 20, 30 и 35.

```
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./bfs <test10
368639003
Time: 0.0002965 s
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./dfs <test10
368639003
Time: 0.0038175 s
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./bfs <test20
808017301
Time: 0.0005277 s
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./dfs <test20
808017301
Time: 0.0012016 s
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./bfs <test30
1252856464
Time: 0.0008176 s
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./dfs <test30
1252856464
Time: 1.36699 s
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./bfs <test35
1462071249
Time: 0.0008947 s
roma@DESKTOP-JD58QU2:~/Diskran/lab9$ ./dfs <test35
1462071249
Time: 0.120481 s
```

Иногда 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. Однако асимптотическая сложность этих алгоритмов одинакова: $O(m + n)$, где m - количество ребер графа, n - количество вершин графа.

5 Выводы

Выполнив девятую лабораторную работу по курсу «Дискретный анализ», я изучил способы представления и обработки графов на C++. Граф можно представить списком ребер, матрицей смежности или списком смежности (что я и использовал).

Изучен алгоритм Форда-Фалкерсона нахождения максимального потока. Этот алгоритм часто используется в оптимизационных задачах на практике, хотя его идея чрезмерно проста.

Асимптотическая сложность моей реализации алгоритма Форда-Фалкерсона - $O(flow* m*(m+n))$, где $flow$ - величина максимального потока, m - количество ребер в графе, n - количество вершин в графе ($(m+n)$ - поиск пути в матрице смежности, $flow*m$ - обновление потоков)

Список литературы

[1] *Алгоритм Форда-Фалкерсона.*
URL: `http://urban-sanjoo.narod.ru/ford-fulkerson.html` (дата обращения: 30.10.2022).