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: Ф. А. Иванов
Преподаватель: И. Н. Симахин
Группа: М8О-208Б-19

Дата: 14.04.2022
Оценка:
Подпись:

Москва, 2022

Лабораторная работа №9

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

Задан взвешенный ориентированный граф, состоящий из n вершин и m ребер. Вершины пронумерованы целыми числами от 1 до n. Необходимо найти длины кратчайших путей между всеми парами вершин при помощи алгоритма Джонсона. Длина пути равна сумме весов ребер на этом пути. Обратите внимание, что в данном варианте веса ребер могут быть отрицательными, поскольку алгоритм умеет с ними работать. Граф не содержит петель и кратных ребер.

Формат входных данных: В первой строке заданы 1 n 2000, 1 m 4000. В следующих m строках записаны ребра. Каждая строка содержит три числа – номера вершин, соединенных ребром, и вес данного ребра. Вес ребра – целое число от -109 до 109.

Формат результата: Если граф содержит цикл отрицательного веса, следует вывести строку "Negative cycle"(без кавычек). В противном случае следует выести матрицу из n строк и n столбцов, где j-е число в i-й строке равно длине кратчайшего пути из вершины i в вершину j. Если такого пути не существует, на соответствующей позиции должно стоять слово "inf"(без кавычек). Элементы матрицы в одной строке разделяются пробелом.

1 Описание

Требуется разработать программу на языке C или C++, реализующую указанный алгоритм согласно заданию: алгоритмом Джонсона.

Как известно, алгоритм Джонсона — позволяет найти кратчайшие пути между всеми парами вершин взвешенного ориентированного графа. Данный алгоритм работает, если в графе содержатся рёбра с положительным или отрицательным весом, но отсутствуют циклы с отрицательным весом.

Зачем нужен алгоритм Джонсона, если есть алгоритм Дейкстры? Дело в том, что алгоритм Дейкстры не работает с графом, имеющим отрицательные веса ребер. В этом случае применяется алгоритм Джонсона.

Пройдемся по-этапно в реализации алгоритма Джонсона.
1) Строится дополнительная вершина соединяется со всеми другими вершинами. При этом новые дуги изначально имеют вес, равный нулю
2) Применяется алгоритм Бельмана-Форда - вычисляются расстояния от новой вершины до остальных. Проверяется на наличие циклов с отрицательными весами.
3) Изменяем веса дуг, используя значения кратчайших путей, полученные в предыдущем пункте. Используем формулу: $D(u, v) = D(u, v) + h(u) - h(v)$.
4) Дополнительная вершина удаляется.
5) Алгоритм Дейкстры применяется ко всем вершинам.
6) Возвращаем прежний вид графа применяя формулу: $D(u, v) = D(u, v) - h(u) + h(v)$.

2 Исходный код

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

using namespace std;

const long INF = 1e18;

struct Edge
{
    int startVertex, endVertex;
    long weight;
};

struct item
{
    int id;
    long weight;
    friend bool operator < (const item & first, const item second) {
        if (first.weight != second.weight) {
            return first.weight > second.weight;
        } else {
            return first.id < second.id;
        }
    }
};

class Graph
{
    int numberOfVertices, numberOfEdges;
    vector<Edge> edges;
    vector<vector< pair<int, long> > > g;
    vector<long> potentialFunctionValues;
public:
    Graph();
    ~Graph();
    void build();
    bool fordBelman();
    void dijkstra(int startVertex);
    void johnson();
};

Graph::Graph()
{
    build();
}

```

```
48 Graph::~Graph() {}
49 void Graph::build()
50 {
51     cin >> numberOfVertices >> numberOfEdges;
52     g.resize(numberOfVertices);
53     edges.resize(numberOfEdges);
54     for (int i = 0; i < numberOfEdges; ++i) {
55         int startVertex, endVertex;
56         long weight;
57         cin >> startVertex >> endVertex >> weight;
58         edges[i] = {startVertex - 1, endVertex - 1, weight};
59         g[startVertex - 1].push_back(make_pair(endVertex - 1, weight));
60     }
61     for (int i = 0; i < numberOfVertices; ++i) {
62         edges.push_back({numberOfVertices, i, 0});
63     }
64     potentialFunctionValues.resize(numberOfVertices+1);
65     for (int i = 0; i < numberOfVertices + 1; ++i) {
66         potentialFunctionValues[i] = INF;
67     }
68 }
69 void Graph::dijkstra(int startVertex)
70 {
71     vector<long> shortestPaths(numberOfVertices, INF);
72     vector<bool> usedVertex(numberOfVertices);
73     priority_queue<item> pq;
74 
75     shortestPaths[startVertex] = 0;
76     pq.push({startVertex, 0});
77     while (!pq.empty()) {
78         item cur = pq.top();
79         pq.pop();
80         int u = cur.id;
81         if (!usedVertex[u]) {
82             for (long i = 0; i < g[u].size(); ++i) {
83                 int v = g[u][i].first;
84                 long w = g[u][i].second;
85                 if (shortestPaths[u] + w < shortestPaths[v]) {
86                     shortestPaths[v] = shortestPaths[u] + w;
87                     pq.push({v, shortestPaths[v]});
88                 }
89             }
90             usedVertex[u] = true;
91         }
92     }
93     for (int k = 0; k < numberOfVertices; ++k) {
94         if (shortestPaths[k] >= INF) {
95             cout << "inf ";
96         } else {
```

```
97              cout << shortestPaths[k] + potentialFunctionValues[k] -
                    potentialFunctionValues[startVertex] << " ";
98          }
99      }
100     cout << endl;
101 }
102 bool Graph::fordBelman()
103 {
104     potentialFunctionValues[numberOfVertices] = 0;
105     bool changed = true;
106     for (int i = 0; changed and i < numberOfVertices + 1; ++i) {
107         changed = false;
108         for (long j = 0; j < edges.size(); ++j) {
109             int from = edges[j].startVertex;
110             int to = edges[j].endVertex;
111             long weight = edges[j].weight;
112             if (potentialFunctionValues[from] + weight < potentialFunctionValues[to]) {
113                 changed = true;
114                 potentialFunctionValues[to] = potentialFunctionValues[from] + weight;
115             }
116         }
117     }
118     return changed;
119 }
120 void Graph::johnson()
121 {
122     if (fordBelman()) {
123         cout << "Negative cycle" << endl;
124     } else {
125         for (int i = 0; i < numberOfVertices; ++i) {
126             for (long j = 0; j < g[i].size(); ++j) {
127                 int currentVertex = g[i][j].first;
128                 g[i][j].second = g[i][j].second + potentialFunctionValues[i] -
                        potentialFunctionValues[currentVertex];
129             }
130         }
131         for (int i = 0; i < numberOfVertices; ++i) {
132             dijkstra(i);
133         }
134     }
135 }
```

3 Консоль

```
orion@orion-laptop:~/University/DA/Lab_09/solution$ ./solution
5 4
1 2 -1
2 3 2
1 4 -5
3 1 1
0 -1 1 -5 inf
3 0 2 -2 inf
1 0 0 -4 inf
inf inf inf 0 inf
inf inf inf inf 0

orion@orion-laptop:~/University/DA/Lab_09/solution$ ./solution
5 5
1 2 -8
2 3 -6
3 1 5
3 4 10
4 5 3
Negative cycle
```

4 Тест производительности

Для сравнения времени работы обратимся к алгоритму Флойда-Уоршелла, который можно считать наивным, поскольку сложность алгоритма $O(n^3)$.
Тест производится на рандомно генерируемом графе без отрицательных циклов.
Прведем 2 теста.

Для 100.

```
orion@orion-laptop:~/University/DA/Lab_09$ floyd.cpp <test.txt
3.53485
orion@orion-laptop:~/University/DA/Lab_09$ ./main <test.txt
0.40982
```

Для 10000.

```
orion@orion-laptop:~/University/DA/Lab_09$ floyd.cpp <test.txt
29.82074
orion@orion-laptop:~/University/DA/Lab_09$ ./main <test.txt
21.00438
```

Из тестов видно, что алгоритм Джонсона работает несколько быстрее алгоритма Флойда, т.к. он асимптотически является более быстрым алгоритмом. Сложность: $O(n^2 * log(n) + nm)$

5 Выводы

Выполнив данную лабораторную работу, я научился работать с алгоритмом Джонсона и понял как он устроен.
Основная идея алгоритма заключалась в том, что мы соединяем два других алгоритма. Идея достаточно проста. Тем не менее мне было трудно совместить два алгоритма, на которых основан алгоритм Джонсона, поскольку они используют немного разные представления графа в своей стандартной реализации.

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.))