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. О. Ильин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №1

Задача: Вариант №6

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 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 написать реализацию алгоритма Джонсона. Как сказано в [1]: «Алгоритм Джонсона (англ. Johnson's algorithm) находит кратчайшие пути между всеми парами вершин во взвешенном ориентированном графе с любыми весами ребер, но не имеющем отрицательных циклов. В этом алгоритме используется метод изменения веса».

Основная идея: перевесить каждое ребро так, чтобы не было отрицательных рёбер и при этом кратчайшие пути остались таковыми, затем запустить алгоритм Дейкстры для каждой вершины.

Для того, чтобы перевесить рёбра, создаём граф G' из изначального графа G путём добавления вершины S, из которой проведены ориентированные рёбра нулевого веса во все остальные вершины графа, запускаем алгоритм Форда-Беллмана из неё. На данном этапе проводим проверку на наличие отрицательного цикла. В результате работы алгоритма Форда-Беллмана получаем массив ψ, изменяем вес каждой дуги по формуле $w_\psi(u, v) = w(u, v) + \psi(u) - \psi(v)$.

Запускаем алгоритм Дейкстры из каждой вершины, находим расстояния между всеми парами вершин, не забываем обратно изменить вес дуг.

Сложность алгоритма Джонсона складывается из одного запуска Алгоритма Форда-Беллмана — $O(m{\cdot}n)$ и n запуском алгоритма Дейкстры (реализован на *priority_ queue*, двоичной куче) — $O(n \cdot m \cdot log(n))$. Итоговая сложность $O(n \cdot m \cdot log(n))$.

В файле *graph.h* опишем класс для работы с графами, которые содержит вышеописанные алгоритмы Форда-Беллмана, Дейкстры и Джонсона а также структуры для хранения графа в виде вектора рёбер и списка смежности.

2 Исходный код

```
1  #pragma once
2  #include <vector>
3  #include <climits>
4  #include <iostream>
5
6  namespace NGraph {
7      const long long INF = LLONG_MAX;
8
9      struct TEdge {
10         size_t from, to;
11         long long cost;
12     };
13     std::istream& operator>> (std::istream& is, TEdge& edge);
14
15     class TGraph {
16     public:
17         using TVertex = std::vector< std::pair<size_t, long long> >; // stores
               connected vertexes and dist to them
18         using TAdjacencyList = std::vector< TVertex >;
19
20         TGraph(const std::vector<TEdge>& graphEdges, const size_t vertexesCount); //
               for unit tests
21         TGraph(const size_t vertexesCount, const size_t edgesCount);
22
23         std::vector<long long> FordBellman(const size_t target); // O(m * n)
24         std::vector<long long> Dijkstra(const size_t target); // O(m * log n)
25         std::vector< std::vector<long long> > Jonshon();
26
27         friend std::istream& operator>> (std::istream& is, TGraph& graph);
28     private:
29         size_t vertexCount;
30         std::vector<TEdge> edges;
31         TAdjacencyList adjList;
32     };
33
34     /*
35      * target = [0 ... vertexCount]
36      * from/to = [0 ... vertexCount]
37      */
38
39 } // namespace NMyGraph
```

В *graph.cpp* реализуем методы класса *TGraph*. В момент создания объекта этого класса будем сохранять граф в двух экземплярах: в виде вектора рёбер (для алгоритма Форда-Беллмана) и списка смежности (для алгоритма Дейкстры). При обнаружении отрицательного циклы происходит выбрасывание соответствующего исключения.

В алгоритме Дейкстры используется *std::priority_ queue*, поскольку на каждой итерации нас интересует вершина с `минимальным` расстоянием до заданной вершины, то в очередь будем добавлять отрицательные расстояния.

```
  #include "graph.h"
#include <queue>

namespace NGraph {

    std::istream& operator>> (std::istream& is, TEdge& edge) {
        size_t from, to;
        is >> from >> to >> edge.cost;
        edge.from = from - 1;
        edge.to = to - 1;
        return is;
    }

    TGraph::TGraph(const std::vector<TEdge>& graphEdges, const size_t vertexesCount)
        : vertexCount(vertexesCount),
        edges(graphEdges)

    {
        TAdjacencyList tmpAdjList(vertexCount,
                TVertex(0));
        for (size_t i = 0; i < edges.size(); ++i) {
            const TEdge& currEdge = edges[i];
            tmpAdjList[currEdge.from].push_back(std::make_pair(currEdge.to, currEdge.
                cost));
        }
        adjList = std::move(tmpAdjList);
    }

    TGraph::TGraph(const size_t vertexesCount, const size_t edgesCount)
        : vertexCount(vertexesCount),
        edges(edgesCount),
        adjList(vertexesCount,TVertex(0))
    {}

    std::istream& operator>> (std::istream& is, TGraph& graph) {
        for (size_t i = 0; i < graph.edges.size(); ++i) {
            TEdge& edge = graph.edges[i];
            is >> edge;
            graph.adjList[edge.from].push_back(std::make_pair(edge.to, edge.cost));
        }
```

```
40          return is;
41      }
42
43      std::vector<long long> TGraph::FordBellman(const size_t target) {
44          std::vector<long long> distances (vertexCount, INF);
45          distances[target] = 0;
46          for (size_t i = 0; i <= vertexCount; ++i) {
47              bool changed = false;
48              for (size_t j = 0; j < edges.size(); ++j) {
49                  const size_t & from = edges[j].from ;
50                  const size_t & to = edges[j].to;
51                  const long long cost = edges[j].cost;
52                  if (distances[from] < INF && distances[to] > distances[from] + cost) {
53                      changed = true;
54                      distances[to] = std::max(
55                              -INF,
56                              distances[from] + cost
57                      );
58                  }
59              }
60              if (!changed) {
61                  break;
62              } else if (i == vertexCount) {
63                  throw std::runtime_error("Negative cycle");
64              }
65          }
66          return distances;
67      }
68
69      std::vector<long long> TGraph::Dijkstra(const size_t target) {
70          std::vector<long long> distances (vertexCount, INF);
71          distances[target] = 0;
72          std::priority_queue< std::pair<long long, size_t> > q;
73          q.push (std::make_pair (0, target));
74
75          while (!q.empty()) {
76              size_t vertex = q.top().second;
77              long long curDistance = -q.top().first;
78              q.pop();
79              if (curDistance > distances[vertex]) {
80                  continue;
81              }
82
83              if (vertex < adjList.size()) {
84                  for (size_t j = 0; j < adjList[vertex].size(); ++j) {
85                      size_t to = adjList[vertex][j].first;
86                      long long cost = adjList[vertex][j].second;
87
88                      if (distances[vertex] + cost < distances[to]) {
```

```
89                      distances[to] = distances[vertex] + cost;
90                      q.push(std::make_pair (-distances[to], to));
91                  }
92              }
93          }
94      }
95
96      return distances;
97  }
98
99  std::vector< std::vector<long long> > TGraph::Jonshon() {
100     for (size_t i = 0; i < vertexCount; ++i) {
101         edges.push_back( {vertexCount, i, 0} );
102     }
103     ++vertexCount;
104     std::vector<long long> newCosts = FordBellman(vertexCount - 1);
105     --vertexCount;
106
107     for (size_t fromIdx = 0; fromIdx < vertexCount; ++fromIdx) {
108         TVertex & from = adjList[fromIdx];
109         for (size_t to = 0; to < from.size(); ++to) {
110             from[to].second += (newCosts[fromIdx] - newCosts[from[to].first]);
111         }
112     }
113
114     std::vector< std::vector<long long> > result (
115             vertexCount,
116             std::vector<long long> (vertexCount)
117             );
118     for (size_t from = 0; from < vertexCount; ++from) {
119         result[from] = Dijkstra(from);
120         for (size_t to = 0; to < result[from].size(); ++to) {
121             if (result[from][to] != INF) {
122                 result[from][to] += (newCosts[to] - newCosts[from]);
123             }
124         }
125     }
126
127     return result;
128 }
129
130 } // namespace NMyGraph
```

В *main.cpp* произведём считывание графа и вывод результата работы алгоритма Джонсона, причём выполнение происходит в блоке *try - catch* для обработки исключения, связанного с наличием отрицательного цикла.

```cpp
  #include <iostream>
#include "graph.h"

int main() {
    std::ios_base::sync_with_stdio(false);

    size_t vertexCount, edgesCount;
    std::cin >> vertexCount >> edgesCount;
    NGraph::TGraph graph(vertexCount, edgesCount);

    std::cin >> graph;

    try {
        std::vector< std::vector<long long> > result = graph.Jonshon();

        for (size_t i = 0; i < result.size(); ++i) {
            for (size_t j = 0; j < result[i].size(); ++j) {
                long long & cost = result[i][j];
                if (cost == NGraph::INF) {
                    std::cout << "inf";
                } else {
                    std::cout << cost;
                }
                if (j != result[i].size() - 1) {
                    std::cout << " ";
                }
            }
            std::cout << std::endl;
        }
    } catch (std::exception & ex) {
        std::cout << ex.what() << std::endl;
    }

    return 0;
}
```

3 Консоль

```
MacBook-Pro:da_lab_09 mr-ilin$ ./wrapper.sh
[2021-06-10 15:55:26] [INFO] Compiling...
g++ -std=c++17 -pedantic -Wall -Wno-unused-variable -c main.cpp -o main.o
g++ -std=c++17 -pedantic -Wall -Wno-unused-variable -c graph.cpp -o graph.o
g++ -std=c++17 -pedantic -Wall -Wno-unused-variable main.o graph.o -o solution
[2021-06-10 15:55:27] [INFO] Making unittest...
g++ -std=c++17 -pedantic -Wall -Wno-unused-variable -g -fsanitize=address -DDEBUG
-c unit_tests.cpp -o unit_tests.o
g++ -std=c++17 -pedantic -Wall -Wno-unused-variable -g -fsanitize=address -DDEBUG
unit_tests.o graph.o -o unit_test -lgtest -lgtest_main
[==========] Running 4 tests from 3 test suites.
[----------] Global test environment set-up.
[----------] 2 tests from FordBellmanSuite
[ RUN      ] FordBellmanSuite.FordBellmanAnswerTest
[       OK ] FordBellmanSuite.FordBellmanAnswerTest (0 ms)
[ RUN      ] FordBellmanSuite.FordBellmanNegativeCycleTest
[       OK ] FordBellmanSuite.FordBellmanNegativeCycleTest (0 ms)
[----------] 2 tests from FordBellmanSuite (0 ms total)

[----------] 1 test from DijkstraSuite
[ RUN      ] DijkstraSuite.DijkstraAnswerTest
[       OK ] DijkstraSuite.DijkstraAnswerTest (0 ms)
[----------] 1 test from DijkstraSuite (0 ms total)

[----------] 1 test from JonshonSuite
[ RUN      ] JonshonSuite.JonshonAnswerTest
[       OK ] JonshonSuite.JonshonAnswerTest (0 ms)
[----------] 1 test from JonshonSuite (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 3 test suites ran. (1 ms total)
[  PASSED  ] 4 tests.
[2021-06-10 15:55:30] [INFO] Executing tests/t00.t...
OK
[2021-06-10 15:55:30] [INFO] No failed tests,hooray
MacBook-Pro:da_lab_09 mr-ilin$ ./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
```

4 Выводы

Выполнив девятую лабораторную работу по курсу «Дискретный анализ», я вспомнил как работать с графами. Повторил ранее изученные алгоритмы Форда-Беллмана и Дейкстры, с помощью которых реализовал алгоритм Джонсона. Особенно пригодились конспекты олимпиадного программирования, в которых был подробный разбор первых двух алгоритмов, а также навыки написания юнит тестов, с помощью которых удалось избавиться от всех ошибок ещё до финальной версии программы, поэтому она прошла тестирование с первого раза.

Про улучшение алгоритма. Можно реализовать алгоритм Дейкстры с помощью Фибоначчиевой кучи, тогда можно добиться итоговой сложности $O(n^2 \cdot log(n) + n \cdot m)$.

Список литературы

[1] *Алгоритм Джонсона — Вики ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Джонсона` (дата обращения: 01.06.2021).