МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

**«Московский технический университет связи и информатики»**

Кафедра «Информационные технологии»

Лабораторная работа №4

**«Разработка алгоритмов нахождения решения задач о максимальном потоке и назначениях»**

по дисциплине

Информационные технологии и программирование

Выполнил: студент гр. БЭИ2202

Кулешов А. С.

Вариант 16

Проверил: Халабия Р.Ф.

Москва, 2024 г

1. Цели и задачи

Изучить понятия, формирования, особенности решения задач о назначениях и максимальном потоке, с использованием алгоритмов рекурсии, а также алгоритмов нахождения максимального потока и назначения.

1. Постановка задачи

2.1 Дана сеть G=(I,U) . В таблице 2 ниже приведены множества I, U и веса дуг соответственно перечислению их в U. Вес дуги – пропускная способность дуги. Считаем вершину 1 источником, вершину 9 – стоком. Написать программу:

- поиска потока максимальной величины из источника в сток, используя алгоритм Форда-Фалкерсона.

- поиска соответствующего найденному потоку разреза минимальной пропускной способности.

2.2 Написать программу для:

а) поиска оптимального назначения, минимизирующее суммарную стоимость назначений.

б) поиска оптимального назначение, максимизирующее суммарную эффективность назначений.

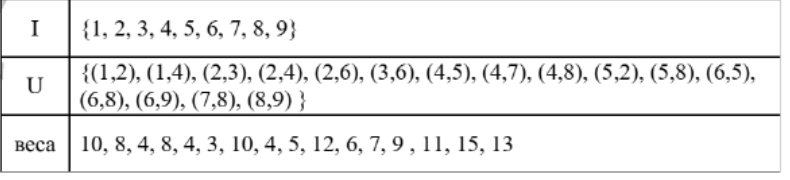


Рисунок 1 – условие индивидуального варианта (первая часть)

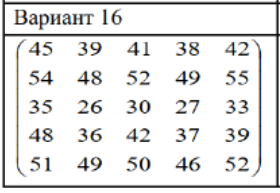


Рисунок 2 – условие индивидуального варианта (вторая часть)

1. Листинг программы

pair<**int**, std::vector<std::vector<**int**>> > fordFulkerson(std::vector<std::vector<std::pair<**int**, **int**>>> graph\_adj, **int** source, **int** sink) {

**int** N = graph\_adj.size();

**const** **auto**& capacity = to\_adjacency\_matrix(graph\_adj);

std::vector<std::vector<**int**>> adj(N);

**for** (**int** from = **0**; from < N; ++from) {

**for** (**const** **auto**& e : graph\_adj[from]) {

**int** to = e.first;

adj[from].push\_back(to);

adj[to].push\_back(from); // Развернём граф

}

}

std::vector<std::vector<**int**>> flow;

flow.resize(N);

**for** (**int** i = **0**; i < N; ++i) {

flow[i].resize(N);

**for** (**int** j = **0**; j < N; ++j) {

flow[i][j] = **0**;

}

}

**double** maxFlow = **0**;

// Ford-Fulkerson algorithm using BFS (Edmonds-Karp implementation)

**while** (true) {

// Find an augmenting path

std::vector<**int**> parent(N, -**1**);

std::queue<**int**> q;

q.push(source);

parent[source] = -**2**; // Mark the source node

**while** (!q.empty()) {

**int** u = q.front();

q.pop();

**for** (**int** v : adj[u]) {

// Check for residual capacity

**if** (parent[v] == -**1** && capacity[u][v] - flow[u][v] > **0**) {

parent[v] = u;

**if** (v == sink)

**break**; // Early exit if sink is reached

q.push(v);

}

}

}

// No augmenting path found

**if** (parent[sink] == -**1**)

**break**;

// Find minimum residual capacity along the path

**int** pathFlow = inf;

**for** (**int** v = sink; v != source; v = parent[v]) {

**int** u = parent[v];

pathFlow = std::min(pathFlow, capacity[u][v] - flow[u][v]);

}

// Update flows along the path

**for** (**int** v = sink; v != source; v = parent[v]) {

**int** u = parent[v];

flow[u][v] += pathFlow;

flow[v][u] -= pathFlow; // Update reverse flow

}

// Add to max flow

maxFlow += pathFlow;

}

**return** { maxFlow, flow };

}

std::pair<**int**, std::vector<std::vector<**int**>> > findMinimalCut(std::vector<std::vector<std::pair<**int**, **int**>>> graph\_adj, **int** source, **int** sink) {

**int** N = graph\_adj.size();

**const** **auto**& capacity = to\_adjacency\_matrix(graph\_adj);

// Build adjacency list for the residual graph

std::vector<std::vector<**int**>> adj(N);

**for** (**int** from = **0**; from < N; ++from) {

**for** (**const** **auto**& e : graph\_adj[from]) {

**int** to = e.first;

adj[from].push\_back(to);

adj[to].push\_back(from); // Развернём граф

}

}

std::vector<std::vector<**int**>> flow = fordFulkerson(graph\_adj, source, sink).second;

// Find vertices reachable from source in residual graph

std::vector<**bool**> visited(N, false);

std::queue<**int**> q;

q.push(source);

visited[source] = true;

**while** (!q.empty()) {

**int** u = q.front();

q.pop();

**for** (**int** v : adj[u]) {

**if** (!visited[v] && capacity[u][v] - flow[u][v] > **0**) {

visited[v] = true;

q.push(v);

}

}

}

// Output the minimal cut edges and calculate its capacity

std::vector<std::vector<**int**>> res;

**int** minCutCapacity = **0**;

**for** (**int** u = **0**; u < N; ++u) {

**if** (visited[u]) {

**for** (**int** v = **0**; v < N; ++v) {

**if** (!visited[v] && capacity[u][v] > **0** && capacity[u][v] != inf) {

// Edge u -> v (capacity[u][v])

res.push\_back({ u + **1**, v + **1**, capacity[u][v] });

minCutCapacity += capacity[u][v];

}

}

}

}

**return** { minCutCapacity, res };

}

std::vector<**int**> hungarianMethod(**const** std::vector<std::vector<**int**>>& costMatrix) {

**int** n = costMatrix.size();

std::vector<**int**> u(n + **1**), v(n + **1**), p(n + **1**), way(n + **1**);

**for** (**int** i = **1**; i <= n; i++) {

p[**0**] = i;

std::vector<**int**> minv(n + **1**, std::numeric\_limits<**int**>::max());

std::vector<**char**> used(n + **1**, false);

**int** j0 = **0**;

**do** {

used[j0] = true;

**int** i0 = p[j0], delta = std::numeric\_limits<**int**>::max(), j1;

**for** (**int** j = **1**; j <= n; j++) {

**if** (!used[j]) {

**int** cur = costMatrix[i0 - **1**][j - **1**] - u[i0] - v[j];

**if** (cur < minv[j]) {

minv[j] = cur;

way[j] = j0;

}

**if** (minv[j] < delta) {

delta = minv[j];

j1 = j;

}

}

}

**for** (**int** j = **0**; j <= n; j++) {

**if** (used[j]) {

u[p[j]] += delta;

v[j] -= delta;

}

**else**

minv[j] -= delta;

}

j0 = j1;

} **while** (p[j0]);

**do** {

**int** j1 = way[j0];

p[j0] = p[j1];

j0 = j1;

} **while** (j0);

}

std::vector<**int**> assignment;

assignment.resize(n);

**for** (**int** j = **1**; j <= n; j++)

assignment[p[j] - **1**] = j - **1**;

**return** assignment;

}

1. Псевдокод

Алгоритм Форда-Фалкерсона

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим любой путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный путь (он называется ***увеличивающим путём*** или ***увеличивающей цепью***) максимально возможный поток:
   1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью cmin cmin.
   2. Для каждого ребра на найденном пути увеличиваем поток на cmincmin, а в противоположном ему — уменьшаем на cmin cmin.
   3. Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных (антипараллельных) им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.
4. Возвращаемся на шаг 2.

Венгерский алгоритм

* 1. Дополнить матрицу до квадратной, если необходимо
  2. Из каждой строки вычесть минимальный элемент строки, из каждого столбца вычесть минимальный элемент столбца
  3. Найти минимальное количество строк и столбцов, необходимых для покрытия всех нулей.
  4. Если их сумма равна размеру матрицы, то алгоритм закончен и результат получен (назначения – найденные нули)
  5. Найти минимальный элемент среди непокрытых элементов и вычесть его из непокрытых, добавить к покрытым
  6. Вернуться на шаг 3.

1. Контрольный тест. Результаты программы

Результат запуска программы на контрольных данных можно увидеть на рисунке 3.

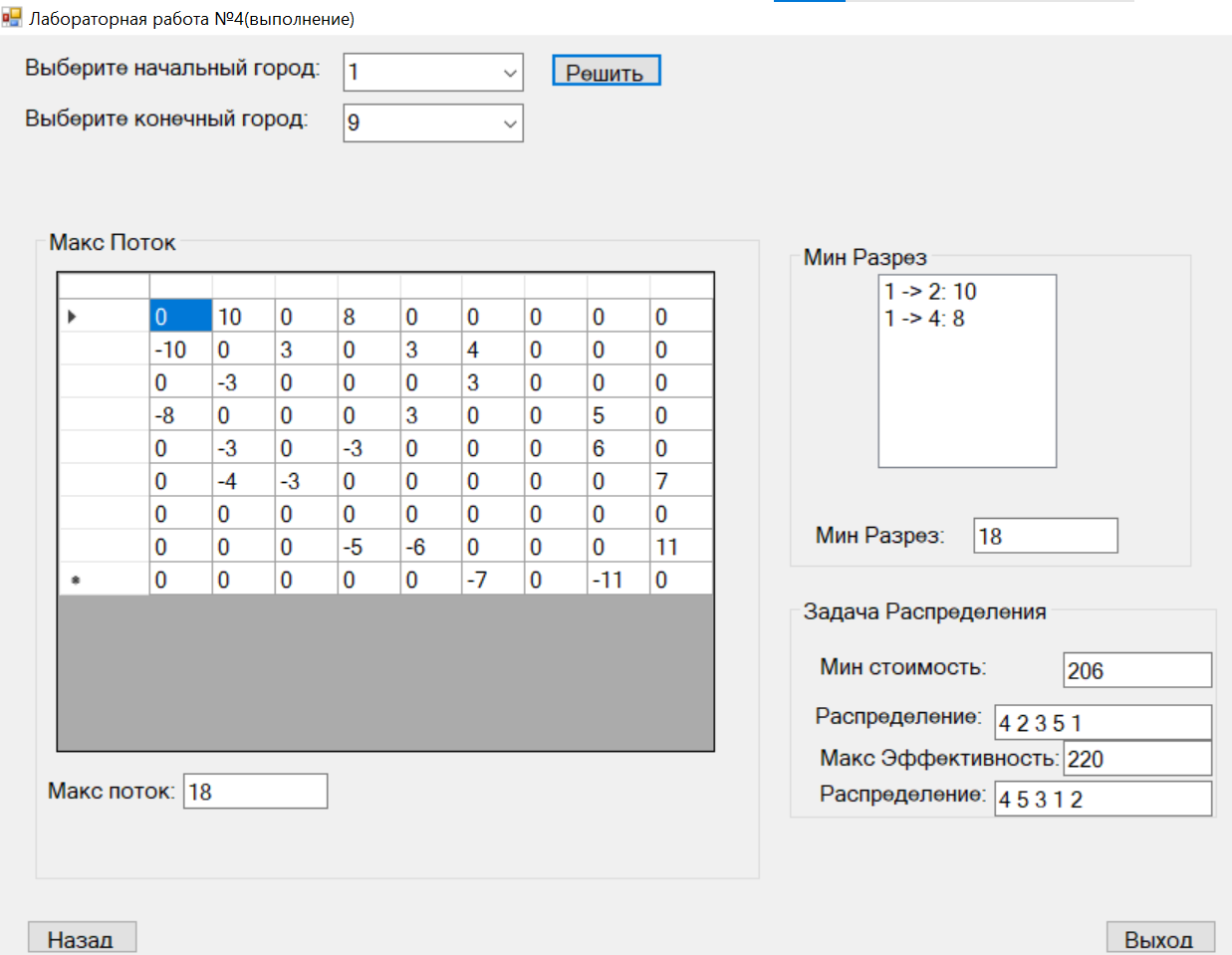


Рисунок 3 – результат запуска программы

1. Выводы по работе

В результате выполнения лабораторной работы мы смогли реализовать несколько алгоритмов для работы над графами: Форда-Фалкерсона, Венгерский алгоритм