|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
| path817.png | | |
| Кафедра прикладной математики | | |
|  | | |
| Практическое задание | | |
| по дисциплине «Основа теории графов» | | |
|  | | |
| **Алгоритм Дейкстры** | | |
|  | | |
| эмблема_светлая.png | Факультет: | ПМИ |
| Группа: | ПМ-84 |
|  | Место для ввода текста. |
| Студенты: | Фадейкин Леонид, |
|  | Салахиев Вячеслав, |
|  | Пешкин Семен |
| Преподаватель: | Рояк Михаил Эммануилович |
|  | | |
| Новосибирск | | |
| 2019 | | |

1. **Условие задачи**

На языке С++ реализовать алгоритм Дейкстры.

1. **Анализ задачи**

Алгоритм Дейкстры можно реализовать несколькими способами, для нашей работы мы выбрали два: простейшая реализация через матрицу смежности и реализация через список смежности с использованием очереди с приоритетом.

В обоих реализациях для хранения длин ребер в графе используется массив чисел, а для хранения принадлежности элемента множеству *А* – массив булевых переменных, где *А* - множество просмотренных вершин. В начале алгоритма расстояние для начальной вершины полагается равным нулю, а все остальные расстояния заполняются *-1*. Массив флагов заполняется значением *false*. Затем запускается основной цикл. На каждом шаге цикла мы ищем вершину *V* с минимальным расстоянием и флагом равным *false*. Затем мы устанавливаем в ней флаг в true и проверяем все соседние с ней вершины *U*. Если в них расстояние больше, чем сумма расстояния до текущей вершины и длина ребра, то уменьшаем его. Цикл завершается, когда флаги всех вершин становятся равны *true*, либо, когда у всех вершин с флагом *false* длина ребра равна *– 1*. Последний случай возможен тогда и только тогда, когда граф G несвязный.

1. Матрица смежности

Граф представлен двумерным массивом элементов, хранящий длины ребер инцидентных вершин, размерности , где – количество вершин в графе.

Поиск минимального пути осуществляется просмотром массива, хранящего длины ребер инцидентных вершин.

|  |  |  |  |
| --- | --- | --- | --- |
| N | **0** | **1** | **2** |
| **0** | 0 | 12 | 0 |
| **1** | 12 | 0 | 5 |
| **2** | 0 | 5 | 0 |

*Выделенные жирным – номера вершин.*

*На перекрестии – длина ребра (0 – если ребро между вершинами отсутствует).*

1. Список смежности

В данной реализации граф представлен двумерным массивом, хранящий структуру элементов *neighbor*, в которой содержится номер инцидентной вершины и длину пути до нее. Поиск минимального пути сводится до минимума: организация приоритетной очереди по возрастанию.

Пример элемента массива, хранящего элементы структуры *neighbor* :  
*graph[a] = v(b, c),* где *a* – рассматриваемая вершина, *b* – смежная с ней вершина и *c* – расстояние между ними.

1. **Текст программы**

graph.cpp

#include <queue>

#include <iostream>

#include <vector>

#include <fstream>

#include <string>

#include <time.h>

using namespace std;

struct neighbor {

int number, weight;

neighbor(int \_number = -1, int \_weight = -1) : number(\_number), weight(\_weight) {}; // структура элемента вершины, содержащую номер инцидентной

}; // вершины и вес ребра.

void matrixResult(ifstream \*fIn, ofstream \*fOut) {

int n, m, a, b, c, v, temp; bool stop; clock\_t t;

\*fIn >> n >> m >> v; // ввод с файла: кол-во вершин, кол-во ребер, номер // вершины, с которой осуществляется поиск

vector<vector<int>> matrix; matrix.resize(n); // объявляем матрицу

vector<int> column; column.assign(n, -1); // смежнсти; массив, содержащий

vector<bool> check; check.assign(n, false); // длины ребер; массив флагов

t = clock();

for (int i = 0; i < n; i++)

matrix[i].resize(n);

for (int i = 0; i < m; i++) { // заполняем матрицу смежности

\*fIn >> a >> b >> c;

matrix[a][b] = c;

matrix[b][a] = c;

}

column[v] = 0; // отмечаем начальную вершину

temp = 0; // временная переменная, которая хранит минимальное расстояние

while (temp != -1) {

stop = false;

for (int i = 0; i < n && !stop; i++) // поиск вершины, расстояние до

if (column[i] == temp && !check[i]) { // которой минимальное

for (int j = 0; j < n; j++) { // просмотр вершин смежных с найденной

if (matrix[i][j] > 0 && !check[j]) {

if (column[j] != -1) {

if (column[j] > temp + matrix[i][j])

column[j] = temp + matrix[i][j];

}

else

column[j] = temp + matrix[i][j];

}

}

check[i] = true; // отмечаем просмотренную вершину

stop = true; // конец поиска минимального расстояния

}

temp = -1;

for (int i = 0; i < n; i++) { // поиск минимального расстояния

if (column[i] > 0 && !check[i])

if (temp == -1)

temp = column[i];

else

if (column[i] < temp)

temp = column[i];

}

}

t = clock() - t;

cout << ((double)t) / CLOCKS\_PER\_SEC << endl; // вывод времени выполнения

if (fOut) { // вывод результатов

for (int i = 0; i < column.size(); i++) {

if (column[i] == -1)

\*fOut << "Вершина " << v << " не связана с вершиной " << i << endl;

else

if (column[i] > 0)

\*fOut << v << " - " << i << " : " << column[i] << endl;

}

\*fOut << endl;

}

}

void listResult(ifstream \*fIn, ofstream \*fOut) {

int a, b, c, temp, dist, n, m, v; clock\_t t;

\*fIn >> n >> m >> v; // ввод с файла: кол-во вершин, кол-во ребер, номер // вершины, с которой осуществляется поиск

bool stop = false, flag = false;

vector<vector<neighbor>> graph; graph.resize(n); // объявляем список смежности

vector<int> column; column.assign(n, -1); // массив, содержащий длины ребер

vector<bool> check; check.assign(n, false); // массив флагов

priority\_queue<int, vector<int>, greater<int>> pQueue; // очередь с // приоритетом, хранящая длины ребер

t = clock();

for (int i = 0; i < m; i++) { // заполняем список смежности

\*fIn >> a >> b >> c;

neighbor u(a, c), v(b, c);

graph[a].push\_back(v);

graph[b].push\_back(u);

}

column[v] = 0; // отмечаем начальную вершину

pQueue.push(0);

while (!pQueue.empty() && !stop) {

dist = pQueue.top();

pQueue.pop();

flag = false;

for (int i = 0; i < column.size() && !flag; i++) { // поиск вершины,

if (column[i] == dist && !check[i]) { // расстояние до которого

temp = i; // минимальное

flag = true;

}

}

if (flag) {

check[temp] = true; // отмечаем просмотренную вершину

for (int i = 0; i < graph[temp].size(); i++) {

if (!check[graph[temp][i].number]) { // просмотр вершин смежных с

if (column[graph[temp][i].number] == -1) { // найденной

column[graph[temp][i].number] = dist + graph[temp][i].weight;

pQueue.push(graph[temp][i].weight + dist);

}

else

if (column[graph[temp][i].number] > dist + graph[temp][i].weight) {

column[graph[temp][i].number] = dist + graph[temp][i].weight;

pQueue.push(graph[temp][i].weight + dist); // кладем в

} // расстояние до смежной вершины

}

}

}

else stop = true;

}

t = clock() - t;

cout << ((double)t) / CLOCKS\_PER\_SEC << endl; // вывод времени выполнения

if (fOut) {

for (int i = 0; i < column.size(); i++) { // вывод результатов

if (column[i] == -1)

\*fOut << "Вершина " << v << " не связана с вершиной " << i << endl;

else

if (column[i] > 0)

\*fOut << v << " - " << i << " : " << column[i] << endl;

}

\*fOut << endl;

}

}

int main() {

ifstream fIn; ofstream fOut;

ofstream TableOut;

TableOut.open("table.txt");

If (TableOut)

for (int i = 100; i < 1001; i += 100) {

fOut.open("testAlot.txt");

generator(&fOut, i, (i \* (i - 1) \* 0.5)); // генерируем тест

fOut.close();

TableOut << i << "\t";

fIn.open("testAlot.txt");

if (fIn) matrixResult(&fIn, &TableOut);

fIn.close();

fIn.seekg(0);

fIn.open("testAlot2.txt");

if (fIn) listResult(&fIn, &TableOut);

fIn.close();

TableOut << endl;

}

TableOut.close(); return 0;

}

1. **Тип генерации**

Для решения задачи было реализовано два типа генерации:

* Генерация случайного графа
* Генерация полного графа

#include <iostream>

#include <vector>

#include <fstream>

#include <time.h>

#include <random>

#include <string>

using namespace std;

const int Nmax = 1000;

struct neighbor {

int number, weight;

neighbor(int \_number = -1, int \_weight = -1) : number(\_number), weight(\_weight) {};

};

void generator(ofstream \*file, int n) { // генерация случайного графа

int v = 0;

m = (rand () % (n \* (n – 1) / 2)) + 1;

v = rand() % n;

\*file << n << ' ' << m << ' ' << v << endl;

vector<neighbor> tempVec; // объявляем массив, где будем запоминать записанные

int temp0, temp1, iter = 0; bool flag; // ребра

while (iter < m) {

temp0 = rand() % n; // используя структуру neighbor мы можем обозначить

temp1 = rand() % n; // элемент как ребро между двумя вершинами: temp0

neighbor temp(temp0, temp1); // – первая вершина, temp1 – вторая вершина

if (iter == 0) {

if (temp.number != temp.weight) { // если вершины не совпадают

tempVec.push\_back(temp);

\*file << temp0 << ' ' << temp1 << ' ' << rand() % 20 + 1 << endl;

iter++;

}

}

else {

flag = true;

for (int j = 0; j < tempVec.size() && flag; j++) {

if (temp.number == tempVec[j].number && temp.weight == tempVec[j].weight

|| temp.number == tempVec[j].weight && temp.weight == tempVec[j].number

|| temp.number == temp.weight) { // смотрим, было ли уже такое

flag = false; // ребро записано

}

}

if (flag) {

tempVec.push\_back(temp); // запоминаем ребро

\*file << temp0 << ' ' << temp1 << ' ' << rand() % 20 + 1 << endl;

iter++;

}

}

}

\*file << endl;

}

void fullGen(ofstream \*file, int n) { // генерация полного графа

int m = n \* (n - 1) / 2, v = rand() % n;

\*file << n << ' ' << m << ' ' << v << endl;

for (int i = 0; i < n; i++) {

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

\*file << i << ' ' << j << ' ' << rand() % 20 + 1 << endl;

}

\*file << endl;

}

int main() {

ofstream fOut; string str; bool flag = true; int input, num, n, m;

srand(time(NULL));

while (flag) {

cout << "======" << endl;

cin >> input;

if (input == -1)

flag = false;

else {

str = "test" + to\_string(input) + ".txt";

fOut.open(str);

cin >> num >> n;

switch (num) {

case 1:

generator(&fOut, n);

break;

case 2:

fullGen(&fOut, n);

break;

}

generator(&fOut);

fOut.close();

}

}

fOut.close();

}

1. **Результаты и графики**

Для составления объективных результатов работы программы (по времени) было принято решение сделать большое количество тестов с фиксированием числа ребер относительно числа вершин и небольшим шагом в изменении числа вершин.

В полученной реализации программы мы сможем запускать генерирование и обработку тестов, в которых количество ребер зависит от числа вершин, с любым шагом и в любых диапазонах.

Для отчета были выбраны следующие данные:

1. реализация через матрицу смежности
2. реализация через очередь с приоритетом

тношение числа ребер к числу вершин:

отношение числа ребер к числу вершин:

отношение числа ребер к числу вершин: (полный граф)

1. **Сложность алгоритма**

Для этого рассмотрим два случая.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **100** | **1000** | **10000** |
| **Время (матрица)** | 0 | 0.83 | 88.53 |
| **Время (список)** | 0 | 0.8 | 89.07 |

Данный тест был проведен на полных графах, количеством вершин 100, 1000 и 10000 соответственно. Как мы видим, при увеличении числа вершин в 10 раз, время увеличивается приблизительно в 100 раз. Это говорит об общем времени работы алгоритма при данной зависимости числа вершин и ребер -

Для проверки данной зависимости мы решили взять промежуток . И построить графики по полученным значениям, и сравнить их с теоретической

Если проанализировать график подпрограммы реализации матрицей, видна квадратичная зависимость. В случае же с графиком с приоритетом, зависимость времени от количества вершин близка к зависимости в графике .

1. **Выводы**

Мы сделали две реализации алгоритма Дейкстры. Через матрицу смежности и через очередь с приоритетом. По полученным результатам можно судить, что в случае неполного графа первая реализация значительно проигрывает второй по времени. В случае же полного графа реализации выдают близкое время работы. А функция отношения времени к количеству вершин квазилинейна.