**ИТМО Кафедра Информатики и прикладной математики**

Отчет по лабораторной работе №5

«Поиск пути между городами»   
Вариант 4

**Выполнил: студент группы P3217**

**Плюхин Дмитрий**

**Преподаватель: Зинчик А. А.**

**2017 год**

1. **Задание**

Дана система односторонних дорог, определяемая набором пар городов. Каждая такая пара (i,j) указывает, что из города i можно проехать в город j, но это не значит, что можно проехать в обратном направлении. Необходимо определить, можно ли проехать из заданного города А в заданный город В таким образом, чтобы посетить город С и не проезжать ни по какой дороге более одного раза. Входные данные задаются в файле с именем PATH.IN следующим образом. В первой строке находится натуральное N(N<=50) - количество городов (города нумеруются от 1 до N). Во второй строке находится натуральное M(M<=100) - количество дорог. Далее в каждой строке находится пара номеров городов, которые связывает дорога. В последней (M+3)-й строке находятся номера городов А, В и С. Ответом является последовательность городов, начинающаяся городом А и заканчивающаяся городом В, удовлетворяющая условиям задачи, который должен быть записан в файл PATH.OUT в виде последовательности номеров городов по одному номеру в строке. Первая строка файла должна содержать количество городов в последовательности. При отсутствии пути записать в первую строку файла число -1.

1. **Исходный код**

**Основная функция, в которой происходит парсинг исходного файла, вывод результата и запуск функции поиска пути**

int main(int argc, char\* argv[]){

//

// Parsing file

//

string readStr;

ifstream file("path.in");

getline(file, readStr);

int numberOfVertexes = atoi(readStr.c\_str());

getline(file, readStr);

int numberOfEdges = atoi(readStr.c\_str());

GraphAdjRecordPointer graphAdj = ( GraphAdjRecordPointer )malloc( (numberOfVertexes + 1) \* sizeof(GraphAdjRecord) );

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

graphAdj[i].next = NULL;

graphAdj[i].name = 0;

}

for (int i = 0; i < numberOfEdges; i++) { getline(file, readStr); addNewEdge(readStr, graphAdj); }

showGraph(graphAdj, numberOfVertexes);

getline(file, readStr);

file.close();

int firstSpacePosition = readStr.find(' ');

int secondSpacePosition = readStr.find(' ', firstSpacePosition + 1);

int startVertex = atoi(readStr.substr(0,firstSpacePosition).c\_str());

int endVertex = atoi(readStr.substr(firstSpacePosition + 1,secondSpacePosition - firstSpacePosition - 1).c\_str());

int midVertex = atoi(readStr.substr(secondSpacePosition + 1,readStr.length() - secondSpacePosition).c\_str());

//

// Counting

//

QueueRecordPointer queueStart = NULL;

QueueRecordPointer queueEnd = NULL;

PathPointer path = findPath(graphAdj, numberOfVertexes, startVertex, endVertex, midVertex, numberOfEdges);

//

// Writing to an output file

//

ofstream myfile;

myfile.open ("path.out");

if (path == NULL){

myfile << "-1";

} else {

myfile << path->length << endl;

for (int i = path->length - 1; i >= 0; i--) myfile << path->values[i] << " ";

myfile << endl;

}

myfile.close();

}

**Функция поиска пути, использующая алгоритм поиска в ширину**

PathPointer findPath(GraphAdjRecordPointer graphAdj, int numberOfVertexes, int startVertex, int endVertex, int midVertex, int numberOfEdges){

short\* color = (short\*)malloc((numberOfVertexes + 1) \* sizeof(short));

int\* predecessor = (int\*)malloc((numberOfVertexes + 1) \* sizeof(int));

bool result = BreadthFirstSearch(color, predecessor, numberOfVertexes, graphAdj, midVertex, endVertex);

if (result == false) return NULL;

showArr(predecessor, numberOfVertexes);

int pathVertexIndex = 0;

int currentPathVertex = endVertex;

PathPointer path = (PathPointer)malloc(sizeof(PathPointer));

path->values = (int\*)malloc(numberOfEdges \* sizeof(int));

while (currentPathVertex != 0){

path->values[pathVertexIndex] = currentPathVertex;

pathVertexIndex++;

currentPathVertex = predecessor[currentPathVertex];

}

result = BreadthFirstSearch(color, predecessor, numberOfVertexes, graphAdj, startVertex, midVertex);

if (result == false) return NULL;

showArr(predecessor, numberOfVertexes);

currentPathVertex = predecessor[midVertex];

while (currentPathVertex != 0){

path->values[pathVertexIndex] = currentPathVertex;

pathVertexIndex++;

currentPathVertex = predecessor[currentPathVertex];

}

path->length = pathVertexIndex;

return path;

}

**Функция, реализующая модифицированный алгоритм поиска в ширину**

bool BreadthFirstSearch(short\* color, int\* predecessor, int numberOfVertexes, GraphAdjRecordPointer graphAdj, int startVertex, int endVertex){

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

color[i] = 0;

predecessor[i] = 0;

}

color[startVertex] = 1;

QueueRecordPointer queueStart = NULL;

QueueRecordPointer queueEnd = NULL;

enqueue(&queueStart, &queueEnd, startVertex);

int exploringVertex;

GraphAdjRecordPointer connectedVertex;

while (queueEnd != NULL){

exploringVertex = dequeue(&queueStart, &queueEnd);

connectedVertex = graphAdj[exploringVertex].next;

while (connectedVertex != NULL){

if (color[connectedVertex->name] == 0){

color[connectedVertex->name] = 1;

predecessor[connectedVertex->name] = exploringVertex;

if (connectedVertex->name == endVertex) return true;

enqueue(&queueStart, &queueEnd, connectedVertex->name);

}

connectedVertex = connectedVertex->next;

}

color[exploringVertex] = 2;

}

return false;

}

1. **Примеры работы алгоритма**

|  |  |  |  |
| --- | --- | --- | --- |
| Файл path.in  9  10  1 2  2 6  6 8  8 4  1 5  5 3  3 9  9 7  7 1  5 4  1 4 6 | Файл path.out  5  1 2 6 8 4 | Файл path.in  9  10  1 2  2 6  6 8  8 4  1 5  5 3  3 9  9 7  7 1  5 4  1 5 8 | Файл path.out  -1 |

1. **Вывод**

В лабораторной работе был применен на практике алгоритм поиска в ширину и сделан вывод о том, что с его помощью путем частичной модификации можно решать широкий круг задач алгоритмизации, связанных с обработкой графов. В качестве альтернативной реализации также может быть рассмотрена версия программы, запускающая два параллельно выполняющихся потока – в этом случае появится необходимость использования объектов синхронизации, однако, надо полагать, временная эффективность алгоритма несколько повысится в ряде случаев (а именно, в тех, где примерно одинакова длина путей от начальной до промежуточной и от промежуточной до конечной вершины), но не в большой степени (поскольку асимптотически оно останется таким же) – при том основным условием повышения скорости является возможность использования двух вычислительных ядер.