За основу алгоритма для нахождения гамильтонового цикла в невзвешенном графе возьмём так называемый перебор с возвратом. Такой подход позволяет к уже выстроенному отрезку маршрута прибавляться лишь те вершины, которые соединяются ребром с конечной вершиной маршрута и не посещались раньше. После прибавления новой вершины к пути происходит рекурсия, то есть запуск алгоритма из новой вершины. В случае возврата пути из какой-либо вершины на предыдущую точку, с оставленной вершины удаляется пометка о её посещении. При таком способе возможен возврат алгоритма на эту же вершину повторно, но уже другим путём. Более того, это обязательно случится, если гамильтонов цикл существует в рассматриваемом графе, поскольку путь должен пройти через каждую вершину.

Реализация данного алгоритма на языке программирования C++ представлена в листинге 1. В программе используется пользовательский класс Graph и методы работы с ним. Реализация данного класса приведена в листинге 1 приложения 1.

**Листинг 2.1 – Программа для поиска гамильтонова цикла в графе**

#include <iostream>

#include "Graph.h"

#include <Windows.h>

#include <vector>

using namespace std;

vector <bool> visited; //массив для помечания пройденных вершин

vector <int> path; //массив, хранящий гамильтонов путь

//в функцию передаётся начальная вершина cur и сам граф graph

bool FindGamilton(Graph graph, int cur)

{

//считывание матрицы смежности

vector <vector <int>> matrix = graph.MakeMatrixFromList();

//добавление начальной вершины в путь

path.push\_back(cur);

if (path.size() == graph.nodeCount) //если пройдены все вершины

{

if (matrix[path[0]][path.back()] == 1) //если начало и конец связаны

{

path.push\_back(path[0]); //замкнуть цикл

return true; //гамильтонов цикл найден, вернуть true

}

else

{

path.pop\_back(); //удалить вершину из пути

return false; //гамильтонов цикл не существует

}

}

visited[cur] = true; //если все вершины ещё не пройдены, двигаться далее

for (int i = 0; i < graph.nodeCount; i++) //проход по матрице смежности

{

//есть ребро между текущей вершиной и следующей

// при этом следующая не посещена

if (matrix[cur][i] == 1 && !visited[i])

if (FindGamilton(graph, i)) //запускаем поиск из следующей вершины

return true;

}

visited[cur] = false; //выход из рекурсии, если цикл не найден

path.pop\_back();

return false;

}

int main()

{

setlocale(LC\_ALL, "RUSSIAN");

Graph graph;

visited.resize(graph.nodeCount, false); //отметить все вершины непройденными

path.clear(); //очистить путь

graph.ReadMatrix(); //считать матрицу смежности

if (FindGamilton(graph, 0)) //если гамильтонов цикл существует

{

cout << "\n\n\t\t\t\tГамильтонов цикл найден: ";

int i = 0;

for (auto to : path) //вывод цикла

{

cout << to + 1;

if (i < path.size() - 1) cout << " -> ";

i++;

} cout << "\n\n";

}

return 0;

}

**Листинг 3.1 – Программа для решения задачи коммивояжёра методом ближайшего соседа**

#include <iostream>

#include "Graph.h"

#include <Windows.h>

#include <vector>

#include <fstream>

using namespace std;

int weight = 0; //поле для хранения веса маршрута

vector <bool> visited; //массив для помечания пройденных вершин

vector <int> path; //массив, хранящий найденный путь

int FindGamiltonWithWeight(Graph graph, int cur)

{

vector <vector <int>> matrix = graph.MakeMatrixFromList(); //матрица смежности

path.push\_back(cur); //пометили начальную вершину

if (path.size() == graph.nodeCount) //если пройдены все вершины

{

if (matrix[path[0]][path.back()] != 0) //если цикл замкнут

{

weight += matrix[path[0]][path.back()]; //внести вес последнего ребра

path.push\_back(path[0]); //пометить последнюю вершину

return weight; //вернуть вес найденного пути

}

else { path.pop\_back(); //отметить вершину непройденной

return -1;

}

}

visited[cur] = true; //отметить вершину пройденной

for (int i = 0; i < graph.nodeCount; i++) //проход по матрице смежности

{

//если есть ребро и следующая вершина не была пройдена

if (matrix[cur][i] != 0 && !visited[i])

{

//внести вес новой пройденный вершины в общий вес пути

weight += matrix[cur][i];

//запуск алгоритма из новой вершины

if (FindGamiltonWithWeight(graph, i) != -1)

return weight;

}

}

visited[cur] = false; //отметить вершину непройденной

path.pop\_back(); //исключить её из пути

return -1;

}

int main()

{

setlocale(LC\_ALL, "RUSSIAN");

Graph graph;

graph.ReadMatrix();

int minWeight = INT\_MAX; //поле для хранения мин. веса пути

vector <int> bestPath; //массив для записи наилучшего пути

int k = 0;

while (k < graph.nodeCount) //пока не перебраны все варианты

{

visited.clear(); //отметить все вершины не пройденными

visited.resize(graph.nodeCount, false);

path.clear(); //очистить путь

weight = 0;

//если существует гамильтонов цикл

if (FindGamiltonWithWeight(graph, k) != -1)

{

if (weight < minWeight) //если вес является минимальным

{

minWeight = weight; //зафиксировать найденный вес пути

bestPath = path; //зафиксировать найденный путь

}

}

k++;

}

cout << "\n\n\t\t\t\tГамильтонов цикл найден. Его вес: " << minWeight << "\n";

int i = 0;

cout << "\n\n\t\t\t\tМаршрут: ";

for (auto to : bestPath)

{

cout << to + 1;

if (i < bestPath.size() - 1) cout << " -> ";

i++;

} cout << "\n\n";   
return 0;}

Программная реализация метода ветвей и границ представлена в листинге 3.2.

#include <iostream>

#include "Graph.h"

using namespace std;

int minWeight = INT\_MAX;

int lowerBound = 0;

vector <int> bestPath;

vector <int> currentPath;

vector <bool> visited;

void FinalPath(vector <int> currentPath) //запись конечного маршрута

{

bestPath.clear();

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

bestPath.push\_back(currentPath[i]);

bestPath.push\_back(currentPath[0]);

return;

}

int FirstMin(vector <vector <int>> matrix, int k) //поиск первого подходящего ребра,

{ //ведущего в вершину k

int min = INT\_MAX;

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

if (matrix[k][i] < min && k != i)

min = matrix[k][i];

return min;

}

int SecondMin(vector <vector <int>> matrix, int k) //поиск второго подходящего ребра

{ //ведущего в вершину k

int first = INT\_MAX, second = INT\_MAX;

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

{

if (k!= i && matrix[k][i] <= first)

{

second = first;

first = matrix[k][i];

}

if (k != i && matrix[k][i] <= second

&& matrix[k][i] != first)

second = matrix[k][i];

}

return second;

}

int Solver(vector <vector <int>> matrix, //функция для поиска пути

int currentBound, int currentWeight,   
 int level, vector <int> currentPath)

{

if (level == matrix.size()) //текущий уровень дерева решений

{ //достиг ранга матрицы смежности

if (matrix[currentPath[level - 1]][currentPath[0]] != 0)

{

int currentRes = currentWeight +

matrix[currentPath[level - 1]][currentPath[0]];

if (currentRes < minWeight)

{

FinalPath(currentPath);

minWeight = currentRes;

}

}

return minWeight;

}

for (int i = 0; i < matrix.size(); i++) //ветвление не закончено

{

if (matrix[currentPath[level - 1]][i] != 0 //вершина не посещена

&& visited[i] == false)

{

int temp = currentBound; //запомнить текущую границу

currentWeight += matrix[currentPath[level - 1]][i];

//вычисление текущего веса пути

if (level == 1)

currentBound -= ((FirstMin(matrix, currentPath[level - 1]) +

FirstMin(matrix, i)) / 2);

else

currentBound -= ((SecondMin(matrix, currentPath[level - 1]) +

FirstMin(matrix, i)) / 2);

if (currentBound + currentWeight  
< minWeight) //если текущее ветвление выгодно

{ //продолжить путь из него

currentPath[level] = i;

visited[i] = true;

Solver(matrix, currentBound, currentWeight, level + 1,

currentPath);

}

currentWeight -= matrix[currentPath[level - 1]][i];

currentBound = temp;

visited.clear();

visited.resize(matrix.size(), false); //вернуться назад

for (int j = 0; j <= level - 1; j++)

visited[currentPath[j]] = true;

}

}

}

void BranchAndBoundSolver(vector <vector <int>> matrix)

{

minWeight = INT\_MAX; //сброс минимального веса пути

lowerBound = 0; //обнуление нижней границы

currentPath.resize(matrix[0].size(), -1); //массив для хранения текущего пути

visited.resize(matrix[0].size(), false); //массив для отметки пройденных вершин

int min;

for (int i = 0; i < matrix.size(); i++) //вычисление нижней границы

{

min = INT\_MAX;

for (int j = 0; j < matrix[i].size(); j++)

{

if (matrix[i][j] < min && matrix[i][j] != 0) min = matrix[i][j];

}

lowerBound += min;

}

visited[0] = true;

currentPath[0] = 0;

Solver(matrix, lowerBound, 0, 1, currentPath); //рекурсивный поиск пути

return;

}

int main()

{

setlocale(LC\_ALL, "RUSSIAN");

Graph graph;

graph.ReadMatrix();

vector <vector <int>> matrix = graph.MakeMatrixFromList();

BranchAndBoundSolver(matrix);

printf("Минимальная длина пути: %d\n", minWeight);

printf("Путь: ");

for (int i = 0; i < graph.nodeCount; i++)

printf("%d ", bestPath[i]);   
return 0;  
}

Программная реализация муравьиного алгоритма представлена в листинге 3.3.

**Листинг 3.3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом**

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <ctime>

#include <time.h>

#include <iomanip>

#include <vector>

#include "Graph.h"

using namespace std;

#define N 5 //количество городов

#define RUNS 50 //число поколений муравьёв

#define alpha 1 //расчётная константа

#define beta 5 //расчётная константа

#define Q 1 //расчётная константа

float minW = FLT\_MAX; //поле памяти, хранящее

//минимальный вес пути

vector <vector <float>> path;

vector <vector <float>> feromon; //массив значений феромонов

vector <float> temp;

**Листинг 3.3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)**

vector <int> instance; //текущий путь

vector <bool> visited;

vector <double> weights;

vector <vector <int>> paths; //все пути

vector <int> bestPath; //наилучший путь

int Search(vector <float> temp, float r)

{

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

{

if (temp[i] >= r)

return i;

}

return 1;

}

vector <vector <float>> UpdateFeromon(vector <vector <float>> f, vector <int> instance)

{

for (int k = 0; k < N; k++)

{

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

{

f[k][i] \*= 0.95;

}

}

for (int i = 0; i < N - 1; i++)

{

int a = instance[i];

int b = instance[i + 1];

f[a][b] += 0.2;

}

f[instance[N - 1]][instance[0]] += 0.2;

return f;

}

float RoadLength(vector <vector <float>> path, vector <int> currentPath)

{

float weight = 0;

for (int i = 0; i < N - 1; i++)

{

weight += path[currentPath[i]][currentPath[i + 1]];

}

weight += path[currentPath[N - 1]][currentPath[0]];

return weight;

}

vector <vector <float>> ReadMatrix(vector <vector <float>> path)

{

ifstream input("input.txt");

for (size\_t i = 0; i < N; i++) //чтение матрицы смежности

{

for (size\_t j = 0; j < N; j++)

{

input >> path[i][j];

}

}

**Листинг 3.3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)**

input.close();

return path;

}

void MakeFeromonMatrix(vector <vector <float>> feromonMatrix)

{

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

{

for (int j = 0; j < N; j++) //чтение матрицы феромонов

{

if (i == j)

feromonMatrix[i][j] = 0.0;

}

}

return;

}

void AntSolverRun()

{

path.clear(); //обновление данных

path.resize(N, vector<float>(N, FLT\_MAX));

feromon.clear();

feromon.resize(N, vector<float>(N, 1));

MakeFeromonMatrix(feromon);

path = ReadMatrix(path);

int sum;

temp.clear(); temp.resize(N, 0);

instance.clear(); instance.resize(N, 0);

visited.clear(); visited.resize(N, false);

int R = rand() % N; //выбор первой вершины

instance[0] = R;

visited[R] = true;

for (int k = 1; k < N; k++)

{

sum = 0;

for (int i = 0; i < N; i++) //расчет феромоновых путей

{

temp[i] = path[R][i] \* feromon[R][i];

sum += path[R][i];

}

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

{

temp[i] /= sum;

if (i > 0)

temp[i] += temp[i - 1];

}

while (true) //проход по всем вершинам

{

float r = ((float)rand() / (RAND\_MAX));

int next = Search(temp, r);

if (!visited[next]) //если вершина не посещена

{

visited[next] = true;

**Листинг 3.3 – Программа для решения задачи коммивояжёра муравьиным алгоритмом (продолжение)**

instance[k] = next;

break;

}

}

}

feromon = UpdateFeromon(feromon, instance); //обновление феромоновых

//следов

float weight = RoadLength(path, instance); //расчёт веса текущего пути

if (weight < minW) //отбор наикратчайшего пути

{

minW = weight;

bestPath = instance;

}

return;

}

int main()

{

int i = 0;

do {

AntSolverRun();

i++;

} while (i < RUNS);

for (auto to : bestPath)

cout << to + 1 << " --> ";

cout << bestPath[0]+1;

cout << "\nweight: " << minW;

return 0;

}