Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации

Бурятский институт инфокоммуникаций (филиал) Федерального государственного бюджетного образовательного учреждения высшего образования

«Сибирский государственный университет телекоммуникаций и информатики» в г. Улан-Удэ

(БИИК СибГУТИ)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | |
| Кафедра | Информатики и вычислительной техники | |
| Допустить к защите | |  |
|  | | |
| Зав.каф. \_\_\_\_\_\_\_\_\_\_\_\_И.Б. Елтунова | | |
|  | |  |

|  |
| --- |
| **РАСЧЕТНО-ГРАФИЧЕСКАЯ РАБОТА** |
| По дисциплине: «Современные технологии программирования» |
| Разработка мобильной игры “Арканоид” на Java |

Пояснительная записка

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Студент |  | | | | /Дондупова О.Б./ | | |
|  |  | | | |  | | |
| Факультет | ИВТ | | Группа | | | И-101 |  |
| Руководитель |  | | | /Извеков Я.О. / | | | |
|  |  |  | |  | | | |
|  | |  | |  | | | |

Улан-Удэ

2024 г.

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_Toc163308319)

[ВВЕДЕНИЕ 3](#_Toc163308320)

[1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 5](#_Toc163308321)

[1.1. Алгоритм полного перебора 5](#_Toc163308322)

[1.2. Жадный алгоритм 6](#_Toc163308323)

[1.3. Метод ветвей и границ 6](#_Toc163308324)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 8](#_Toc163308325)

[ЗАКЛЮЧЕНИЕ 15](#_Toc163308326)

## ВВЕДЕНИЕ

Целью данной расчётно-графической работы является разработка игры «Арканоид» под Android и демонстрация функционала языка программирования Java.

«Арканоид» - аркадная игра является собирательным названием для класса подобных игр, история которого насчитывает более 50 лет.

Суть игры заключается в следующем: игрок направляет небольшую горизонтально двигающуюся платформу, чтобы не допустить касания мяча нижней стороны экрана. Цель игры заключается в том, чтобы выбить все блоки, которые находятся на противоположной стороне поля с минимально возможной потерей жизней. Реализации игры отличаются друг от друга. Некоторые игры имеют различные уровни сложностей, бонусы, выпадающие из разрушенного блока, также могут присутствовать блоки, разных уровней разрушения. Наличие или отсутствие какой-либо характеристики определяется только желанием разработчика.

В данной реализации игры будет присутствовать механизм засекания времени игры, а также укрепленные блоки, которые разрушаются только со второго раза.

## ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

# Алгоритм полного перебора

Метод полного перебора является простым, логичным и широко используемым математическим методом. Он применим во многих, если не во всех, областях математики: задача коммивояжера также не является исключением. Принцип полного перебора заключается в названии: перебираются все возможные решения и из них выбирается решение, удовлетворяющее условиям задачи. Огромным преимуществом метода полного перебора перед другими методами решения задачи коммивояжера является гарантированность нахождения наилучшего маршрута. Другие методы советуют лишь «хороший» маршрут, который совсем не обязательно является лучшим.

К достоинствам метода также относится простота его программной реализации. Однако, в связи с наличием огромного недостатка, метод полного перебора крайне редко используется на практике. Этим недостатком является временная сложность алгоритма. Асимметричная задача коммивояжера с n посещаемых пунктов требует при полном переборе рассмотрения (n-1)! туров, а факториал растет невероятно быстро. Поэтому метод полного перебора может применяться только для задач малой размерности (при рассмотрении до двух десятков посещаемых пунктов).

# Жадный алгоритм

Жадный алгоритм (англ. Greedy algorithm) — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным. Решение принимаемое на каждом шаге должно быть оптимальным только на текущем шаге и должно приниматься без учета предыдущих или последующих решений. Известно, что если структура задачи задается матроидом, тогда применение жадного алгоритма выдаст глобальный оптимум.

Если глобальная оптимальность алгоритма имеет место практически всегда, его обычно предпочитают другим методам оптимизации, таким как динамическое программирование.

Принцип жадного алгоритма заключается в последовательном выборе самого привлекательного варианта в данном случае и в данных условиях задачи. Так, в задаче коммивояжера, жадный алгоритм каждый раз будет выбирать город, расстояние до которого является кратчайшим. В конечном случае, такой недальновидный выбор может плохо сказаться на решении задачи.

# Метод ветвей и границ

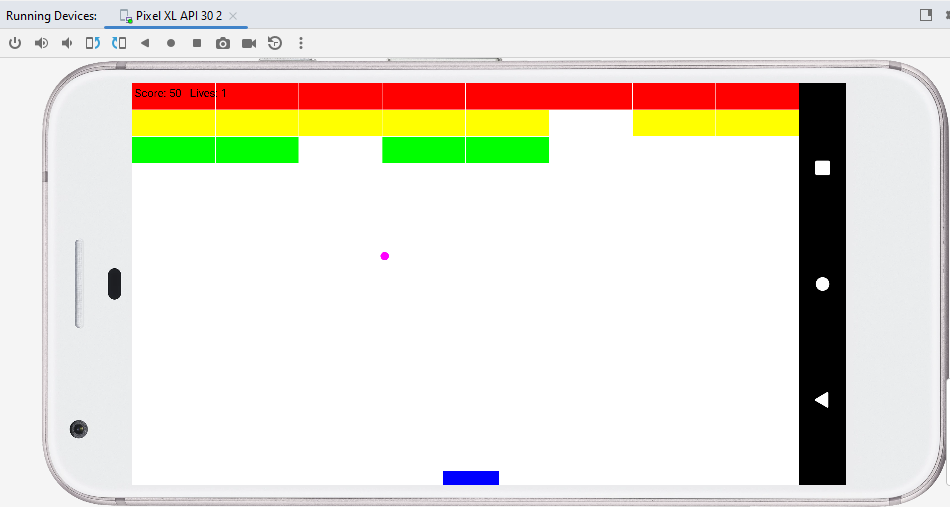
Метод ветвей и границ - общий алгоритмический метод для нахождения оптимальных решений различных задач оптимизации, особенно дискретной и комбинаторной оптимизации. Метод является развитием метода полного перебора, однако в отличие от последнего — отсеивает подмножества допустимых решений, заведомо не содержащих оптимальных решений.

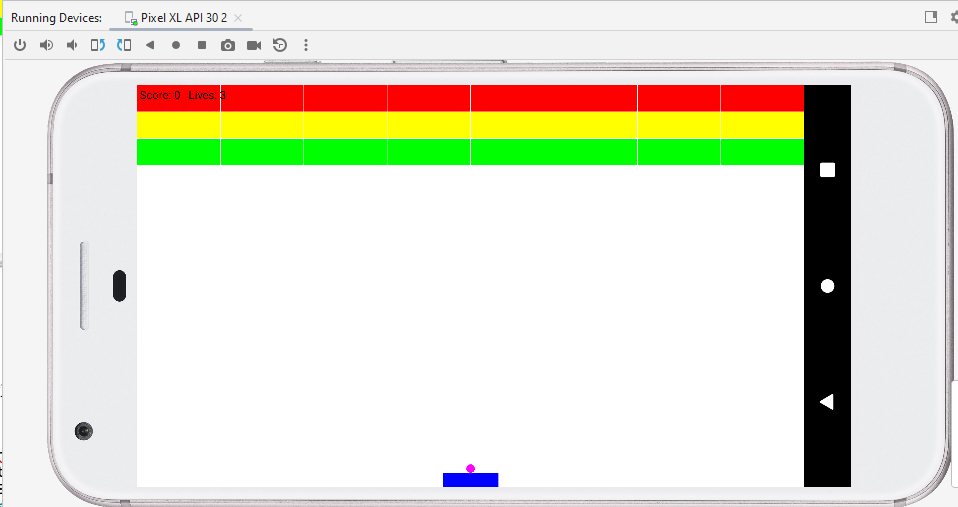
Общая идея метода может быть описана на примере поиска минимума функции на множестве допустимых значений переменной . Функция и переменная могут быть произвольной природы. Для метода ветвей и границ необходимы две процедуры: ветвление и нахождение оценок (границ).

Подход получил название метод ветвей и границ (англ. branch and bound) — как общий алгоритмический метод оптимизации. МВГ связывают с деревом поиска оптимального (Торt) решения, которое строится в процессе обработки исходных данных задачи. Отсюда названия корень, которому в дереве приписывают все возможные в задаче, решения-ветви, соединяющие узлы дерева, Использование понятия границ и их расчет стимулирует или тормозит рост ветвей в таком дереве. Важную роль играет процедура разбиения на узлы области допустимых решений (ОДР) исходной задачи, т.е. на меньшие непересекающиеся подмножества и их оценивание. Другая процедура, названная процедурой ветвления, реализует разбиение на множества допустимых значений переменной х на подобласти меньших размеров. Еще один важный элемент МВГ - процедура вычисления оценок, которая состоит в поиске значений границ ЦФ для решения задачи. Вычисление нижней границы ЦФ (НГЦФ) является важнейшим, ключевым элементом предложенной схемы. Таким образом, в основе метода ветвей и границ лежит идея последовательного разбиения множества допустимых решений на подмножества (стратегия “разделяй и властвуй”) и оценивания получаемых при разбиении частей. Каждый шаг алгоритма разбиения сопровождается проверкой условия того, содержит ли конкретное подмножество оптимальное решение или нет.

# ПРАКТИЧЕСКАЯ ЧАСТЬ

Одной из основных частей решения задачи является рандомная генерация графа с заданным количеством вершин. Чтобы гарантировать «реальность» такого графа, вершинам графа заданы координаты двумерной плоскости, откуда выводится матрица расстояний.





Алгоритм полного перебора

double min\_length = INT\_MAX, current\_length = 0;

vector<int> min\_Path = { 0 };

vector<int> Path;

double step = 0;

cout << endl << endl << "Алгоритм полного перебора: " << endl;

clock\_t start\_time\_1 = clock();

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

Path.insert(Path.end(), i);

}

Path.insert(Path.end(), 0);

while (next\_permutation(Path.begin() + 1, Path.end() - 1)) {

current\_length = 0;

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

step = rast[Path[i]][Path[i + 1]];

if (step == INT\_MAX || current\_length == INT\_MAX) {

current\_length = INT\_MAX;

}

else {

current\_length += step;

}

}

if (current\_length < min\_length) {

min\_length = current\_length;

min\_Path = Path;

}

}

cout << endl << "Путь: " << min\_Path[0] + 1;

for (int i = 1; i <= N; i++) {

cout << " --> " << min\_Path[i] + 1; //вывод всего маршрута от начальной точки и возвращение к исходной

}

cout << endl;

cout << "Длина пути: " << int(min\_length) << endl; //кратчайшее расстояние

cout << "Затраченное время: " << (static\_cast<double>(clock() - start\_time\_1) / CLOCKS\_PER\_SEC); //вывод затраченного времени на прохождение пути

Результаты тестов приведены в таблице ниже. Измерения проводились таким образом: каждый из вариантов количества элементов в массиве был запущен по пять раз, после чего высчитывалось среднее время выполнения на данное число элементов. График наглядно показывает главный недостаток алгоритма – малую скорость, из-за чего время быстро растет.

Жадный алгоритм

bool visit(int g) { //проверка на посещение города

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

{

if (visited[i] == g) return 1;

}

return 0;

}

void close(int bliz) { //ближ. город к текущeму.

int currentMin = INT\_MAX; //текущий минимальный путь

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

if (visit(i) == 0 && zhad\_rast[i][visited[bliz]] < currentMin) {

currentMin = zhad\_rast[i][visited[bliz]];

visited[bliz + 1] = i; //Если город не был посещен, а расстояние до текущего города является кратчайшим, город будет посещен и сохранен в массиве s[]

}

}

for (int i = 0; i < N; i++) { //наличие непосещенных городов

if (visited[i] == -1)

close(visited[bliz + 1]);

}

}

void zhad()

{

int sum = 0; // Сумма кратчайших путей

visited[0] = 0; //отсчет с 0 города (любое другое)

close(visited[0]); //город, ближайший к 2 гор.

cout << "Путь: " << visited[0] + 1 << " --> ";

sum += zhad\_rast[visited[0]][visited[1]];

for (int i = 1; i <= N; i++) {

if (i == N - 1) {

sum += zhad\_rast[visited[N - 1]][visited[0]];

cout << visited[i] + 1 << " --> ";

cout << visited[0] + 1;

cout << endl << "Длина пути: " << sum << endl; //вывод всего маршрута от начальной точки и возвращение к исходной

break;

}

cout << visited[i] + 1 << " --> ";

sum += zhad\_rast[visited[i]][visited[i + 1]];

}

}

В функции main():

cout << endl << endl << "Жадный алгоритм" << endl<<endl;

clock\_t start\_time\_2 = clock();

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

visited[i] = -1;

}

zhad();

cout << "Затраченное время: " << (static\_cast<double>(clock() - start\_time\_2) / CLOCKS\_PER\_SEC);

Жадный алгоритм показал себя лучше полного перебора в плане времени исполнения, однако по точности сильно уступает.

Метод ветвей и границ

// искомый путь

int final\_path[N + 1];

// проверка на посещенные города

bool visited\_bnb[N];

//Длина кратчайшего пути

int final\_res = INT\_MAX;

// Функция для определения минимальной граничной стоимости имеющий конец в вершине i

double firstMin(double adj[N][N], int i)

{

double min = INT\_MAX;

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

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

min = adj[i][k];

return min;

}

// функция для нахождения второй минимальной граничной стоимости имеющий конец в вершине i

double secondMin(double adj[N][N], int i)

{

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

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

{

if (i == j)

continue;

if (adj[i][j] <= first)

{

second = first;

first = adj[i][j];

}

else if (adj[i][j] <= second &&

adj[i][j] != first)

second = adj[i][j];

}

return second;

}

// Функция, принимающая в качестве аргументов:

// curr\_bound -> нижняя граница корневого узла

// curr\_weight-> сохраняет вес пути на данный момент

// уровень-> текущий уровень при перемещении в поиске дерева

// curr\_path[] -> где хранится решение, которое

// позже будет скопировано в final\_path[]

void TSPRec(double adj[N][N], double curr\_bound, double curr\_weight,

int level, int curr\_path[])

{

// базовый случай - это когда мы достигли уровня N, который означает, что мы охватили все узлы один раз

if (level == N)

{

// проверка, есть ли ребро из последней вершины в обратный путь к первой вершине

if (adj[curr\_path[level - 1]][curr\_path[0]] != 0)

{

// curr\_res имеет общий вес решения, которое мы получили

double curr\_res = curr\_weight +

adj[curr\_path[level - 1]][curr\_path[0]];

// Обновляем конечный результат и конечный путь, если текущий результат лучше.

if (curr\_res < final\_res)

{

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

final\_path[i] = curr\_path[i];

final\_path[N] = curr\_path[0];

final\_res = curr\_res;

}

}

return;

}

// для любого другого уровня выполняем итерацию по всем вершинам, чтобы

// рекурсивно построить дерево пространства поиска

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

{

// Рассмотрим следующую вершину, если она не совпадает (диагональная запись в матрице смежности и еще не посещена)

if (adj[curr\_path[level - 1]][i] != 0 &&

visited\_bnb[i] == false)

{

double temp = curr\_bound;

curr\_weight += adj[curr\_path[level - 1]][i];

// различное вычисление curr\_bound для уровеня 2 с других уровней

if (level == 1)

curr\_bound -= ((firstMin(adj, curr\_path[level - 1]) +

firstMin(adj, i)) / 2);

else

curr\_bound -= ((secondMin(adj, curr\_path[level - 1]) +

firstMin(adj, i)) / 2);

// curr\_bound + curr\_weight - фактическая нижняя граница для узла, на который мы прибыли

// Если текущая нижняя граница < final\_res, нам нужно исследовать узел далее

if (curr\_bound + curr\_weight < final\_res)

{

curr\_path[level] = i;

visited\_bnb[i] = true;

// Вызываем TSPRec для перехода на следующий уровень

TSPRec(adj, curr\_bound, curr\_weight, level + 1,

curr\_path);

}

// Иначе мы должны обрезать узел, сбросив все изменения в curr\_weight и curr\_bound

curr\_weight -= adj[curr\_path[level - 1]][i];

curr\_bound = temp;

// Также сбрасываем посещенный массив

memset(visited\_bnb, false, sizeof(visited\_bnb));

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

visited\_bnb[curr\_path[j]] = true;

}

}

}

// Эта функция устанавливает final\_path[]

void BnB(double adj[N][N])

{

int curr\_path[N + 1];

// Вычисляем начальную нижнюю границу для корневого узла

// используя формулу 1/2 \* (сумма первых минимальных + секунда в минуту) для всех ребер.

// Также инициализируем curr\_path и посещенный массив

int curr\_bound = 0;

memset(curr\_path, -1, sizeof(curr\_path));

memset(visited\_bnb, 0, sizeof(curr\_path));

// Вычисляем начальную границу

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

curr\_bound += (firstMin(adj, i) + secondMin(adj, i));

// Округление нижней границы до целого числа

curr\_bound = (curr\_bound & 1) ? curr\_bound / 2 + 1 :

curr\_bound / 2;

// Мы начинаем с вершины 1, так что первая вершина в curr\_path[] равна 0

visited\_bnb[0] = true;

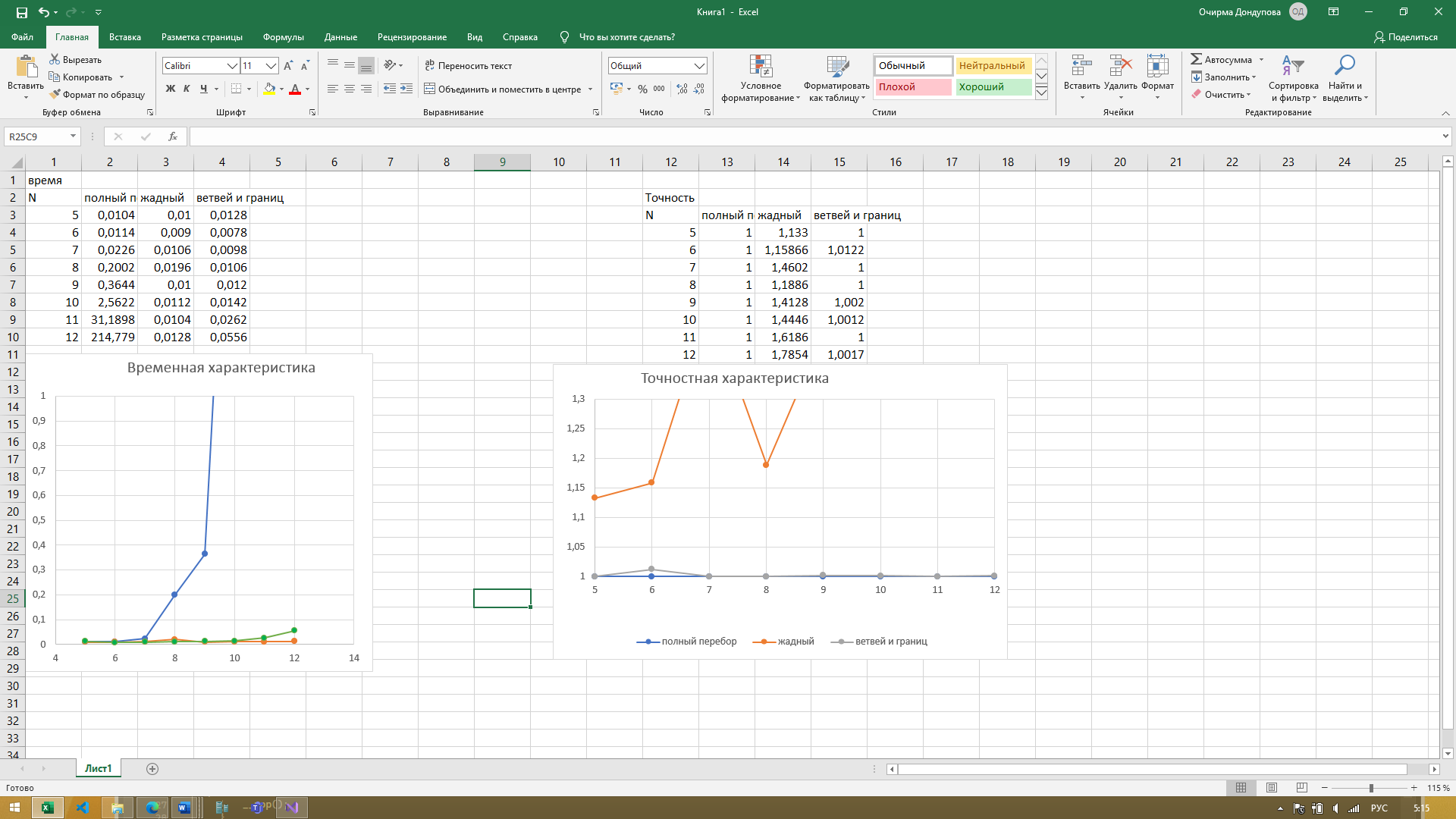
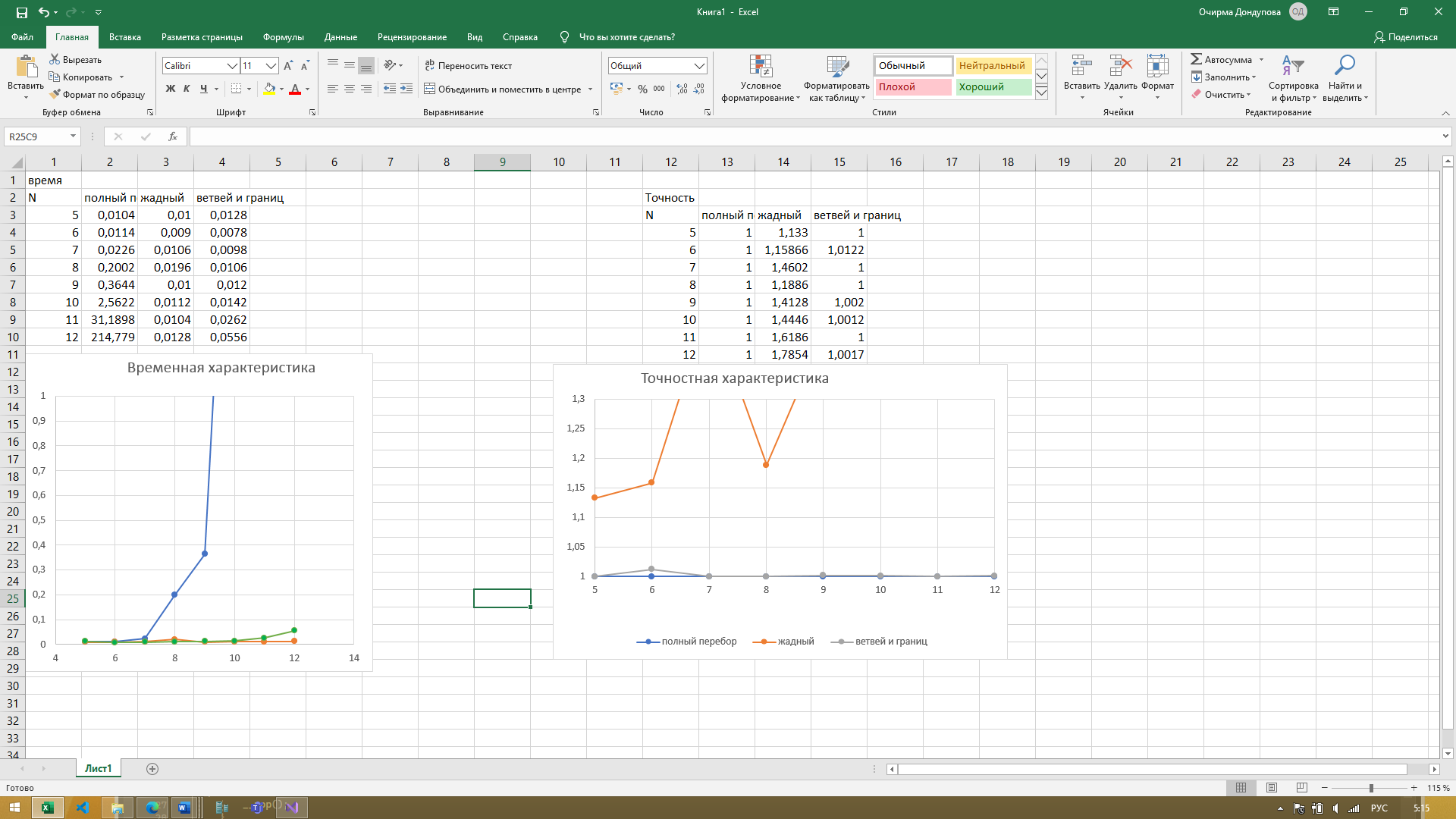
curr\_path[0] = 0;

// Вызов TSPRec для curr\_weight, равного 0 и уровню 1

TSPRec(adj, curr\_bound, 0, 1, curr\_path);

}

Результаты тестов приведены в таблице ниже. Метод ветвей и границ по времени выполнения оказался лучше полного перебора, но хуже жадного алгоритма.



# ЗАКЛЮЧЕНИЕ

В данной курсовой работе был проведено сравнение трех алгоритмов решения задачи коммивояжера, каждое из которых было реализовано на языке программирования С++. Был проведен анализ их эффективности с графиками зависимости времени выполнения алгоритмов от количества элементов. Самым эффективным по времени алгоритмом оказался жадный метод. Также был проведён анализ точности жадного алгоритма и метода ветвей и границ по сравнению с точным решением полного перебора. Метод ветвей и границ показывает почти 100% точность в отличие от жадного алгоритма, погрешность которого увеличивается с увеличением количества вершин.

#include <iostream>

#include <iomanip>

#include <math.h>

#include <time.h>

#include <vector>

#include <algorithm>

using namespace std;

const int N = 1;

int visited[N]; //города были посещены, -1 - не были посещены

double zhad\_rast[N][N];

// искомый путь

int final\_path[N + 1];

// проверка на посещенные города

bool visited\_bnb[N];

//Длина кратчайшего пути

int final\_res = INT\_MAX;

// Функция для копирования временного решения в окончательное

//void copyToFinal(int curr\_path[])

//{

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

// final\_path[i] = curr\_path[i];

// final\_path[N] = curr\_path[0];

//}

// Функция для определения минимальной граничной стоимости имеющий конец в вершине i

double firstMin(double adj[N][N], int i)

{

double min = INT\_MAX;

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

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

min = adj[i][k];

return min;

}

// функция для нахождения второй минимальной граничной стоимости имеющий конец в вершине i

double secondMin(double adj[N][N], int i)

{

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

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

{

if (i == j)

continue;

if (adj[i][j] <= first)

{

second = first;

first = adj[i][j];

}

else if (adj[i][j] <= second &&

adj[i][j] != first)

second = adj[i][j];

}

return second;

}

// Функция, принимающая в качестве аргументов:

// curr\_bound -> нижняя граница корневого узла

// curr\_weight-> сохраняет вес пути на данный момент

// уровень-> текущий уровень при перемещении в поиске дерева

// curr\_path[] -> где хранится решение, которое

// позже будет скопировано в final\_path[]

void TSPRec(double adj[N][N], double curr\_bound, double curr\_weight,

int level, int curr\_path[])

{

// базовый случай - это когда мы достигли уровня N, который означает, что мы охватили все узлы один раз

if (level == N)

{

// проверка, есть ли ребро из последней вершины в обратный путь к первой вершине

if (adj[curr\_path[level - 1]][curr\_path[0]] != 0)

{

// curr\_res имеет общий вес решения, которое мы получили

double curr\_res = curr\_weight +

adj[curr\_path[level - 1]][curr\_path[0]];

// Обновляем конечный результат и конечный путь, если текущий результат лучше.

if (curr\_res < final\_res)

{

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

final\_path[i] = curr\_path[i];

final\_path[N] = curr\_path[0];

final\_res = curr\_res;

}

}

return;

}

// для любого другого уровня выполняем итерацию по всем вершинам, чтобы

// рекурсивно построить дерево пространства поиска

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

{

// Рассмотрим следующую вершину, если она не совпадает (диагональная запись в матрице смежности и еще не посещена)

if (adj[curr\_path[level - 1]][i] != 0 &&

visited\_bnb[i] == false)

{

double temp = curr\_bound;

curr\_weight += adj[curr\_path[level - 1]][i];

// различное вычисление curr\_bound для уровеня 2 с других уровней

if (level == 1)

curr\_bound -= ((firstMin(adj, curr\_path[level - 1]) +

firstMin(adj, i)) / 2);

else

curr\_bound -= ((secondMin(adj, curr\_path[level - 1]) +

firstMin(adj, i)) / 2);

// curr\_bound + curr\_weight - фактическая нижняя граница для узла, на который мы прибыли

// Если текущая нижняя граница < final\_res, нам нужно исследовать узел далее

if (curr\_bound + curr\_weight < final\_res)

{

curr\_path[level] = i;

visited\_bnb[i] = true;

// Вызываем TSPRec для перехода на следующий уровень

TSPRec(adj, curr\_bound, curr\_weight, level + 1,

curr\_path);

}

// Иначе мы должны обрезать узел, сбросив все изменения в curr\_weight и curr\_bound

curr\_weight -= adj[curr\_path[level - 1]][i];

curr\_bound = temp;

// Также сбрасываем посещенный массив

memset(visited\_bnb, false, sizeof(visited\_bnb));

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

visited\_bnb[curr\_path[j]] = true;

}

}

}

// Эта функция устанавливает final\_path[]

void BnB(double adj[N][N])

{

int curr\_path[N + 1];

// Вычисляем начальную нижнюю границу для корневого узла

// используя формулу 1/2 \* (сумма первых минимальных + секунда в минуту) для всех ребер.

// Также инициализируем curr\_path и посещенный массив

int curr\_bound = 0;

memset(curr\_path, -1, sizeof(curr\_path));

memset(visited\_bnb, 0, sizeof(curr\_path));

// Вычисляем начальную границу

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

curr\_bound += (firstMin(adj, i) + secondMin(adj, i));

// Округление нижней границы до целого числа

curr\_bound = (curr\_bound & 1) ? curr\_bound / 2 + 1 :

curr\_bound / 2;

// Мы начинаем с вершины 1, так что первая вершина в curr\_path[] равна 0

visited\_bnb[0] = true;

curr\_path[0] = 0;

// Вызов TSPRec для curr\_weight, равного 0 и уровню 1

TSPRec(adj, curr\_bound, 0, 1, curr\_path);

}

bool visit(int g) { //проверка на посещение города

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

{

if (visited[i] == g) return 1;

}

return 0;

}

void close(int bliz) { //ближ. город к текущeму.

int currentMin = INT\_MAX; //текущий минимальный путь

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

if (visit(i) == 0 && zhad\_rast[i][visited[bliz]] < currentMin) {

currentMin = zhad\_rast[i][visited[bliz]];

visited[bliz + 1] = i; //Если город не был посещен, а расстояние до текущего города является кратчайшим, город будет посещен и сохранен в массиве s[]

}

}

for (int i = 0; i < N; i++) { //наличие непосещенных городов

if (visited[i] == -1)

close(visited[bliz + 1]);

}

}

void zhad()

{

int sum = 0; // Сумма кратчайших путей

visited[0] = 0; //отсчет с 0 города (любое другое)

close(visited[0]); //город, ближайший к 2 гор.

cout << "Путь: " << visited[0] + 1 << " --> ";

sum += zhad\_rast[visited[0]][visited[1]];

for (int i = 1; i <= N; i++) {

if (i == N - 1) {

sum += zhad\_rast[visited[N - 1]][visited[0]];

cout << visited[i] + 1 << " --> ";

cout << visited[0] + 1;

cout << endl << "Длина пути: " << sum << endl; //вывод всего маршрута от начальной точки и возвращение к исходной

break;

}

cout << visited[i] + 1 << " --> ";

sum += zhad\_rast[visited[i]][visited[i + 1]];

}

}

int main()

{

setlocale(LC\_ALL, "RUS");

int XY[N][2]; //массив координат

double rast[N][N]; //массив расстояний между городами

int i, j;

//Координаты городов

srand((unsigned)time(NULL));

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

XY[i][0] = rand() % 1000;

XY[i][1] = rand() % 700;

}

//Вычисление расстояний между городами

cout << endl << "Таблица расстояний:" << endl;

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

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

rast[i][j] = sqrt(pow(abs(XY[j][0] - XY[i][0]), 2) + pow(abs(XY[j][1] - XY[i][1]), 2));

::zhad\_rast[i][j] = rast[i][j];

//cout << setw(6) << setprecision(6) << rast[i][j] << " ";

}

//cout << endl << endl;

}

//ПОЛНЫЙ ПЕРЕБОР

double min\_length = INT\_MAX, current\_length = 0;

vector<int> min\_Path = { 0 };

vector<int> Path;

double step = 0;

cout << endl << endl << "Алгоритм полного перебора: " << endl;

clock\_t start\_time\_1 = clock();

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

Path.insert(Path.end(), i);

}

Path.insert(Path.end(), 0);

while (next\_permutation(Path.begin() + 1, Path.end() - 1)) {

current\_length = 0;

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

step = rast[Path[i]][Path[i + 1]];

if (step == INT\_MAX || current\_length == INT\_MAX) {

current\_length = INT\_MAX;

}

else {

current\_length += step;

}

}

if (current\_length < min\_length) {

min\_length = current\_length;

min\_Path = Path;

}

}

cout << endl << "Путь: " << min\_Path[0] + 1;

for (int i = 1; i <= N; i++) {

cout << " --> " << min\_Path[i] + 1; //вывод всего маршрута от начальной точки и возвращение к исходной

}

cout << endl;

cout << "Длина пути: " << int(min\_length) << endl; //кратчайшее расстояние

cout << "Затраченное время: " << (static\_cast<double>(clock() - start\_time\_1) / CLOCKS\_PER\_SEC); //вывод затраченного времени на прохождение пути

cout << endl << endl << "Жадный алгоритм" << endl<<endl;

clock\_t start\_time\_2 = clock();

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

visited[i] = -1;

}

zhad();

cout << "Затраченное время: " << (static\_cast<double>(clock() - start\_time\_2) / CLOCKS\_PER\_SEC);

cout << endl << endl << "Алгоритм ветвей и границ: " << endl;

clock\_t start\_time\_3 = clock();

BnB(rast);

cout << endl << "Путь: " << final\_path[0] + 1;

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

{

cout << " --> " << final\_path[i] + 1;

}

cout << endl << "Длина пути: " << final\_res << endl;

cout << "Затраченное время: " << (static\_cast<double>(clock() - start\_time\_3) / CLOCKS\_PER\_SEC);

return 0;

}