**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 2 з дисципліни

«Проектування алгоритмів»

**«Неінформативний, інформативний та локальний пошук»**

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-41 Хижняк Анна*

Київ 2025

Зміст

[1 Мета лабораторної роботи 3](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070688)

[2 ЗаВдання 4](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070689)

[3 Виконання 8](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070690)

[3.1 Псевдокод алгоритмів 8](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070691)

[3.2 Програмна реалізація 8](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070692)

[3.2.1 Вихідний код 8](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070693)

[3.2.2 Приклади роботи 8](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070694)

[3.3 Дослідження алгоритмів 8](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070695)

[Висновок 11](file:///C:\Users\Anne\Downloads\Telegram%20Desktop\lr2_2025.ukr.docx#_Toc81070696)

# Мета лабораторної роботи

Мета роботи – розглянути та дослідити алгоритми неінформативного, інформативного та локального пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

# Завдання

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту за допомогою алгоритму локального пошуку **АЛП та бектрекінгу,** що використовує задану евристичну функцію Func. Розробити власну евристичну функцію та порівняти її із заданою за варіантом.

Програму реалізувати на довільній мові програмування.

**Увага!** Алгоритм неінформативного пошуку **АНП,** реалізовується за принципом «AS IS», тобто так, як є, без додаткових модифікацій (таких як перевірка циклів, наприклад).

Провести серію експериментів для вивчення ефективності роботи алгоритмів. Кожний експеримент повинен відрізнятись початковим станом. Серія повинна містити не менше 20 експериментів для кожного алгоритму. Початковий стан зафіксувати у таблиці експериментів. За проведеними серіями необхідно визначити:

* середню кількість етапів (кроків), які знадобилось для досягнення розв’язку (ітерації);
* середню кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе;
* середню кількість згенерованих станів під час пошуку;
* середню кількість станів, що зберігаються в пам’яті під час роботи програми.

Передбачити можливість обмеження виконання програми за часом (30 хвилин) та використання пам’яті (1 Гб).

**Використані позначення:**

* **TSP** – Задача комівояжера. Необхідно обійти усі вершини графа на 100 вершин і повернутись у початкову. Вагу ребер задати випадково у діапазоні від 10 до 30. Довжина знайденого шляху повинна бути не більше 2000.
* **BCTR** – алгоритм пошуку із поверненнями (backtracking).
* **ANNEAL** – Локальний пошук із симуляцією відпалу. Робоча характеристика – залежність температури Т від часу роботи алгоритму t. Можна розглядати лінійну залежність: T = 1000 - k∙t, де k – змінний коефіцієнт.
* **N1** – Евристика найближчого сусіда.

**Завдання за варіантом**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Задача** | **АНП** | **АІП** | **АЛП** | **Func** |
| 25 | TSP | BCTR |  | ANNEAL | N1 |

# Виконання

## Псевдокод алгоритмів

**початок**(SimulatedAnnealing(List<int> route))

current := route

**поки** T > 10^-3 && GetValue(current) > limit

**повторити**

Т := Schedule(t)

next := GetSuccessor(current)

deltaE := GetValue(current) – GetValue(next)

**якщо** (deltaE > 0 || exp(deltaE/T) > random(0,1))

**то**

current := next

**все якщо**

**все повторити**

**кінець**

**початок**(Backtracking(int current, int cost, List<int> route)

**якщо**(found)

**то**

**повернути**

**все якщо**

**якщо**(AllCitiesVisited())

**то**

finalCost = cost + cost[current][0]

**якщо**(finalCost < limit)

**то**

totalCost := finalCost

finalRoute := route

found := true

**все якщо**

**повернути**

**все якщо**

sortedNeighbors := SortNeighbors(current)

**для** і **від** 0 **до** sortedNeighbors.length

**повторити**

**якщо**(!visited[sortedNeighbors[i]])

**то**

newCost := cost + cost[current][sortedNeighbors[i]];

**якщо**(newCost >= limit)

**то**

**перервати**

**все якщо**

visited[sortedNeighbors[i]] := true

route.add(sortedNeighbors[i]);

Backtracking(sortedNeighbors[i], newCost, route)

route.remove(route.size() - 1)

visited[sortedNeighbors[i]] := false

**все якщо**

**все повторити**

**кінець**

## Програмна реалізація

### Вихідний код

import java.io.File;

public class Main {

public static void main(String[] args) {

File input = new File("E:\\FICE\\3\_sem\\AD\\lab2\\input.txt");

GraphGenerator g = new GraphGenerator();

g.Write(input);

System.out.println("Started Anneal:");

AnnealTSP solver1 = new AnnealTSP(g.edges, 1050);

solver1.Solve();

System.out.println();

System.out.println();

System.out.println("Started BCTR:");

BCTR solver2 = new BCTR(g.edges, 1080);

solver2.Solve();

}

}

import java.io.File;

import java.io.FileWriter;

import java.io.PrintWriter;

import java.util.Random;

public class GraphGenerator {

private static int vertexes = 100;

public static int[][] edges = new int[vertexes][vertexes];

private static Random r = new Random();

public static void Write(File input){

try (

PrintWriter w = new PrintWriter(new FileWriter(input));

){

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

w.print(" " + (i + 1));

}

w.print("\n");

Generate(edges);

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

StringBuilder sb = new StringBuilder();

w.print((i + 1) + " ");

for(int j = 0; j < vertexes; j++){

sb.append(edges[i][j]);

sb.append(" ");

}

String row = sb.toString();

w.println(row);

}

} catch (Exception e){

throw new RuntimeException(e);

};

}

private static void Generate(int[][] edges){

for(int i = 0; i < edges.length; i++){

for(int j = 0; j < edges.length; j++){

if(edges[i][j] == 0) {

if (i == j) continue;

edges[i][j] = edges [j][i] = r.nextInt(21) + 10;

}

}

}

}

}

import java.util.List;

import java.util.ArrayList;

import java.util.Random;

public class AnnealTSP {

private int[][] edges;

private int n;

private int limit;

private static float k = 0.999999f;

private float T = 1000f;

List<Integer> finalRoute;

private Random r = new Random();

private int iterations = 0;

private int nodes = 0;

private int worseOptionTaken = 0;

public AnnealTSP(int[][] edges, int limit){

this.limit = limit;

this.edges = edges;

n = edges.length;

}

public void Solve(){

List<Integer> route = PrimaryRoute();

SimulatedAnnealing(route);

if(GetValue(finalRoute) <= limit){

StringBuilder sb = new StringBuilder();

for (int i = 0; i < finalRoute.size() - 1; i++) {

sb.append(finalRoute.get(i) + 1);

sb.append(" - (");

sb.append(edges[finalRoute.get(i)][finalRoute.get(i+1)]);

sb.append(") -> ");

}

sb.append(finalRoute.get(finalRoute.size() - 1) + 1);

sb.append(" - (");

sb.append(edges[finalRoute.get(finalRoute.size() - 1)][finalRoute.get(0)]);

sb.append(") -> ");

sb.append(finalRoute.get(0) + 1);

System.out.println(sb.toString());

System.out.println("Path length: " + GetValue(finalRoute));

System.out.println("Iterations: " + iterations);

System.out.println("Nodes: " + nodes);

System.out.println("Worse option was taken: " + worseOptionTaken);

} else {

System.out.println("Couldn`t find solution within limit");

System.out.println("Iterations: " + iterations);

System.out.println("Nodes: " + nodes);

System.out.println("Worse option was taken: " + worseOptionTaken);

}

}

private void SimulatedAnnealing(List<Integer> route){

List<Integer> current = route;

nodes++;

while (T > 1e-3 && GetValue(current) > limit){

iterations++;

T = Schedule(T);

List<Integer> next = GetSuccessor(current);

nodes++;

int deltaE = GetValue(current) - GetValue(next);

if(deltaE > 0 ){

current = next;

} else if(Math.exp(deltaE / T) > r.nextDouble()){ // коли темпераутра низька Math.exp(deltaE / T) майже рівне нулю (ймовірність Больцмана)

current = next;

worseOptionTaken++;

}

}

finalRoute = new ArrayList<>(current);

}

private List<Integer> PrimaryRoute(){

List<Integer> route = new ArrayList<>();

for(int i = 1; i < edges.length; i++) route.add(i);

java.util.Collections.shuffle(route, new Random());

route.add(0, 0);

return route;

}

private List<Integer> GetSuccessor(List<Integer> route){

List<Integer> newRoute = new ArrayList<>(route);

int size = newRoute.size();

if(size <= 2) {

return newRoute;

}

int i = 1 + r.nextInt(size - 1);

int j = 1 + r.nextInt(size - 1);

if(i == j) {

j = (j + 1) % (size - 1) + 1;

}

int left = Math.min(i, j);

int right = Math.max(i, j);

if (r.nextBoolean()) {

while (left < right) {

Swap(newRoute, left, right);

left++;

right--;

}

} else {

List<Integer> segment = new ArrayList<>(newRoute.subList(left, right + 1));

newRoute.subList(left, right + 1).clear();

int availableSize = newRoute.size();

int insertPos;

if (availableSize <= 1) {

insertPos = 1;

} else {

insertPos = 1 + r.nextInt(availableSize);

}

newRoute.addAll(insertPos, segment);

}

return newRoute;

}

private int GetValue(List<Integer> route){

int value = 0;

for(int i = 0; i < route.size() - 1; i++){

value += edges[route.get(i)][route.get(i + 1)];

}

value += edges[route.get(route.size() - 1)][route.get(0)];

return value;

}

private static void Swap(List<Integer> list, int i, int j){

int t = list.get(i);

list.set(i, list.get(j));

list.set(j, t);

}

private static float Schedule(float t){

return (k \* t);

}

}

import java.util.\*;

public class BCTR {

private int[][] edges;

private int n;

private static int limit;

private boolean[] visited;

private int totalCost = Integer.MAX\_VALUE;

private List<Integer> finalRoute;

private boolean found = false;

private int iterations = 0;

private int deadEnds = 0;

private int nodes = 0;

private int nodesInMemory = 0;

public BCTR(int[][] edges, int limit) {

this.edges = edges;

this.n = edges.length;

this.visited = new boolean[n];

this.limit = limit;

}

public void Solve(){

int start = 0;

visited[start] = true;

List<Integer> route = new ArrayList<>();

route.add(start);

Backtracking(start, 0, route);

if(totalCost == Integer.MAX\_VALUE){

System.out.println("Couldn't find solution within given limit");

} else{

StringBuilder sb = new StringBuilder();

for (int i = 0; i < finalRoute.size() - 1; i++) {

sb.append(finalRoute.get(i) + 1);

sb.append(" - (");

sb.append(edges[finalRoute.get(i)][finalRoute.get(i+1)]);

sb.append(") -> ");

}

sb.append(finalRoute.get(finalRoute.size() - 1) + 1);

sb.append(" - (");

sb.append(edges[finalRoute.get(finalRoute.size() - 1)][finalRoute.get(0)]);

sb.append(") -> ");

sb.append(finalRoute.get(0) + 1);

System.out.println(sb.toString());

System.out.println("Path length: " + totalCost);

System.out.println("Iterations: " + iterations);

System.out.println("Nodes: " + nodes);

System.out.println("Max noodes in memory: " + nodesInMemory);

System.out.println("Dead ends: " + deadEnds);

}

}

private void Backtracking(int current, int cost, List<Integer> route){

nodes++;

nodesInMemory = Math.max(nodesInMemory, route.size());

if (found) return;

if(AllCitiesVisited()){

int finalCost = cost + edges[current][route.get(0)];

if (finalCost < limit) {

totalCost = finalCost;

finalRoute = new ArrayList<>(route);

found = true;

} else deadEnds++;

return;

}

//Integer[] sortedNeighbors = SortNeighbors(current);

Integer[] sortedNeighbors = SortNeighbors2(current);

for(int i = 0; i < sortedNeighbors.length; i++){

iterations++;

if(!visited[sortedNeighbors[i]]){

int newCost = cost + edges[current][sortedNeighbors[i]];

if(newCost >= limit){

deadEnds++;

break;

}

visited[sortedNeighbors[i]] = true;

route.add(sortedNeighbors[i]);

Backtracking(sortedNeighbors[i], newCost, route);

route.remove(route.size() - 1);

visited[sortedNeighbors[i]] = false;

}

}

}

private boolean AllCitiesVisited(){

boolean allVisited = true;

for(int i = 0; i < visited.length; i++){

if(!visited[i]){

allVisited = false;

}

}

return allVisited;

}

private Integer[] SortNeighbors(int current){

Integer[] neighbors = new Integer[n];

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

neighbors[i] = i;

}

Arrays.sort(neighbors, (a, b) -> Integer.compare(edges[current][a], edges[current][b]));

return neighbors;

}

private Integer[] SortNeighbors2(int current){

Integer[] neighbors = new Integer[n];

for (int i = 0; i < n; i++) neighbors[i] = i;

final double alpha = 0.3; // коефіцієнт впливу "lookahead"

Arrays.sort(neighbors, (a, b) -> {

if (a == current) return 1;

if (b == current) return -1;

if (visited[a] && !visited[b]) return 1;

if (visited[b] && !visited[a]) return -1;

// --- Рахуємо середню відстань до НЕвідвіданих вершин ---

double sumA = 0, sumB = 0;

int countA = 0, countB = 0;

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

if (j == a || visited[j]) continue;

sumA += edges[a][j];

countA++;

}

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

if (j == b || visited[j]) continue;

sumB += edges[b][j];

countB++;

}

double nextA = (countA > 0) ? sumA / countA : 0;

double nextB = (countB > 0) ? sumB / countB : 0;

// --- Оцінюємо сумарний "score" з урахуванням локальної та прогнозованої вартості ---

double scoreA = edges[current][a] + alpha \* nextA;

double scoreB = edges[current][b] + alpha \* nextB;

return Double.compare(scoreA, scoreB);

});

return neighbors;

}

}

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.

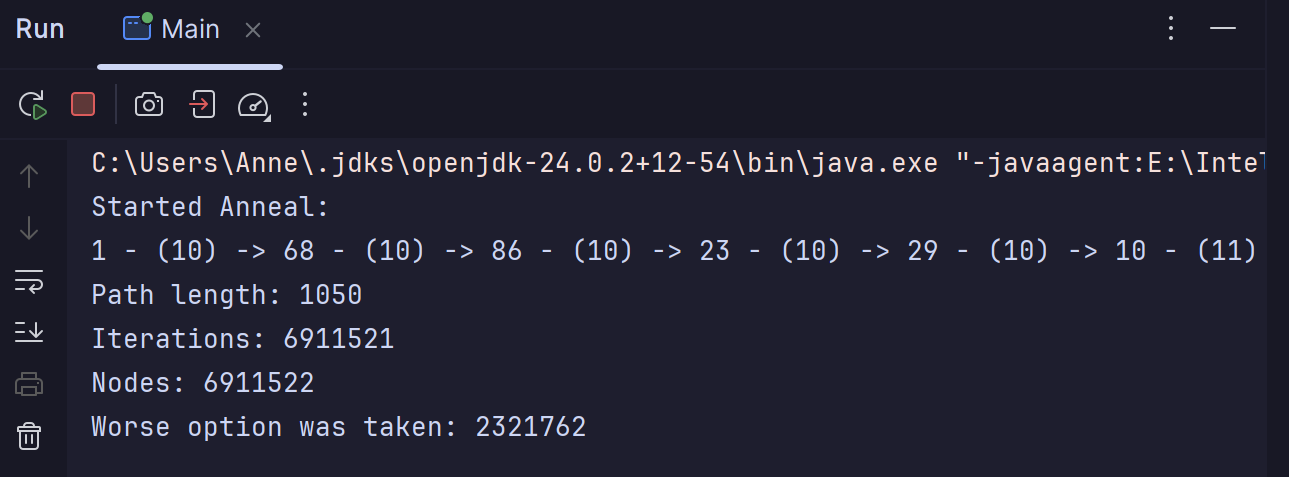


Рисунок 3.1 – Алгоритм…

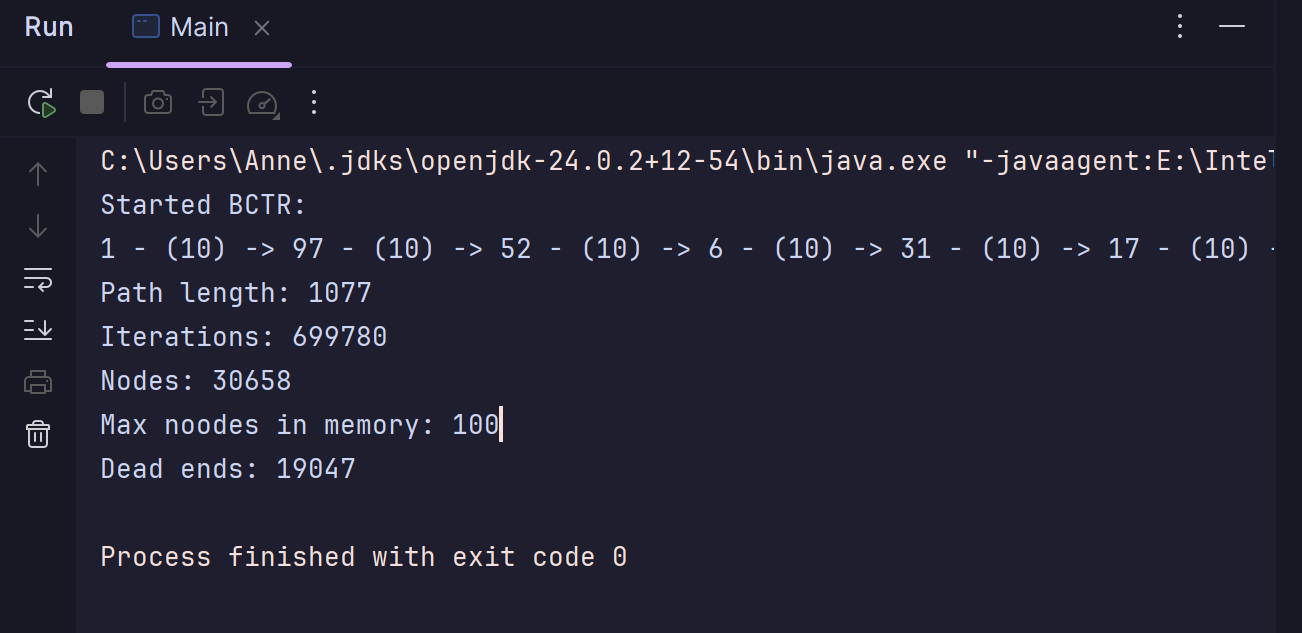


Рисунок 3.2 – Алгоритм BCTR

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BCTR з евристикою найближчого сусіда, задачі TSP для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання BCTR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього вузлів | Всього вузлів у пам’яті |
| 1 Ліміт довжини шляху: 1050 | 4951 | 1 | 102 | 100 |
| 2 | 4958 | 4 | 110 | 100 |
| 3 | 7331 | 1606 | 2485 | 100 |
| 4 | 4950 | 0 | 100 | 100 |
| 5 | 5401 | 239 | 554 | 100 |
| 6 | 5598 | 460 | 750 | 100 |
| 7 | 5000 | 24 | 151 | 100 |
| 8 | 5141 | 111 | 292 | 100 |
| 9 | 13930 | 5434 | 9084 | 100 |
| 10 | 4964 | 6 | 115 | 100 |
| 11 | 4954 | 2 | 105 | 100 |
| 12 | 6837 | 1347 | 1991 | 100 |
| 13 | 5014 | 30 | 166 | 100 |
| 14 | 589088 | 489224 | 584242 | 100 |
| 15 | 4992 | 24 | 143 | 100 |
| 16 | 5003 | 26 | 155 | 100 |
| 17 | 146093 | 112183 | 141247 | 100 |
| 18 | 37028 | 24029 | 32181 | 100 |
| 19 | 5051 | 64 | 204 | 100 |
| 20 | 8401 | 2573 | 3554 | 100 |

В таблиці 3.2 наведені характеристики оцінювання алгоритму ANNEAL, задачі TSP для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання симуляції відпалу

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього вузлів | Всього вузлів у пам’яті |
| 1 Ліміт довжини шляху: 1050 | 6698792 | 6698793 | 2 |
| 2 | 6898087 | 6898088 | 2 |
| 3 | 6800382 | 6800383 | 2 |
| 4 | 6747781 | 6747782 | 2 |
| 5 | 6779667 | 6779668 | 2 |
| 6 | 6972745 | 6972746 | 2 |
| 7 | 6976602 | 6976603 | 2 |
| 8 | 6801967 | 6801968 | 2 |
| 9 | 6811448 | 6811449 | 2 |
| 10 | 6733127 | 6733128 | 2 |
| 11 | 6830507 | 6830508 | 2 |
| 12 | 7004074 | 7004075 | 2 |
| 13 | 6799749 | 6799750 | 2 |
| 14 | 6894414 | 6894415 | 2 |
| 15 | 6764407 | 6764408 | 2 |
| 16 | 6881246 | 6881247 | 2 |
| 17 | 6920555 | 6920556 | 2 |
| 18 | 6851562 | 6851563 | 2 |
| 19 | 6972985 | 6972986 | 2 |
| 20 | 6647721 | 6647722 | 2 |

В таблиці 3.3 наведені характеристики оцінювання алгоритму BCTR з власною евристикою, задачі TSP для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання BCTR з власною евристикою

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього вузлів | Всього вузлів у пам’яті |
| 1 Ліміт довжини шляху: 1080 | 8541 | 2640 | 3695 | 100 |
| 2 | 4950 | 0 | 100 | 100 |
| 3 | 5760 | 540 | 911 | 100 |
| 4 | 4962 | 6 | 113 | 100 |
| 5 | 4954 | 2 | 105 | 100 |
| 6 | 5427 | 240 | 579 | 100 |
| 7 | 4964 | 6 | 115 | 100 |
| 8 | 5165 | 117 | 316 | 100 |
| 9 | 4951 | 1 | 102 | 100 |
| 10 | 4950 | 0 | 100 | 100 |
| 11 | 4964 | 7 | 116 | 100 |
| 12 | 6168 | 698 | 1319 | 100 |
| 13 | 33709 | 20923 | 28862 | 100 |
| 14 | 4964 | 6 | 115 | 100 |
| 15 | 5362 | 240 | 517 | 100 |
| 16 | 6573 | 1072 | 1726 | 100 |
| 17 | 5938 | 625 | 1090 | 100 |
| 18 | 4950 | 0 | 100 | 100 |
| 19 | 5182 | 121 | 334 | 100 |
| 20 | 6033 | 658 | 1187 | 100 |

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритм симуляції відпалу та Backtracking з евристикою найближчого сусіда. Було виявлено, що точність алгоритму симуляції відпалу залежить від швидкості зниження температури. Чим температура вище, тим ймовірніше алгоритм йде на ризикований крок прийняти негативні зміни, щоб вийти з локального мінімуму.

Евристика найближчого сусіда в базовому випадку виявилася ефективнішою: при невеликому діапазоні значень відстаней між містами вона часто знаходить короткі маршрути. Проте вона не враховує перспективу подальшого обходу невідвіданих вершин. Власна евристика виправляє цей недолік, оцінюючи не лише відстань до найближчого сусіда, а й середню відстань цієї вершини до всіх ще невідвіданих міст, що дозволяє вибирати маршрути з більш оптимальними перспективами розвитку шляху.