**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»**

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*Бобрик Максим Геннадійович ІП-12*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов Олексій Олександрович*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc81070688)

[2 ЗаВдання 4](#_Toc81070689)

[3 Виконання 8](#_Toc81070690)

[3.1 Псевдокод алгоритмів 8](#_Toc81070691)

[3.2 Програмна реалізація 8](#_Toc81070692)

[3.2.1 Вихідний код 8](#_Toc81070693)

[3.2.2 Приклади роботи 8](#_Toc81070694)

[3.3 Дослідження алгоритмів 8](#_Toc81070695)

[Висновок 11](#_Toc81070696)

[Критерії оцінювання 12](#_Toc81070697)

# 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 (таблиця 2.1) за допомогою алгоритму неінформативного пошуку **АНП**, алгоритму інформативного пошуку **АІП,** що використовує задану евристичну функцію Func, або алгоритму локального пошуку **АЛП та бектрекінгу,** що використовує задану евристичну функцію Func.

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:**

* **8-ферзів** – Задача про вісім ферзів полягає в такому розміщенні восьми ферзів на шахівниці, що жодна з них не ставить під удар один одного. Тобто, вони не повинні стояти в одній вертикалі, горизонталі чи діагоналі.
* **8-puzzle** – гра, що складається з 8 однакових квадратних пластинок з нанесеними числами від 1 до 8. Пластинки поміщаються в квадратну коробку, довжина сторони якої в три рази більша довжини сторони пластинок, відповідно в коробці залишається незаповненим одне квадратне поле. Мета гри – переміщаючи пластинки по коробці досягти впорядковування їх по номерах, бажано зробивши якомога менше переміщень.
* **Лабіринт** – задача пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху. Структура лабіринту зчитується з файлу, або генерується програмою.
* **LDFS** – Пошук вглиб з обмеженням глибини.
* **BFS** – Пошук вшир.
* **IDS** – Пошук вглиб з ітеративним заглибленням.
* **A\*** – Пошук А\*.
* **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
* **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь А може стояти на одній лінії з ферзем В, проте між ними стоїть ферзь С; тому А не б’є В).
* **F2** – кількість пар ферзів, які б’ють один одного без урахування видимості.
* **H1** – кількість фішок, які не стоять на своїх місцях.
* **H2** – Манхетенська відстань.
* **H3** – Евклідова відстань.
* **COLOR** – Задача розфарбування карти самостійно обраної країни, не менше 20 регіонів (областей). Необхідно розфарбувати карту не більше ніж у 4 різні кольори. Мається на увазі приписування кожному регіону власного кольору так, щоб кольори сусідніх регіонів відрізнялись. Використовувати евристичну функцію, яка повертає кількість пар суміжних вузлів, що мають однаковий колір (тобто кількість конфліктів). Реалізувати алгоритм пошуку із поверненнями (backtracking) для розв’язання поставленої задачі. Для підвищення швидкодії роботи алгоритму використати евристичну функцію, а початковим станом вважати випадкову вершину.
* **HILL –** Пошук зі сходженням на вершину з використанням із використанням руху вбік (на 100 кроків) та випадковим перезапуском (кількість необхідних разів запуску визначити самостійно).
* **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.
* **BEAM** – Локальний променевий пошук. Робоча характеристика – кількість променів k. Експерименти проводи із кількістю променів від 2 до 21.
* **MRV** – евристика мінімальної кількості значень;
* **DGR** – ступенева евристика.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Задача** | **АНП** | **АІП** | **АЛП** | **Func** |
| 1 | Лабіринт | LDFS | A\* |  | H2 |
| 2 | Лабіринт | LDFS | RBFS |  | H3 |
| 3 | Лабіринт | BFS | A\* |  | H2 |
| 4 | Лабіринт | BFS | RBFS |  | H3 |
| 5 | Лабіринт | IDS | A\* |  | H2 |
| 6 | Лабіринт | IDS | RBFS |  | H3 |
| 7 | 8-ферзів | LDFS | A\* |  | F1 |
| 8 | 8-ферзів | LDFS | A\* |  | F2 |
| 9 | 8-ферзів | LDFS | RBFS |  | F1 |
| 10 | 8-ферзів | LDFS | RBFS |  | F2 |
| 11 | 8-ферзів | BFS | A\* |  | F1 |
| 12 | 8-ферзів | BFS | A\* |  | F2 |
| 13 | 8-ферзів | BFS | RBFS |  | F1 |
| 14 | 8-ферзів | BFS | RBFS |  | F2 |
| 15 | 8-ферзів | IDS | A\* |  | F1 |
| 16 | 8-ферзів | IDS | A\* |  | F2 |
| 17 | 8-ферзів | IDS | RBFS |  | F1 |
| 18 | Лабіринт | LDFS | A\* |  | H3 |
| 19 | 8-puzzle | LDFS | A\* |  | H1 |
| 20 | 8-puzzle | LDFS | A\* |  | H2 |
| 21 | 8-puzzle | LDFS | RBFS |  | H1 |
| 22 | 8-puzzle | LDFS | RBFS |  | H2 |
| 23 | 8-puzzle | BFS | A\* |  | H1 |
| 24 | 8-puzzle | BFS | A\* |  | H2 |
| 25 | 8-puzzle | BFS | RBFS |  | H1 |
| 26 | 8-puzzle | BFS | RBFS |  | H2 |
| 27 | Лабіринт | BFS | A\* |  | H3 |
| 28 | 8-puzzle | IDS | A\* |  | H2 |
| 29 | 8-puzzle | IDS | RBFS |  | H1 |
| 30 | 8-puzzle | IDS | RBFS |  | H2 |
| 31 | COLOR |  |  | HILL | MRV |
| 32 | COLOR |  |  | ANNEAL | MRV |
| 33 | COLOR |  |  | BEAM | MRV |
| 34 | COLOR |  |  | HILL | DGR |
| 35 | COLOR |  |  | ANNEAL | DGR |
| 36 | COLOR |  |  | BEAM | DGR |

# Виконання

## Псевдокод алгоритмів

3.1.1 BFS

**function** bfs(graph, start\_node, end\_node) **do**

**queue**[self.start\_node]

visited = {start\_node: None}

**while** queue:

cur\_node = **queue**.**popleft**

**if** cur\_node == end\_node **do**

**break**

next\_nodes = graph[cur\_node]

**for** next\_node in next\_nodes **do**

**if** next\_node not in visited **do**

**add** next\_node **to end of queue**

visited[next\_node] = cur\_node

**return** **visited**

3.1.2 RBFS

**function rbfs**(node, f\_limit) **do**

successors = empty list

result = None

**if** node == goal **do**

**return** node, None

count = -1

neighbour = **generate** neighbour **of** node

**for** (neighbour **in** neighbours) **do**

**if neighbour not in visited do**

count += 1

**add list**(value **of** neighbour, count, child) **to end of successors**

**end for**

**if not** (**length** **of** successors):

**return** None, maxsize

**while** (**length** **of** successors):

**sort** successors

best\_node = successors[0][2]

**if** (value **of** best\_node > f\_limit) **do**

**return** None, value **of** best\_node

**end if**

**try:**

alternative = successors[1][0]

**except:**

alternative = f\_limit+1

result, best\_node.value = **RBFS\_search**(best\_node, **min**(f\_limit, alternative))

successors[0] = (best\_node.value, successors[0][1], best\_node)

**if** (result != None) **do**

**break**

**end while**

**return** result, None

## Програмна реалізація

### Вихідний код

3.2.1.1. Maze.py

import random  
import pygame as pg  
from collections import deque  
  
  
class Maze:  
 def \_\_init\_\_(self, cols, rows, visualize, size, speed):  
 self.cols = cols  
 self.rows = rows  
 self.visualize = visualize  
 self.size = size  
 self.speed = speed  
 self.maze = [[1]\*self.cols for \_ in range(self.rows)]  
 self.unvisited = []  
  
 def get\_rect(self, x, y):  
 return x \* self.size + 1, y \* self.size + 1, self.size - 2, self.size - 2  
  
 def generate\_matrix(self):  
 for i in range(self.cols):  
 for j in range(self.rows):  
 if i % 2 != 0 and j % 2 != 0:  
 self.unvisited.append([i, j])  
 self.maze[j][i] = 0  
 return self.unvisited  
  
 def remove\_wall(self, first, second):  
 x\_diff = second[0] - first[0]  
 y\_diff = second[1] - first[1]  
 add\_x = int((x\_diff / abs(x\_diff))) if (x\_diff != 0) else 0  
 add\_y = int((y\_diff / abs(y\_diff))) if (y\_diff != 0) else 0  
 self.maze[first[0] + add\_x][first[1] + add\_y] = 0  
  
 def get\_neighbours(self, x, y):  
 check\_next\_node = lambda x, y: True if 0 <= x < self.cols and 0 <= y < self.rows and not self.maze[x][y] else False  
 ways = ([-2, 0], [0, -2], [2, 0], [0, 2])  
 return [[x + dx, y + dy] for dx, dy in ways if check\_next\_node(x + dx, y + dy)]  
  
 def create\_graph(self, x, y):  
 check\_next\_node = lambda x, y: True if 0 <= x < self.cols and 0 <= y < self.rows and not self.maze[y][x] else False  
 ways = ([-1, 0], [0, -1], [1, 0], [0, 1])  
 return [(x + dx, y + dy) for dx, dy in ways if check\_next\_node(x + dx, y + dy)]  
  
 def maze\_generate(self):  
 start = [1, 1]  
 cur\_node = start  
 unvisited = self.generate\_matrix()  
 stack = deque([start])  
  
 if self.visualize:  
 pg.init()  
 sc = pg.display.set\_mode([self.cols \* self.size, self.rows \* self.size], pg.FULLSCREEN)  
 clock = pg.time.Clock()  
 [[pg.draw.rect(sc, pg.Color(190, 0, 0), self.get\_rect(x, y))  
 for x, col in enumerate(row) if col] for y, row in enumerate(self.maze)]  
  
 while len(unvisited) > 0:  
 neighbours\_node = self.get\_neighbours(cur\_node[0], cur\_node[1])  
 if len(neighbours\_node) != 0:  
 random\_number = random.randint(0, len(neighbours\_node)-1)  
 neighbour\_node = neighbours\_node[random\_number]  
 neighbours\_node.remove(neighbour\_node)  
 unvisited.remove(neighbour\_node)  
 stack.append(neighbour\_node)  
 self.remove\_wall(cur\_node, neighbour\_node)  
 cur\_node = neighbour\_node  
 self.maze[cur\_node[0]][cur\_node[1]] = 2  
 if self.visualize:  
 [[pg.draw.rect(sc, pg.Color(145, 105, 205), self.get\_rect(x, y))  
 for x, col in enumerate(row) if col == 0 or col == 2] for y, row in enumerate(self.maze)]  
 [exit() for event in pg.event.get() if event.type == pg.QUIT]  
 pg.display.flip()  
 clock.tick(self.speed)  
  
 elif len(stack) > 0:  
 cur\_node = stack.popleft()  
 else:  
 random\_number = random.randint(0, len(unvisited))-1  
 cur\_node = unvisited[random\_number]  
 unvisited.remove(cur\_node)  
  
 for i in range(self.cols):  
 for j in range(self.rows):  
 if self.maze[i][j] != 1:  
 self.maze[i][j] = 0  
  
 graph = {}  
 for y, row in enumerate(self.maze):  
 for x, col in enumerate(row):  
 if not col:  
 graph[(x, y)] = graph.get((x, y), []) + self.create\_graph(x, y)  
  
 return self.maze, graph

3.2.1.2 Cell.py

from math import sqrt  
  
  
class Cell:  
 def \_\_init\_\_(self, x: int, y: int, stat: int, value=None):  
 self.x = x  
 self.y = y  
 self.stat = stat  
 self.value = value  
  
 def set\_value(self, cell):  
 if self.stat != 1:  
 self.value = sqrt((self.x - cell.x) \*\* 2 + (self.y - cell.y) \*\* 2)  
  
 def \_\_eq\_\_(self, other):  
 if other == None:  
 return False  
 else:  
 return self.x == other.x and self.y == other.y

3.2.1.4 BFS.py

import pygame as pg  
from collections import deque  
  
  
class BFS:  
 iterations = 0  
 states = 0  
  
 def \_\_init\_\_(self, maze, graph, start\_node, end\_node, visual, speed, size):  
 self.maze = maze  
 self.graph = graph  
 self.start\_node = start\_node  
 self.end\_node = end\_node  
 self.speed = speed  
 self.size = size  
 self.visual = visual  
  
 if self.visual:  
 pg.init()  
 self.sc = pg.display.set\_mode([len(maze) \* size, len(maze[0]) \* size], pg.FULLSCREEN)  
 self.clock = pg.time.Clock()  
 self.sc.fill(pg.Color(21, 35, 51))  
 [[pg.draw.rect(self.sc, pg.Color(190, 0, 0), self.get\_rect(x, y, self.size))  
 for x, col in enumerate(row) if col] for y, row in enumerate(self.maze)]  
  
 @staticmethod  
 def get\_rect(x, y, size):  
 return x \* size + 1, y \* size + 1, size - 2, size - 2  
  
 def print\_path(self, visited, cur\_node):  
 [pg.draw.rect(self.sc, pg.Color(120, 150, 0), self.get\_rect(x, y, self.size)) for x, y in visited]  
 path\_segment = cur\_node  
 while path\_segment:  
 pg.draw.rect(self.sc, pg.Color('white'), self.get\_rect(\*path\_segment, self.size))  
 path\_segment = visited[path\_segment]  
 [exit() for event in pg.event.get() if event.type == pg.QUIT]  
 pg.display.flip()  
 self.clock.tick(self.speed)  
  
 def bfs(self):  
 queue = deque([self.start\_node])  
 visited = {self.start\_node: None}  
  
 while queue:  
 BFS.iterations += 1  
 cur\_node = queue.popleft()  
 if cur\_node == self.end\_node:  
 break  
 next\_nodes = self.graph[cur\_node]  
 for next\_node in next\_nodes:  
 if next\_node not in visited:  
 BFS.states += 1  
 queue.append(next\_node)  
 visited[next\_node] = cur\_node  
 if self.visual:  
 [pg.draw.rect(self.sc, pg.Color(120, 150, 0), self.get\_rect(x, y, self.size)) for x, y in visited]  
 path\_segment = cur\_node  
 while path\_segment:  
 pg.draw.rect(self.sc, pg.Color('white'), self.get\_rect(\*path\_segment, self.size))  
 path\_segment = visited[path\_segment]  
 [exit() for event in pg.event.get() if event.type == pg.QUIT]  
 pg.display.flip()  
 self.clock.tick(self.speed)  
 self.print\_path(visited, cur\_node)  
 return BFS.iterations, BFS.states

3.2.1.4 BFS.py

import pygame as pg  
from Cell import Cell  
from BFS import BFS  
from sys import maxsize  
  
  
class RBFS:  
 iterations = 0  
 states = 0  
 stored\_states = 0  
  
 def \_\_init\_\_(self, maze, start\_node, end\_node, visual, speed, size):  
 self.maze = maze  
 self.start\_node = start\_node  
 self.end\_node = end\_node  
 self.speed = speed  
 self.size = size  
 self.visual = visual  
 self.rows = len(maze)  
 self.cols = len(maze[0])  
 self.maze\_cells = [[Cell]\*self.cols for \_ in range(self.rows)]  
 self.last = []  
 self.visited = []  
 self.graph = {}  
  
 if self.visual:  
 pg.init()  
 self.sc = pg.display.set\_mode([len(maze) \* size, len(maze[0]) \* size], pg.FULLSCREEN)  
 self.clock = pg.time.Clock()  
 self.sc.fill(pg.Color(21, 35, 51))  
 [[pg.draw.rect(self.sc, pg.Color(190, 0, 0), BFS.get\_rect(x, y, self.size))  
 for x, col in enumerate(row) if col] for y, row in enumerate(self.maze)]  
 [exit() for event in pg.event.get() if event.type == pg.QUIT]  
  
 def \_rbfs(self):  
 self.generate\_cells()  
 for y, row in enumerate(self.maze):  
 for x, col in enumerate(row):  
 if not col:  
 self.graph[(x, y)] = self.graph.get((x, y), []) + self.get\_next\_nodes(x, y)  
 self.rbfs(self.start\_node, maxsize - 1)  
 return RBFS.iterations, RBFS.states, RBFS.stored\_states  
  
 def get\_next\_nodes(self, x, y):  
 check\_next\_node = lambda x, y: True if 0 <= x < self.cols and 0 <= y < self.rows and self.maze[y][x] != 1 else False  
 ways = ([-1, 0], [0, -1], [1, 0], [0, 1])  
 return [self.maze\_cells[x + dx][y + dy] for dx, dy in ways if check\_next\_node(x + dx, y + dy)]  
  
 def generate\_cells(self):  
 for i in range(self.cols):  
 for j in range(self.rows):  
 self.maze\_cells[i][j] = Cell(i, j, self.maze[j][i])  
 self.maze\_cells[i][j].set\_value(self.end\_node)  
  
 def rbfs(self, node, f\_limit):  
 successors = []  
 result = None  
  
 if node == self.end\_node:  
 return node, None  
  
 neighbours = self.graph[node.x, node.y]  
 self.last.append(node)  
 self.visited.append(node)  
  
 count = -1  
  
 for neighbour in neighbours:  
 count += 1  
 if neighbour not in self.last:  
 successors.append((neighbour.value, count, neighbour))  
 RBFS.states += len(successors)  
 if RBFS.stored\_states < len(successors): RBFS.stored\_states = len(successors)  
  
 if not len(successors):  
 return None, maxsize  
  
 if self.visual:  
 [pg.draw.rect(self.sc, pg.Color(120, 150, 0), BFS.get\_rect(point.x, point.y, self.size)) for point in  
 self.visited]  
 [pg.draw.rect(self.sc, pg.Color("white"), BFS.get\_rect(point.x, point.y, self.size)) for point in self.last]  
 pg.draw.rect(self.sc, pg.Color("white"), BFS.get\_rect(self.end\_node.x, self.end\_node.y, self.size))  
 pg.display.flip()  
 self.clock.tick(self.speed)  
  
 while len(successors):  
 RBFS.iterations += 1  
 successors.sort()  
 best\_node = successors[0][2]  
  
 if best\_node.value > f\_limit:  
 self.last.remove(node)  
 return None, best\_node.value  
  
 try:  
 alternative = successors[1][0]  
 except (Exception,):  
 alternative = f\_limit+1  
  
 result, best\_node.value = self.rbfs(best\_node, min(f\_limit, alternative))  
 successors[0] = (best\_node.value, successors[0][1], best\_node)  
  
 if result != None:  
 break  
 return result, None

3.2.1.5 main.py

from Maze import Maze  
from RBFS import RBFS  
from Cell import Cell  
from BFS import BFS  
import time  
  
def main():  
 cols, rows, visual, speed, size = 21, 21, False, 60, 5  
 start\_node\_bfs, end\_node\_bfs = (1, 1), (cols-2, rows-2)  
 start\_node\_rbfs, end\_node\_rbfs = Cell(1, 1, 0), Cell(cols-2, rows-2, 0)  
  
 for i in range(20):  
 maze = Maze(cols, rows, visual, size, speed).maze\_generate()  
 bfs\_data = BFS(maze[0], maze[1], start\_node\_bfs, end\_node\_bfs, visual, speed, size).bfs()  
 time.sleep(1)  
 rbfs\_data = RBFS(maze[0], start\_node\_rbfs, end\_node\_rbfs, visual, speed, size).\_rbfs()  
 time.sleep(1)  
 print(f"{'-'\*10}BFS{'-'\*10}\n"  
 f"Iterations: {bfs\_data[0]}\nNumber of states: {rbfs\_data[1]}\n")  
 print(f"{'-'\*10}RBFS{'-'\*10}\n"  
 f"Iterations: {rbfs\_data[0]}\nNumber of states: {rbfs\_data[1]}\nStored states: {rbfs\_data[2]}\n")  
main()

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму «Пошук вшир», задачі проходження лабіринту для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму «Пошук вшир»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
| Стан 1 | 116 | - | 116 | 441 |
| Стан 2 | 152 | - | 155 | 441 |
| Стан 3 | 163 | - | 163 | 441 |
| Стан 4 | 167 | - | 169 | 441 |
| Стан 5 | 129 | - | 129 | 441 |
| Стан 6 | 152 | - | 154 | 441 |
| Стан 7 | 180 | - | 183 | 441 |
| Стан 8 | 155 | - | 158 | 441 |
| Стан 9 | 200 | - | 199 | 441 |
| Стан 10 | 182 | - | 183 | 441 |
| Стан 11 | 108 | - | 300 | 441 |
| Стан 12 | 165 | - | 165 | 441 |
| Стан 13 | 161 | - | 160 | 441 |
| Стан 14 | 175 | - | 177 | 441 |
| Стан 15 | 157 | - | 160 | 441 |
| Стан 16 | 155 | - | 155 | 441 |
| Стан 17 | 128 | - | 128 | 441 |
| Стан 18 | 181 | - | 181 | 441 |
| Стан 19 | 163 | - | 164 | 441 |
| Стан 20 | 172 | - | 174 | 441 |

1) Середня кількість етапів , які знадобилось для досягнення розв’язку (ітерації): 157;

2) Середня кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе: 0;

3) Середня кількість згенерованих станів під час пошуку: 158;

4) Середня кількість станів, що зберігаються в пам’яті під час роботи програми: 441;

В таблиці 3.2 наведені характеристики оцінювання алгоритму «Рекурсивний пошук за першим найкращим співпадінням»», задачі проходження лабіринту для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання алгоритму «Рекурсивний пошук за першим найкращим співпадінням»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
| Стан 1 | 57 | - | 60 | 49 |
| Стан 2 | 321 | - | 190 | 56 |
| Стан 3 | 192 | - | 138 | 80 |
| Стан 4 | 307 | - | 190 | 71 |
| Стан 5 | 237 | - | 166 | 93 |
| Стан 6 | 142 | - | 126 | 106 |
| Стан 7 | 110 | - | 105 | 95 |
| Стан 8 | 177 | - | 149 | 117 |
| Стан 9 | 63 | - | 60 | 53 |
| Стан 10 | 176 | - | 133 | 85 |
| Стан 11 | 49 | - | 58 | 55 |
| Стан 12 | 70 | - | 70 | 66 |
| Стан 13 | 151 | - | 119 | 82 |
| Стан 14 | 174 | - | 120 | 58 |
| Стан 15 | 78 | - | 82 | 78 |
| Стан 16 | 221 | - | 147 | 74 |
| Стан 17 | 191 | - | 127 | 61 |
| Стан 18 | 221 | - | 149 | 72 |
| Стан 19 | 118 | - | 81 | 41 |
| Стан 20 | 97 | - | 84 | 69 |

1) Середня кількість етапів , які знадобилось для досягнення розв’язку (ітерації): 157;

2) Середня кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе: 0;

3) Середня кількість згенерованих станів під час пошуку: 117;

4) Середня кількість станів, що зберігаються в пам’яті під час роботи програми: 73;

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритми неінформативного (Пошук вшир), інформативного та локального пошуку (Рекурсивний пошук за першим найкращим співпадінням), було проведено порівняльний аналіз ефективності використання алгоритмів, згідно якого видно що RBFS є більш економним по пам’яті, але повільніше в порівнянні з BFS.

Критерії оцінювання

За умови здачі лабораторної роботи до 30.10.2022 включно максимальний бал дорівнює – 5. Після 30.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 10%;
* програмна реалізація алгоритму – 60%;
* дослідження алгоритмів – 25%;
* висновок – 5%.