**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)

*ІП-12 Басараб Олег Андрійович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc118832804)

[2 ЗаВдання 4](#_Toc118832805)

[3 Виконання 8](#_Toc118832806)

[3.1 Псевдокод алгоритмів 8](#_Toc118832807)

[3.1.1 Пошук з обмеженням глибини 8](#_Toc118832808)

[3.1.2 Рекурсивний пошук по першому найкращому збігу 9](#_Toc118832809)

[3.2 Програмна реалізація 11](#_Toc118832810)

[3.2.1 Вихідний код 11](#_Toc118832811)

[3.2.2 Приклади роботи 16](#_Toc118832812)

[3.3 Дослідження алгоритмів 18](#_Toc118832813)

[Висновок 21](#_Toc118832814)

[Критерії оцінювання 22](#_Toc118832815)

# 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

Варіант 2

Лабіринт, пошук з обмеженням глибини, рекурсивний пошук по першому найкращому збігу з евристикою евклідова відстань

## Псевдокод алгоритмів

### Пошук з обмеженням глибини

**procedure** DLS\_search (maze, limit)

curr = Node (state = maze.initial\_state, parent = None)

stack = stack ()

stack.push (curr)

**while** notstack.empty () **do**

curr = stack.pop ()

path = curr.get\_path\_to\_node ()

**if** path.length () – 1 == limit **then**

continue

**end if**

**if** curr.state == maze.goal\_state **then**

**return** curr.get\_path\_to\_node ()

**end if**

children = expand (curr, maze)

**for** child **in** children **do**

stack.push (child)

**end for**

**end while**

**return** None

**end procedure**

**procedure** expand (node, maze)

children = list ()

path = node.get\_path\_to\_node ()

**for** dir **in** maze\_directions **do**

**if** maze[node.state].dir != wall **then**

calculate child\_state

**if** child\_state **not in** path **then**

child = Node (state = child\_state, parent = node)

children.append (child)

**end if**

**end if**

**end for**

**return** children

**end procedure**

### Рекурсивний пошук по першому найкращому збігу

**procedure** recursive\_best\_first\_search (maze)

initial\_state = maze.initial\_state

node = RBFS\_search (maze, Node (state = initial\_state, parent = None), inf)[0]

**return** node.get\_path\_to\_node ()

**end procedure**

**procedure** RBFS\_search (maze, node, f\_limit)

**if** node.state == maze.goal\_state **then**

**return** node, None

**end if**

successors = expand (node, maze)

**if** successors.empty () **then**

**return** None, inf

**end if**

**for** s **in** successors **do**

s.f\_value = max (s.f\_value, node.f\_value)

**end for**

**while** notsuccessors.empty () **do**

successors.sort ()

path = curr.get\_path\_to\_node ()

**if** successors[0].f\_value > f\_limit **then**

return None, successors[0].f\_value

**end if**

**if** successors.length () > 1 **then**

alternative\_f\_value = successors[1].f\_value

**else**

alternative\_f\_value = inf

**end if**

result, successors[0].f\_value = RBFS\_search (maze, successors[0], min (f\_limit, alternative\_f\_value))

**if** result != None **then**

break

**end if**

**end while**

**return** result, None

**end procedure**

**procedure** expand (node, maze)

children = list ()

path = node.get\_path\_to\_node ()

**for** dir **in** maze\_directions **do**

**if** maze[node.state].dir != wall **then**

calculate child\_state

**if** child\_state **not in** path **then**

child = Node (state = child\_state, parent = node)

children.append (child)

**end if**

**end if**

**end for**

**return** children

**end procedure**

**procedure** calculate\_h (node, maze)

node.h\_value = sqrt (pow(node.state[0] – maze.goal\_state[0], 2) + pow(node.state[1] – maze.goal\_state[1], 2))

**end procedure**

## Програмна реалізація

### Вихідний код

from pyamaze import maze

from problem import Problem

import math

class Node:

def \_\_init\_\_(self, state, parent, goal\_state) -> None:

self.\_parent = parent

self.\_state = state

self.\_goal\_state = goal\_state

self.\_h\_value = self.\_calculate\_h\_value()

if self.\_parent:

self.\_g\_value = parent.\_g\_value + 1

else:

self.\_g\_value = 0

self.\_f\_value = self.\_h\_value + self.\_g\_value

@property

def f\_value(self):

return self.\_f\_value

@f\_value.setter

def f\_value(self, value):

self.\_f\_value = value

def \_calculate\_h\_value(self):

return math.sqrt((self.\_state[0] - self.\_goal\_state[0])\*\*2 + (self.\_state[1] - self.\_goal\_state[1])\*\*2)

def is\_goal(self):

if self.\_state == self.\_goal\_state:

return True

return False

def expand(self, m: maze):

directions = {"E": lambda x: (x[0], x[1] + 1),

"W": lambda x: (x[0], x[1] - 1),

"N": lambda x: (x[0] - 1, x[1]),

"S": lambda x: (x[0] + 1, x[1])}

children = []

path = self.get\_path\_to\_node()

for d in "ENSW":

if m.maze\_map[self.\_state][d]:

child\_state = directions[d](self.\_state)

if child\_state not in path:

children.append(Node(child\_state, self, self.\_goal\_state))

return children

def get\_path\_to\_node(self):

path = []

path.append(self.\_state)

curr = self

while curr.\_parent != None:

curr = curr.\_parent

path.append(curr.\_state)

path.reverse()

return path

from pyamaze import maze

from dataclasses import dataclass

@dataclass

class Problem:

\_m: maze

\_initial\_state: tuple

\_number\_of\_iterations: int = 0

\_number\_of\_states: int = 0

\_max\_states\_in\_memory: int = 0

\_number\_of\_dead\_ends: int = 0

\_goal\_state: tuple = (1, 1)

def \_\_init\_\_(self, m: maze):

self.\_m = m

self.\_initial\_state = (self.\_m.rows, self.\_m.cols)

@property

def initial\_state(self):

return self.\_initial\_state

@property

def goal\_state(self):

return self.\_goal\_state

@property

def m(self) -> maze:

return self.\_m

@property

def number\_of\_iterations(self):

return self.\_number\_of\_iterations

@number\_of\_iterations.setter

def number\_of\_iterations(self, value):

self.\_number\_of\_iterations = value

@property

def number\_of\_states(self):

return self.\_number\_of\_states

@number\_of\_states.setter

def number\_of\_states(self, value):

self.\_number\_of\_states = value

@property

def max\_states\_in\_memory(self):

return self.\_max\_states\_in\_memory

@max\_states\_in\_memory.setter

def max\_states\_in\_memory(self, value):

self.\_max\_states\_in\_memory = value

@property

def number\_of\_dead\_ends(self):

return self.\_number\_of\_dead\_ends

@number\_of\_dead\_ends.setter

def number\_of\_dead\_ends(self, value):

self.\_number\_of\_dead\_ends = value

from node import Node

from sys import maxsize

from problem import Problem

from pyamaze import agent, maze

import os

import psutil

import func\_timeout

def recursive\_best\_first\_search(p: Problem):

initial\_state = p.initial\_state

states\_in\_memory = 0

node = func\_timeout.func\_timeout(30 \* 60, RBFS\_search, args = [p, Node(state = initial\_state, parent = None, goal\_state = p.goal\_state), maxsize, states\_in\_memory])[0]

return node.get\_path\_to\_node()

def RBFS\_search(p: Problem, node: Node, f\_limit, states\_in\_memory):

if psutil.Process(os.getpid()).memory\_info().rss > 1024\*\*3:

raise MemoryError("1 Gb memory exceeded")

p.number\_of\_iterations += 1

if node.is\_goal():

return node, None

successors = node.expand(p.m)

if not len(successors):

return None, maxsize

for s in successors:

s.f\_value = max(s.f\_value, node.f\_value)

p.number\_of\_states += len(successors)

while len(successors):

successors.sort(key = lambda x: x.f\_value)

if successors[0].f\_value > f\_limit:

return None, successors[0].f\_value

if len(successors) > 1:

alternative\_f\_value = successors[1].f\_value

else:

alternative\_f\_value = maxsize

if states\_in\_memory > p.max\_states\_in\_memory:

p.max\_states\_in\_memory = states\_in\_memory

result, successors[0].f\_value = RBFS\_search(p, successors[0], min(f\_limit, alternative\_f\_value), states\_in\_memory + len(successors))

if result != None:

break

return result, None

from node import Node

from sys import maxsize

from problem import Problem

import os

import psutil

import func\_timeout

def depth\_limited\_search(p: Problem, limit):

initial\_state = p.initial\_state

node = func\_timeout.func\_timeout(30 \* 60, DLS\_search, args =[p, Node(state = initial\_state, parent = None, goal\_state = p.goal\_state), limit])

return node.get\_path\_to\_node()

def DLS\_search(p: Problem, node: Node, limit):

stack = []

stack.append(node)

p.number\_of\_states += 1

while stack:

if psutil.Process(os.getpid()).memory\_info().rss > 1024\*\*3:

raise MemoryError("1 Gb memory exceeded")

p.number\_of\_iterations += 1

curr\_node = stack.pop()

if len(stack) > p.max\_states\_in\_memory:

p.max\_states\_in\_memory = len(stack)

if len(curr\_node.get\_path\_to\_node()) - 1 == limit:

p.number\_of\_dead\_ends += 1

continue

if curr\_node.is\_goal():

return curr\_node

children = curr\_node.expand(p.m)

stack += children

p.number\_of\_states += len(children)

return None

from recursive\_best\_first\_search import recursive\_best\_first\_search

from depth\_limited\_search import depth\_limited\_search

from pyamaze import maze, agent

import os

import time

from problem import Problem

class Tester:

@staticmethod

def create\_mazes(x = 15, y = 15, loop\_percent = 15, n = 20):

for \_ in range(n):

m = maze(x, y)

m.CreateMaze(loopPercent = loop\_percent, saveMaze = True)

time.sleep(1)

def test\_dls(self, directory = "mazes/", n = 20):

it = 0

st = 0

max\_st = 0

dead\_ends = 0

for filename in os.listdir(directory):

path = os.path.join(directory, filename)

m = maze()

m.CreateMaze(loadMaze = path)

p = Problem(m)

depth\_limited\_search(p, p.m.rows \* p.m.cols // 3)

print(f"Maze {p.m.rows} x {p.m.cols}\n"

f"Number of iterations: {p.number\_of\_iterations}, number of generated states: {p.number\_of\_states}, max number of states in memory: {p.max\_states\_in\_memory} , "

f"number of dead ends: {p.number\_of\_dead\_ends}\n")

it += p.number\_of\_iterations

st += p.number\_of\_states

max\_st += p.max\_states\_in\_memory

dead\_ends += p.number\_of\_dead\_ends

print(f"Average number of iterations: {it/n}\n"

f"Average number of generated states: {st/n}\n"

f"Average max number of states in memory: {max\_st/n}\n"

f"Average number of dead ends: {dead\_ends/n}\n")

def test\_rbfs(self, directory = "mazes/", n = 20):

it = 0

st = 0

max\_st = 0

for filename in os.listdir(directory):

path = os.path.join(directory, filename)

m = maze()

m.CreateMaze(loadMaze = path)

p = Problem(m)

recursive\_best\_first\_search(p)

print(f"Maze {p.m.rows} x {p.m.cols}\n"

f"Number of iterations: {p.number\_of\_iterations}, number of generated states: {p.number\_of\_states}, max number of states in memory: {p.max\_states\_in\_memory}\n")

it += p.number\_of\_iterations

st += p.number\_of\_states

max\_st += p.max\_states\_in\_memory

print(f"Average number of iterations: {it/n}\n"

f"Average number of generated states: {st/n}\n"

f"Average max number of states in memory: {max\_st/n}\n")

def random\_test\_dls(self, x = 15, y = 15, limit = 0, loop\_percent = 15, print\_path = False, visualize = False, save\_maze = False):

m = maze(x, y)

m.CreateMaze(loopPercent = loop\_percent, saveMaze = save\_maze)

p = Problem(m)

if not limit:

limit = x \* y // 2

path = depth\_limited\_search(p, limit)

print(f"Maze {x} x {y}, loop percent = {loop\_percent}\n"

f"Number of iterations: {p.number\_of\_iterations}, number of generated states: {p.number\_of\_states}, max number of states in memory: {p.max\_states\_in\_memory} , "

f"number of dead ends: {p.number\_of\_dead\_ends}\n")

if print\_path:

print(f"Path: {path}\n")

if visualize and path:

a = agent(m, footprints = True, filled = True)

m.tracePath({a: path}, delay = 25)

m.run()

def random\_test\_rbfs(self, x = 15, y = 15, loop\_percent = 15, print\_path = False, visualize = False, save\_maze = False):

m = maze(x, y)

m.CreateMaze(loopPercent = loop\_percent, saveMaze = save\_maze)

p = Problem(m)

path = recursive\_best\_first\_search(p)

print(f"Maze {x} x {y}, loop percent = {loop\_percent}\n"

f"Number of iterations: {p.number\_of\_iterations}, number of generated states: {p.number\_of\_states}, max number of states in memory: {p.max\_states\_in\_memory}\n")

if print\_path:

print(f"Path: {path}\n")

if visualize and path:

a = agent(m, footprints = True, filled = True)

m.tracePath({a: path}, delay = 25)

m.run()

### Приклади роботи

На рисунках 3.1-3.4 показані приклади роботи програми для різних алгоритмів пошуку.

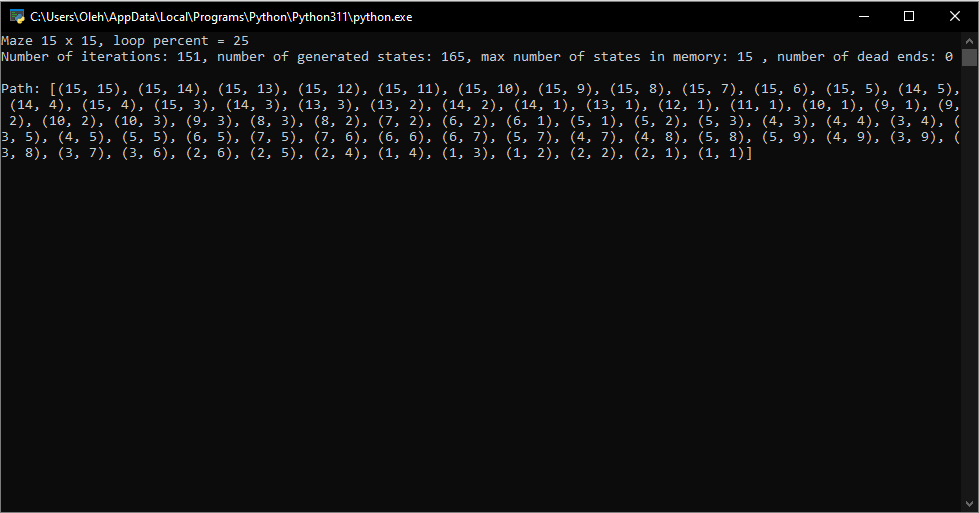


Рисунок 3.1 – Результат виконання алгоритму пошуку з обмеженням глибини

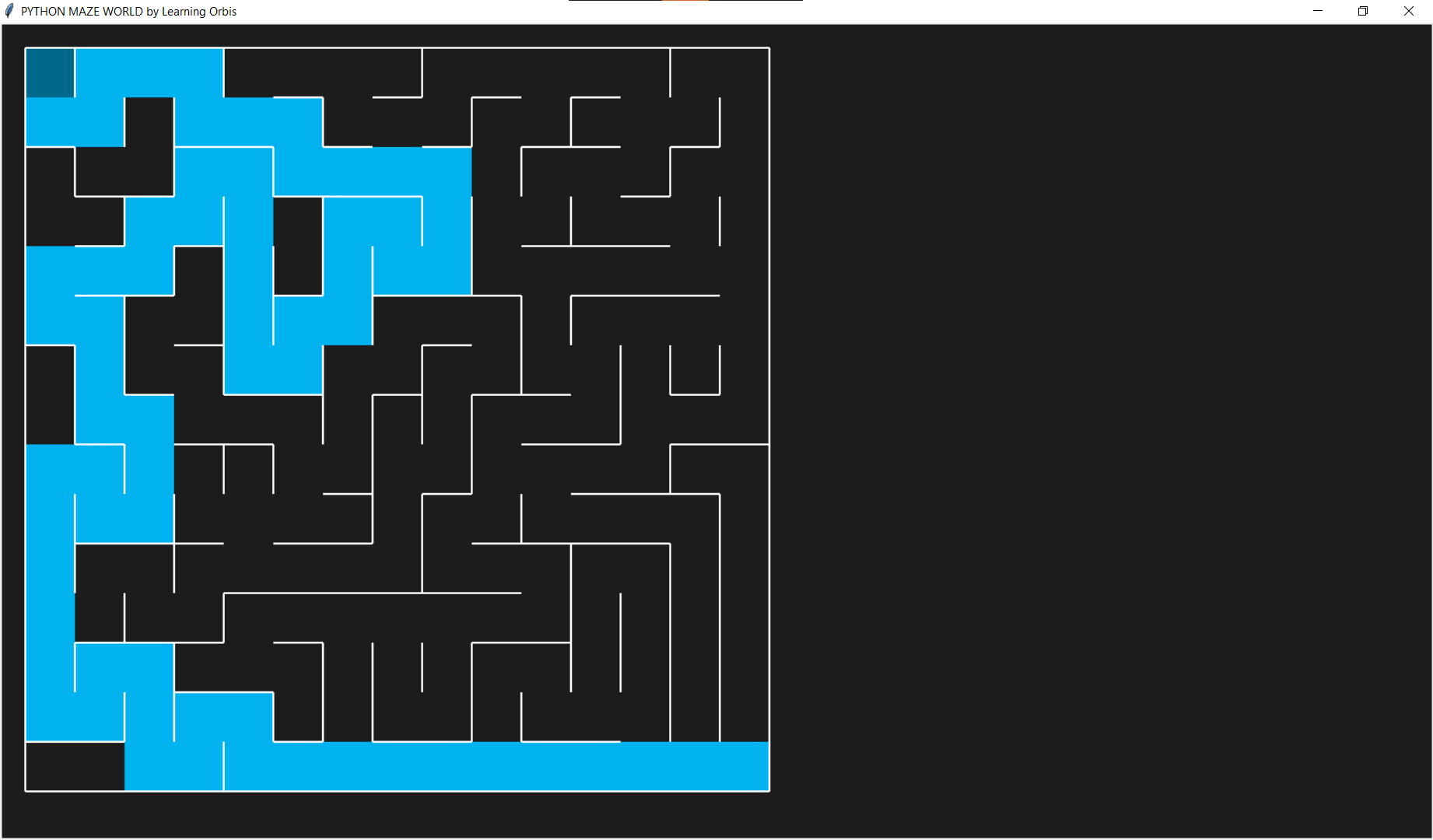


Рисунок 3.4 – Візуалізація отриманого шляху внаслідок виконання алгоритму пошуку з обмеженням глибини

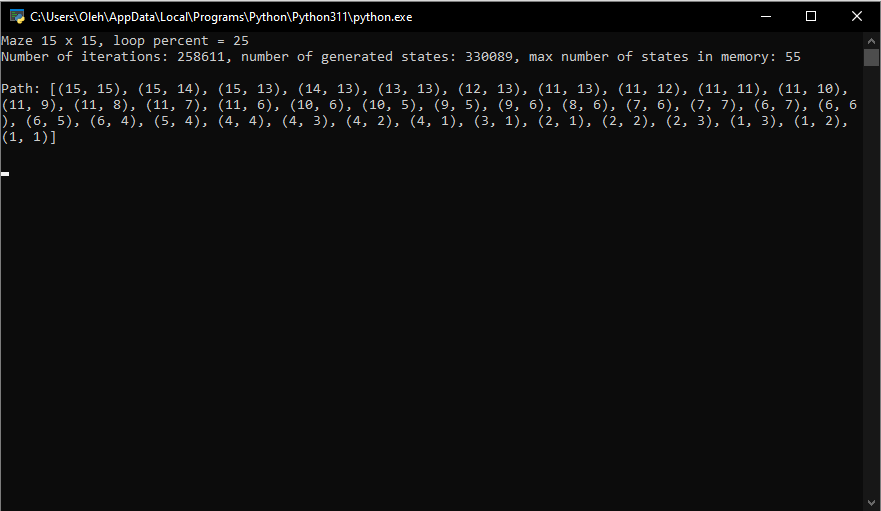


Рисунок 3.3 – Результат виконання алгоритму рекурсивного пошуку по першому найкращому збігу

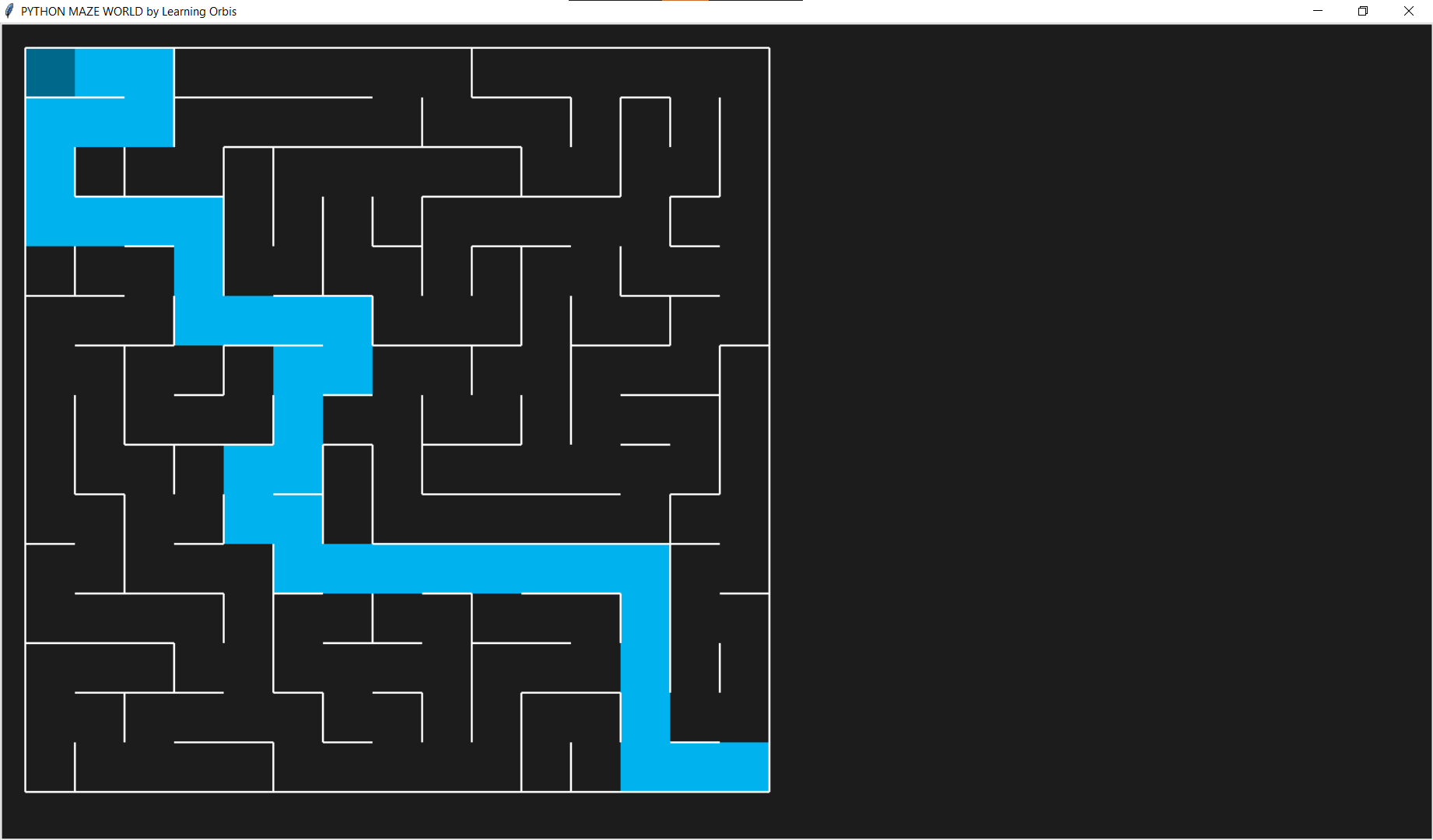


Рисунок 3.4 – Візуалізація отриманого шляху внаслідок виконання рекурсивного пошуку по першому найкращому збігу

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму пошуку з обмеженням глибини задачі лабіринту для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму пошуку з обмеженням глибини

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пом’яті | К-ть глухих кутів |
| Стан 1 | 121 | 126 | 9 | 1 |
| Стан 2 | 243 | 252 | 16 | 7 |
| Стан 3 | 88 | 99 | 11 | 0 |
| Стан 4 | 101 | 113 | 12 | 0 |
| Стан 5 | 43 | 51 | 8 | 0 |
| Стан 6 | 66 | 76 | 11 | 0 |
| Стан 7 | 119 | 132 | 15 | 1 |
| Стан 8 | 81 | 90 | 9 | 0 |
| Стан 9 | 127 | 141 | 15 | 6 |
| Стан 10 | 344 | 352 | 15 | 3 |
| Стан 11 | 76 | 86 | 10 | 0 |
| Стан 12 | 55 | 65 | 11 | 0 |
| Стан 13 | 346 | 356 | 16 | 32 |
| Стан 14 | 57 | 62 | 5 | 0 |
| Стан 15 | 231 | 240 | 17 | 23 |
| Стан 16 | 95 | 110 | 16 | 0 |
| Стан 17 | 123 | 135 | 12 | 2 |
| Стан 18 | 84 | 96 | 13 | 0 |
| Стан 19 | 278 | 288 | 14 | 14 |
| Стан 20 | 42 | 52 | 10 | 0 |
| СЕРЕДНЄ | 136 | 146 | 12 | 4.5 |

В таблиці 3.2 наведені характеристики оцінювання алгоритму рекурсивного пошуку по першому найкращому збігу, задачі лабіринту для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання рекурсивного пошуку по першому найкращому збігу

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пом’яті |
| Стан 1 | 3306 | 3630 | 46 |
| Стан 2 | 34765 | 41394 | 59 |
| Стан 3 | 2894 | 3714 | 49 |
| Стан 4 | 49884 | 55685 | 69 |
| Стан 5 | 2547 | 3001 | 43 |
| Стан 6 | 34260 | 40330 | 57 |
| Стан 7 | 22963 | 28073 | 63 |
| Стан 8 | 3078 | 3484 | 48 |
| Стан 9 | 1681 | 1913 | 57 |
| Стан 10 | 89663 | 103181 | 57 |
| Стан 11 | 29785 | 33372 | 64 |
| Стан 12 | 9737 | 11463 | 47 |
| Стан 13 | 50486 | 58965 | 70 |
| Стан 14 | 7176 | 8214 | 52 |
| Стан 15 | 20167 | 24549 | 56 |
| Стан 16 | 3555 | 4134 | 44 |
| Стан 17 | 17100 | 20389 | 50 |
| Стан 18 | 29832 | 33534 | 70 |
| Стан 19 | 53212 | 60855 | 57 |
| Стан 20 | 13870 | 16095 | 48 |
| СЕРЕДНЄ | 23998 | 27799 | 55 |

Висновок

В ході виконання лабораторної роботи було опрацьовано та здійснено програмну реалізацію алгоритмів пошуку з обмеженням глибини та рекурсивного пошуку по першому найкращому збігу мовою python. Було проведено 20 експериментів для кожного з алгоритмів і занотовано кількість здійснених ітерацій, згенерованих станів, максимальну одночасну кількість станів у пам’яті та кількість глухих кутів (для пошуку з обмеженням глибини).

Тестування продемонструвало, що пошук з обмеженням глибини є неповним та неоптимальним алгоритмом. Однак, якщо підібрати коректне обмеження глибини, то він працюватиме справно, хоч і будуватиме не найкращий з можливих шляхів. Водночас, алгоритм рекурсивного пошуку по першому найкращому збігу, хоч і виявився повним і оптимальним, не є найкращим вибором при роботі з лабіринтами через значну глибину рекурсії, яку досягає алгоритм в ході виконання завдання. Також варто зауважити, що при роботі з великими лабіринтами він потенційно зберігатиме менше станів у пам’яті порівняно з іншими алгоритмами.

Критерії оцінювання

За умови здачі лабораторної роботи до 23.10.2022 включно максимальний бал дорівнює – 5. Після 23.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

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