

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)

ІП-12 Басараб Олег Андрійович
(шифр, прізвище, ім'я, по батькові)

Перевірив

Сопов О.О.
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1	МЕТА ЛАБОРАТОРНОЇ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ	8
3.1	ПСЕВДОКОД АЛГОРИТМІВ.....	8
3.1.1	<i>Пошук з обмеженням глибини</i>	<i>8</i>
3.1.2	<i>Рекурсивний пошук по першому найкращому збігу.....</i>	<i>9</i>
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ	11
3.2.1	<i>Вихідний код.....</i>	<i>11</i>
3.2.2	<i>Приклади роботи.....</i>	<i>16</i>
3.3	ДОСЛІДЖЕННЯ АЛГОРИТМІВ.....	18
	ВИСНОВОК	21
	КРИТЕРІЇ ОЦІНЮВАННЯ	22

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 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 **АПІ**, 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 відрізнятися 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 впорядкування 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 A*.
- **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 T від часу роботи алгоритму t . Можна розглядати лінійну залежність: $T = 1000 - k \cdot t$, 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	АП	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

3 ВИКОНАННЯ

Варіант 2

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

3.1 Псевдокод алгоритмів

3.1.1 Пошук з обмеженням глибини

procedure DLS_search (maze, limit)

curr = Node (state = maze.initial_state, parent = None)

stack = stack ()

stack.push (curr)

while not stack.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

```

3.1.2 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 not successors.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

```

3.2 Програмна реалізація

3.2.1 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)

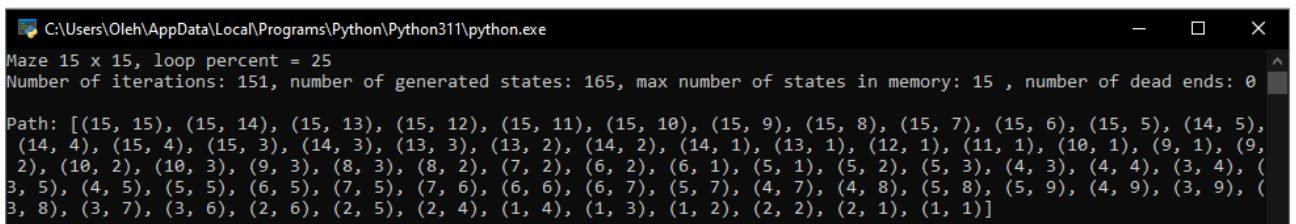
    print(f"Maze {x} x {y}, loop percent = {loop_percent}\n"
          f"Number of iterations: {p.number_of_iterations}, number of generated
states: {p.number_of_states}, max number of states in memory:
{p.max_states_in_memory}\n")
    if print_path:
        print(f"Path: {path}\n")

    if visualize and path:
        a = agent(m, footprints = True, filled = True)
        m.tracePath({a: path}, delay = 25)
        m.run()

```

3.2.2 Приклади роботи

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



```

C:\Users\Oleh\AppData\Local\Programs\Python\Python311\python.exe
Maze 15 x 15, loop percent = 25
Number of iterations: 151, number of generated states: 165, max number of states in memory: 15, number of dead ends: 0
Path: [(15, 15), (15, 14), (15, 13), (15, 12), (15, 11), (15, 10), (15, 9), (15, 8), (15, 7), (15, 6), (15, 5), (14, 5),
(14, 4), (15, 4), (15, 3), (14, 3), (13, 3), (13, 2), (14, 2), (14, 1), (13, 1), (12, 1), (11, 1), (10, 1), (9, 1), (9,
2), (10, 2), (10, 3), (9, 3), (8, 3), (8, 2), (7, 2), (6, 2), (6, 1), (5, 1), (5, 2), (5, 3), (4, 3), (4, 4), (3, 4), (
3, 5), (4, 5), (5, 5), (6, 5), (7, 5), (7, 6), (6, 6), (6, 7), (5, 7), (4, 7), (4, 8), (5, 8), (5, 9), (4, 9), (3, 9), (
3, 8), (3, 7), (3, 6), (2, 6), (2, 5), (2, 4), (1, 4), (1, 3), (1, 2), (2, 2), (2, 1), (1, 1)]

```

Рисунок 3.1 – Результат виконання алгоритму пошуку з обмеженням глибини

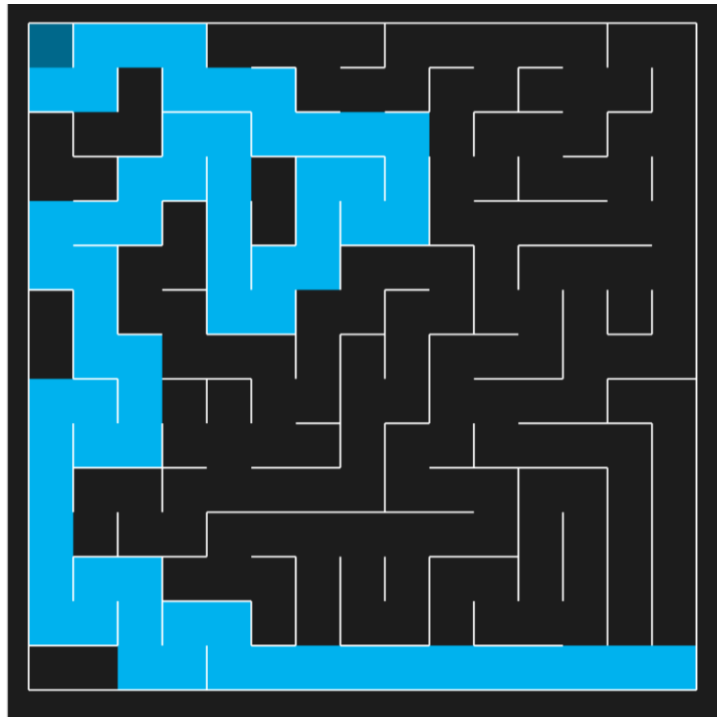


Рисунок 3.4 – Візуалізація отриманого шляху внаслідок виконання алгоритму пошуку з обмеженням глибини

```
C:\Users\Oleh\AppData\Local\Programs\Python\Python311\python.exe
Maze 15 x 15, loop percent = 25
Number of iterations: 258611, number of generated states: 330089, max number of states in memory: 55
Path: [(15, 15), (15, 14), (15, 13), (14, 13), (13, 13), (12, 13), (11, 13), (11, 12), (11, 11), (11, 10),
(11, 9), (11, 8), (11, 7), (11, 6), (10, 6), (10, 5), (9, 5), (9, 6), (8, 6), (7, 6), (7, 7), (6, 7), (6, 6),
(6, 5), (6, 4), (5, 4), (4, 4), (4, 3), (4, 2), (4, 1), (3, 1), (2, 1), (2, 2), (2, 3), (1, 3), (1, 2),
(1, 1)]
```

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

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%.