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-13 Бондаренко М.В.* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2022

ЗМІСТ

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 **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 A*.
- **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
- **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь A може стояти на одній лінії з ферзем B, проте між ними стоїть ферзь C; тому A не б’є B).
- **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	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

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

```
DEFINE FUNCTION Astar(m):
    SET final_path TO {}
    SET a_path TO {}
    SET start_cell TO (m.rows, m.cols)
    SET g_score TO {cell: float('inf') FOR cell IN
m.grid}
    SET g_score[start_cell] TO 0
    SET f_score TO {cell: float('inf') FOR cell IN
m.grid}
    SET f_score[start_cell] TO h(start_cell, (1, 1))
    SET queue TO PriorityQueue()
    queue.put((h(start_cell, (1, 1)), h(start_cell, (1,
1)), start_cell))
    WHILE not queue.empty():
        SET current TO queue.get()[2]
        IF current EQUALS (1, 1):
            break

        FOR direction IN 'ESNW':
            IF m.maze_map[current][direction]:
                SET children TO
neighbour_check(direction, current)
                SET temp_g_score TO g_score[current] + 1
                SET temp_f_score TO temp_g_score +
h(children, (1, 1))
                IF temp_f_score < f_score[children]:
                    SET g_score[children] TO temp_g_score
                    SET f_score[children] TO temp_f_score
                    queue.put((temp_f_score, h(children,
(1, 1)), children))
                    SET a_path[children] TO current
    SET cell TO (1, 1)
    WHILE cell != start_cell:
        SET final_path[a_path[cell]] TO cell
        SET cell TO a_path[cell]
    RETURN final_path
```

```
DEFINE FUNCTION BFS(m):
    SET final_path TO {}
    SET bfs_path TO {}
    SET start_cell TO (m.rows, m.cols)
    SET front_cells TO [start_cell]
    SET passed_list TO [start_cell]
    WHILE len(front_cells) != 0:
        SET current TO front_cells.pop(0)
        IF current EQUALS (1, 1):
            break
        FOR direction IN 'ESNW':
            IF m.maze_map[current][direction]:
                SET children TO
neighbour_check(direction, current)
                IF children IN passed_list:
                    continue
                front_cells.append(children)
                passed_list.append(children)
                SET bfs_path[children] TO current
    SET cell TO (1, 1)
    WHILE cell != start_cell:
        SET final_path[bfs_path[cell]] TO cell
        SET cell TO bfs_path[cell]
    RETURN final_path
```

Допоміжна функція:

```
DEFINE FUNCTION neighbour_check(direction, current):
    IF direction EQUALS 'E':
        SET children TO (current[0], current[1] + 1)
    IF direction EQUALS 'W':
        SET children TO (current[0], current[1] - 1)
    IF direction EQUALS 'N':
        SET children TO (current[0] - 1, current[1])
    IF direction EQUALS 'S':
        SET children TO (current[0] + 1, current[1])
    RETURN children
```

3.2 Програмна реалізація

3.2.1 Вихідний код

main.py

```
from pyamaze import COLOR
from algorithm import *

def main():
    size = int(input("Maze size [2 - 20]: "))
    while size < 2 or size > 20:
        size = int(input("Maze size [2 - 20]: "))

    alg_choose = input("Enter the algorithm you want to use [BFS or Astar]: ")
    while not (alg_choose == "BFS" or alg_choose == "Astar"):
        alg_choose = input("Enter the correct value [BFS or Astar]: ")

    maze_ = maze(size, size)
    maze_.CreateMaze(loopPercent=40)

    path, iterations_counter, states_amount = call_algorithm(maze_, alg_choose)

    a = agent(maze_, shape="arrow", footprints=True, color=COLOR.yellow)
    maze_.tracePath({a: path}, delay=40)
    maze_.run()
    print(
        f"[{alg_choose}] The length of path {len(path)}, the number of
iterations: {iterations_counter}, the amount of unique states: {states_amount}")

if __name__ == '__main__':
    main()
```

algorithm.py

```
import os
import psutil as psutil
import func_timeout
from pyamaze import maze,agent,textLabel
from queue import PriorityQueue
from time import time

def h(cell_a: tuple, cell_b: tuple) -> int: #manhattan distance
    return abs(cell_a[0] - cell_b[0]) + abs(cell_a[1] - cell_b[1])

def call_algorithm(maze, algorithm_name): # with time limit
    if algorithm_name == "BFS":
        return func_timeout.func_timeout(60 * 30, BFS, args=[maze])
    else:
        return func_timeout.func_timeout(60 * 30, Astar, args=[maze])

def neighbour_check(direction, current): #find neighbour
    if direction == 'E':
        children = (current[0], current[1] + 1)
    if direction == 'W':
        children = (current[0], current[1] - 1)
    if direction == 'N':
        children = (current[0] - 1, current[1])
    if direction == 'S':
        children = (current[0] + 1, current[1])
    return children

def Astar(m):
```

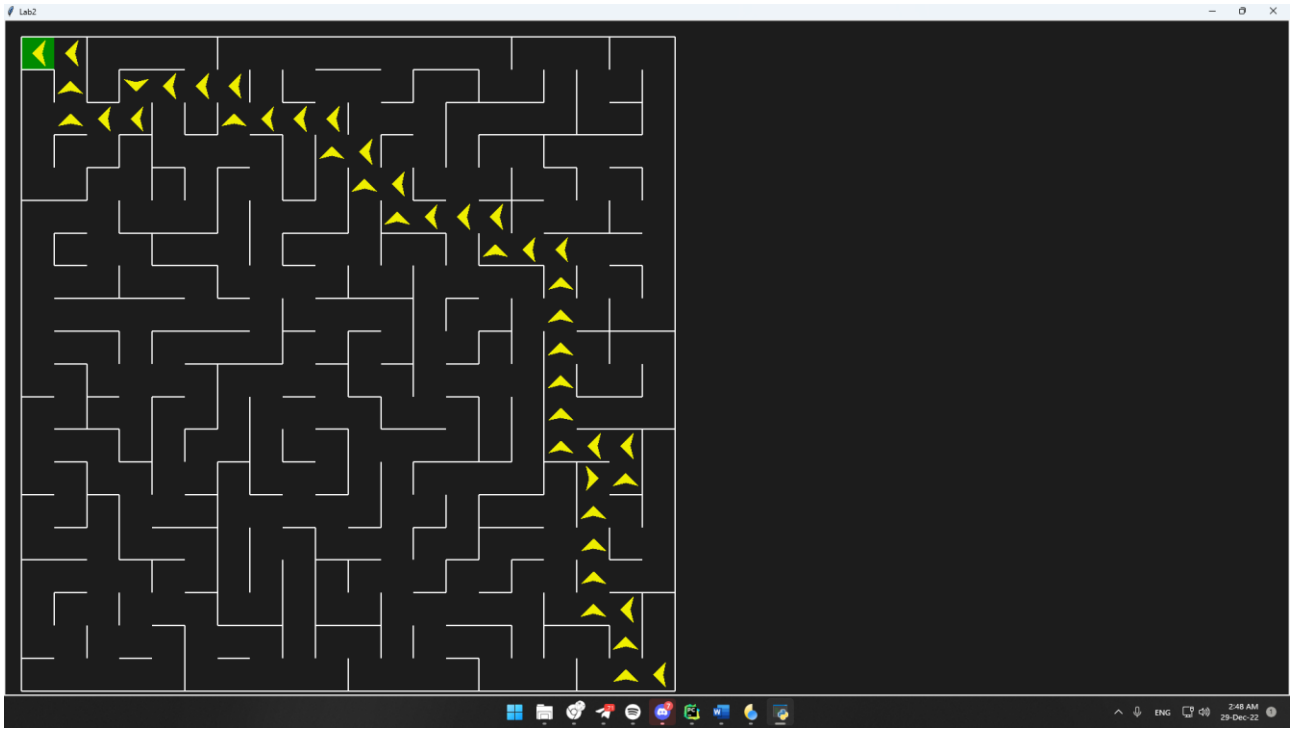
```
    iterations_counter = 0
    states_amount = 0
    states = []
    final_path = {}
    a_path = {}

    start_cell = (m.rows, m.cols)

    g_score = {cell: float('inf') for cell in m.grid}
    g_score[start_cell] = 0

    f_score = {cell: float('inf') for cell in m.grid}
    f_score[start_cell] = h(start_cell, (1, 1))

    queue = PriorityQueue()
    queue.put((h(start_cell, (1, 1)), h(start_cell, (1, 1)), start_cell))

    while not queue.empty():
        if psutil.Process(os.getpid()).memory_info().rss > 1024 ** 3:
            raise MemoryError("1 Gb memory exceeded")
        iterations_counter += 1
        current = queue.get()[2]
        if current == (1, 1):
            states_amount = len(states)
            break

        if current not in states:
            states.append(current)

        for direction in 'ESNW':
            if m.maze_map[current][direction]:
                children = neighbour_check(direction, current)

                temp_g_score = g_score[current] + 1
                temp_f_score = temp_g_score + h(children, (1, 1))

                if temp_f_score < f_score[children]:
                    g_score[children] = temp_g_score
                    f_score[children] = temp_f_score
                    queue.put((temp_f_score, h(children, (1, 1)), children))
                    a_path[children] = current
    cell = (1, 1)
    while cell != start_cell:
        final_path[a_path[cell]] = cell
        cell = a_path[cell]
    return final_path, iterations_counter, states_amount

def BFS(m):
    iterations_counter = 0
    states = []
    states_amount = 0
    final_path = {}
    bfs_path = {}

    start_cell = (m.rows, m.cols)
    front_cells = [start_cell]
    passed_list = [start_cell]

    while len(front_cells) != 0:
        if psutil.Process(os.getpid()).memory_info().rss > 1024 ** 3:
            raise MemoryError("1 Gb memory exceeded")
        iterations_counter += 1
        current = front_cells.pop(0)
        if current == (1, 1):
```

```
            states_amount = len(states)
            break
        if current not in states:
            states.append(current)
        for direction in 'ESNW':
            if m.maze_map[current][direction]:
                children = neighbour_check(direction, current)
                if children in passed_list:
                    continue
                front_cells.append(children)
                passed_list.append(children)
                bfs_path[children] = current
    cell = (1, 1)
    while cell != start_cell:
        final_path[bfs_path[cell]] = cell
        cell = bfs_path[cell]
    return final_path, iterations_counter, states_amount
```

3.2.2 Приклади роботи

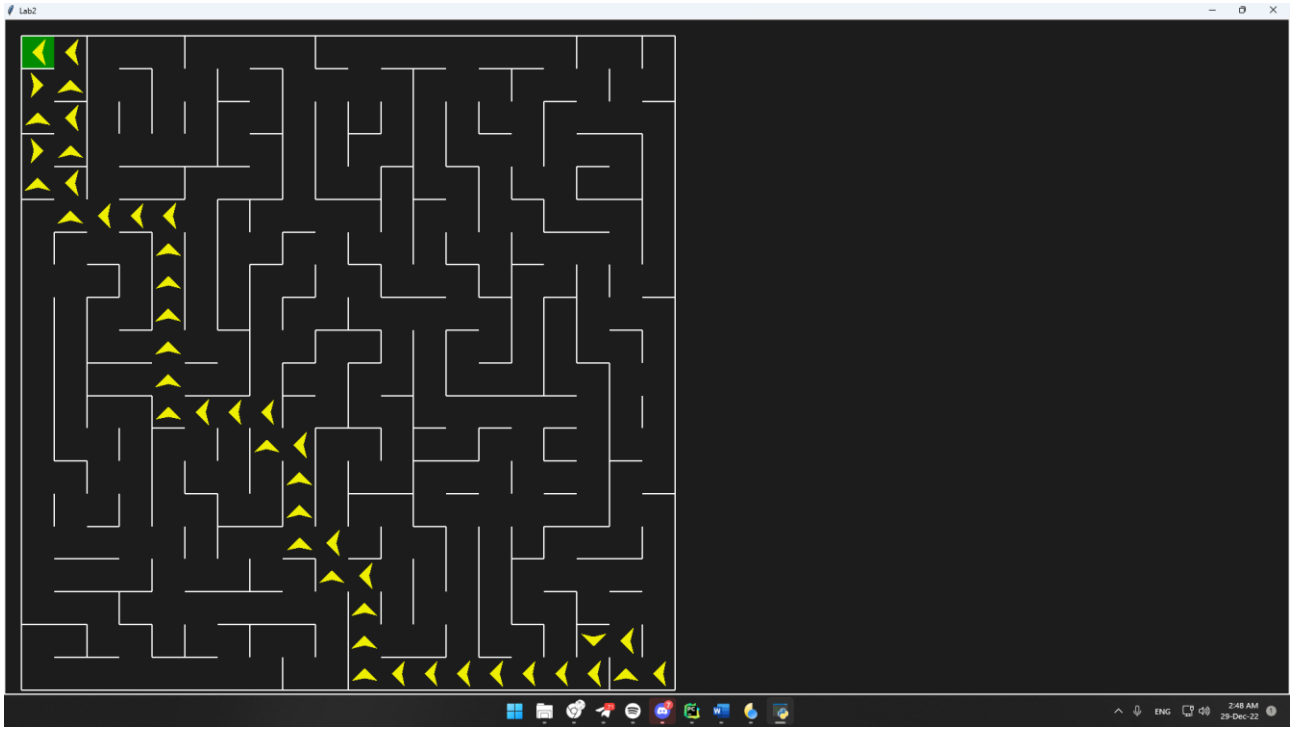
На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.

Рисунок 3.1 – Алгоритм A*



```
D:\KPI\PA-2-5\lab2\venv\Scripts\python.exe D:/KPI/PA-2-5/lab2/main.py
Maze size [2 - 20]: 20
Enter the algorithm you want to use [BFS or Astar]: Astar
[Astar] The length of path 42, the number of iterations: 132, the amount of unique states: 130

Process finished with exit code 0
```

Рисунок 3.2 – Алгоритм BFS



```
D:\KPI\PA-2-5\lab2\venv\Scripts\python.exe D:/KPI/PA-2-5/lab2/main.py
Maze size [2 - 20]: 20
Enter the algorithm you want to use [BFS or Astar]: BFS
[BFS] The length of path 44, the number of iterations: 400, the amount of unique states: 399

Process finished with exit code 0
```

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS задачі лабіринту для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

```
Maze size [2 - 20]: 20
Enter the algorithm you want to use [BFS or Astar]: BFS
[BFS] The length of path 40, the number of iterations: 371, the amount of unique states: 370
[BFS] The length of path 48, the number of iterations: 397, the amount of unique states: 396
[BFS] The length of path 44, the number of iterations: 400, the amount of unique states: 399
[BFS] The length of path 38, the number of iterations: 393, the amount of unique states: 392
[BFS] The length of path 44, the number of iterations: 391, the amount of unique states: 390
[BFS] The length of path 44, the number of iterations: 399, the amount of unique states: 398
[BFS] The length of path 40, the number of iterations: 400, the amount of unique states: 399
[BFS] The length of path 40, the number of iterations: 400, the amount of unique states: 399
[BFS] The length of path 38, the number of iterations: 379, the amount of unique states: 378
[BFS] The length of path 46, the number of iterations: 395, the amount of unique states: 394
[BFS] The length of path 44, the number of iterations: 400, the amount of unique states: 399
[BFS] The length of path 42, the number of iterations: 389, the amount of unique states: 388
[BFS] The length of path 44, the number of iterations: 389, the amount of unique states: 388
[BFS] The length of path 44, the number of iterations: 400, the amount of unique states: 399
[BFS] The length of path 44, the number of iterations: 400, the amount of unique states: 399
[BFS] The length of path 40, the number of iterations: 399, the amount of unique states: 398
[BFS] The length of path 46, the number of iterations: 400, the amount of unique states: 399
[BFS] The length of path 44, the number of iterations: 389, the amount of unique states: 388
[BFS] The length of path 40, the number of iterations: 388, the amount of unique states: 387
[BFS] The length of path 44, the number of iterations: 396, the amount of unique states: 395
[BFS] The length of path 42, the number of iterations: 400, the amount of unique states: 399
```

Середня кількість ітерацій – 394

Середня кількість унікальних станів – 393

Середня кількість станів у пам’яті – 42,6

В таблиці 3.2 наведені характеристики оцінювання алгоритму A* задачі лабіринту для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму A*

```
Maze size [2 - 20]: 20
Enter the algorithm you want to use [BFS or Astar]: Astar
[Astar] The length of path 40, the number of iterations: 160, the amount of unique states: 159
[Astar] The length of path 40, the number of iterations: 109, the amount of unique states: 108
[Astar] The length of path 44, the number of iterations: 255, the amount of unique states: 244
[Astar] The length of path 42, the number of iterations: 230, the amount of unique states: 224
[Astar] The length of path 42, the number of iterations: 220, the amount of unique states: 215
[Astar] The length of path 40, the number of iterations: 85, the amount of unique states: 84
[Astar] The length of path 40, the number of iterations: 90, the amount of unique states: 89
[Astar] The length of path 46, the number of iterations: 297, the amount of unique states: 277
[Astar] The length of path 38, the number of iterations: 53, the amount of unique states: 52
[Astar] The length of path 40, the number of iterations: 78, the amount of unique states: 77
[Astar] The length of path 44, the number of iterations: 181, the amount of unique states: 170
[Astar] The length of path 40, the number of iterations: 123, the amount of unique states: 121
[Astar] The length of path 44, the number of iterations: 219, the amount of unique states: 203
[Astar] The length of path 44, the number of iterations: 267, the amount of unique states: 253
[Astar] The length of path 40, the number of iterations: 172, the amount of unique states: 171
[Astar] The length of path 40, the number of iterations: 74, the amount of unique states: 73
[Astar] The length of path 42, the number of iterations: 147, the amount of unique states: 139
[Astar] The length of path 40, the number of iterations: 170, the amount of unique states: 167
[Astar] The length of path 42, the number of iterations: 121, the amount of unique states: 119
[Astar] The length of path 50, the number of iterations: 324, the amount of unique states: 297
[Astar] The length of path 40, the number of iterations: 91, the amount of unique states: 90
```

Середня кількість ітерацій – 154,6

Середня кількість унікальних станів – 158,7

Середня кількість станів у пам’яті – 41,8

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми A^* та BFS, здійснено їх програмну реалізацію, було здійснено 20 експериментів для кожного алгоритму і прораховано середню кількість ітерацій, пройдених станів та кількість станів у пам’яті. Алгоритм A^* більш оптимізований, бо завжди шукає найкоротший шлях, на відміну від BFS, який проходить майже все дерево (залежить від ситуації), це можна побачити по проведеним експериментам

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%.