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 Бабашев О.Д.* _____
(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

3.2 Програмна реалізація 9

3.2.1 Вихідний код 9

3.2.2 Приклади роботи 10

3.3 Дослідження алгоритмів 13

ВИСНОВОК 16

КРИТЕРІЇ ОЦІНЮВАННЯ 17

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·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

LDFS:

1. Create a stack with the current position and the path taken to reach it

2. While the stack is not empty:

a. Check if the memory usage exceeds 1 GB, if so, raise a MemoryError

b. Pop the last element from the stack and assign it to current, path

c. If current has not been visited before, add it to the list of visited states

d. If the length of the path is equal to the limit, increment the stop counter and continue to the next iteration

e. If the current position is the finish, set CanBeSolve to True, set StatesAmount to the number of unique states visited, and break out of the loop

f. For each direction, check if it is a valid move:

i. If the move is valid, calculate the new position

ii. If the new position has not been visited before, add it to the list of neighbours

g. Add all the neighbours to the stack

RBFS:

1. Create a variable called bound and set it to the heuristic value of the start position and the finish position

2. Create a stack with the current position, the path taken to reach it, and the bound value

3. While the stack is not empty:

a. Check if the memory usage exceeds 1 GB, if so, raise a MemoryError

b. Pop the last element from the stack and assign it to curr, path, curr_bound

c. If curr has not been visited before, add it to the list of visited states

d. If the current position is the finish, set CanBeSolve to True, set StatesAmount to the number of unique states visited, and break out of the loop

e. For each direction, check if it is a valid move:

i. If the move is valid, calculate the new position

ii. If the new position has not been visited before, add it to the list of neighbours, along with the f value calculated as max(heuristic value of the neighbour + length of path, curr_bound)

f. Sort the neighbours by the f value

g. Add the neighbours to the stack

3.2 Програмна реалізація

3.2.1 Вихідний код

main.py

```
from pyamaze import maze, agent
from algorithm import LabirintSearcher
from time import time


def main():
    m = maze(12,12)
    m.CreateMaze()

    choose = 0
    while choose != 1 and choose != 2:
        choose = int(input("LDFS = 1\nRBFS = 2\nchoose algorithm = "))

    searcher = LabirintSearcher(m)
    if choose == 1:
        limit = 100
        searcher.limitedLDFS(limit)
        print(f"stops = {searcher.stops}")
        if not searcher.CanBeSolve:
            print("there is no solution with this limit")
    else:
        searcher.limitedRBFS()
        if not searcher.CanBeSolve:
            print("the agent did not reach the end")

    print(f"iterations = {searcher.iterations}")
    print(f"amount of states = {searcher.StatesAmount}")
    print(f"length result path = {len(searcher.path)}")
    print(f"{searcher.name} finised time = {time() - searcher.start_time}
seconds")
```

```
    a = agent(m, shape='square', footprints=True, color=searcher.color)
    m.tracePath({a: searcher.path}, delay=100)
    m.run()


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

algorithm.py

```
import os
import func_timeout
import psutil
import math
from time import time
from pyamaze import COLOR


class LabirintSearcher:
    def __init__(self, m):
        self.m = m

        self.start = (self.m.rows, self.m.cols)
        self.finish = (1, 1)

        self.iterations = 0
        self.start_time = time()
        self.stops = 0
        self.StatesAmount = 0
        self.states = []

        self.CanBeSolve = False
        self.path = {}

        self.color = COLOR.blue
        self.name = ""


    def __del__(self):
        print(f"memory used = {psutil.Process(os.getpid()).memory_info().rss /
1024 ** 2}")

    def limitedLDFS(self, limit):
        return func_timeout.func_timeout(60 * 30, self.LDFS, args=[limit])

    def LDFS(self, limit):
        self.color = COLOR.red
        self.name = "LDFS"

        stack = [(self.start, [self.start])]
        while stack:
            if psutil.Process(os.getpid()).memory_info().rss > 1024 ** 3:
                raise MemoryError("1 GB used")

            self.iterations += 1
            current, self.path = stack.pop()
            if current not in self.states:
                self.states.append(current)

            if len(self.path) - 1 == limit:
                self.stops += 1
                continue
```

```
            if current == self.finish:
                self.StatesAmount = len(self.states)
                self.CanBeSolve = True
                break

            neighbours = []
            for direction in 'ESNW':
                if self.m.maze_map[current][direction] == True:
                    if direction == 'E':
                        neighbour = (current[0], current[1] + 1)
                    elif direction == 'W':
                        neighbour = (current[0], current[1] - 1)
                    elif direction == 'N':
                        neighbour = (current[0] - 1, current[1])
                    elif direction == 'S':
                        neighbour = (current[0] + 1, current[1])

                    if neighbour not in self.path:
                        neighbours.append((neighbour, self.path + [neighbour]))

            stack += neighbours

#================================================================================
================================

    # Euclidean distance
    @staticmethod
    def h(point_a: tuple, point_b: tuple):
        return math.sqrt((point_a[0] - point_b[0]) ** 2 + (point_a[1] -
point_b[1]) ** 2)

    def limitedRBFS(self):
        return func_timeout.func_timeout(60 * 30, self.RBFS)

    def RBFS(self):
        self.color = COLOR.green
        self.name = "RBFS"

        bound = self.h(self.start, self.finish)
        stack = [(self.start, [self.start], bound)]
        while stack:
            if psutil.Process(os.getpid()).memory_info().rss > 1024 ** 3:
                raise MemoryError("1 GB used")

            self.iterations += 1
            current, self.path, curr_bound = stack.pop()
            if current not in self.states:
                self.states.append(current)

            if current == self.finish:
                self.StatesAmount = len(self.states)
                self.CanBeSolve = True
                break

            neighbours = []
            for direction in 'ESNW':
                if self.m.maze_map[current][direction] == True:
                    if direction == 'E':
                        neighbour = (current[0], current[1] + 1)
                    elif direction == 'W':
                        neighbour = (current[0], current[1] - 1)
                    elif direction == 'N':
                        neighbour = (current[0] - 1, current[1])
```

```
                    elif direction == 'S':
                        neighbour = (current[0] + 1, current[1])

                    if neighbour not in self.path:
                        f_val = max(self.h(neighbour, self.finish) +
len(self.path), curr_bound)
                        neighbours.append((neighbour, self.path + [neighbour],
f_val))

            neighbours.sort(key=lambda x: x[2])
            stack += neighbours
```

3.2.2 Приклади роботи

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

```
LDFS = 1
RBFS = 2
choose algorithm = 1
stops = 0
iterations = 140
amount of states = 140
length result path = 61
LDFS finised time = 0.003630399703979492 seconds
```

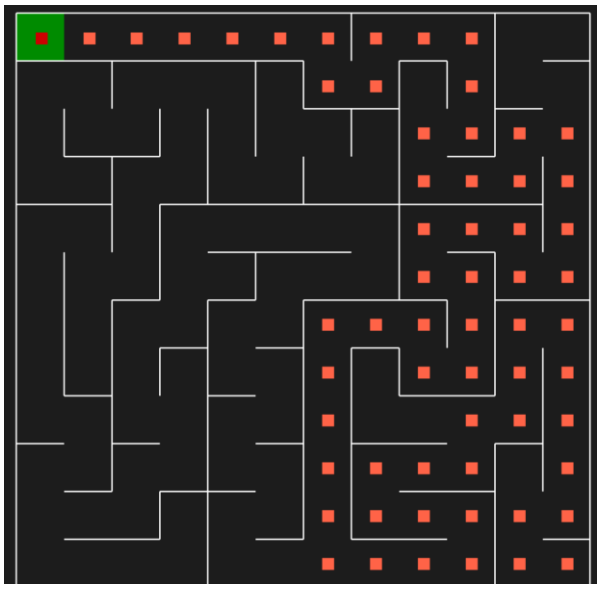

Рисунок 3.1 – Алгоритм LDFS

```
LDFS = 1
RBFS = 2
choose algorithm = 2
iterations = 110
amount of states = 110
length result path = 83
RBFS finised time = 0.0029935836791992188 seconds
```

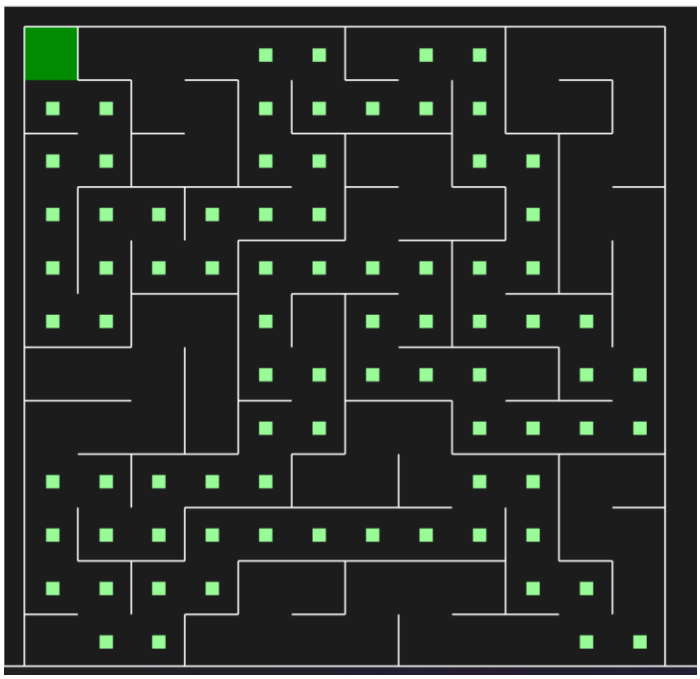

Рисунок 3.2 – Алгоритм RBFS

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму LDFS, задачі лабіринт для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом'яті
Стан 1	194	0	194	46
Стан 2	367	0	367	96
Стан 3	1744	5	90	30
Стан 4	546	0	578	120
Стан 5	1632	4	245	90
Стан 6	1633	4	1633	496
Стан 7	22	0	22	15
Стан 8	462	0	462	297
Стан 9	96	0	96	75
Стан 10	1200	1	365	103
Стан 11	125	0	125	45
Стан 12	1523	5	562	128
Стан 13	1835	9	792	57
Стан 14	456	3	225	74
Стан 15	785	0	785	167
Стан 16	1758	7	356	45
Стан 17	99	0	99	81
Стан 18	115	1	78	51
Стан 19	127	2	68	45
Стан 20	110	0	110	71

В таблиці 3.2 наведені характеристики оцінювання алгоритму RBFS задачі лабіринт для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму RBFS

Початкові стани	Ітерації	Всього станів	Всього станів у пом’яті
Стан 1	40	40	39
Стан 2	141	139	63
Стан 3	108	100	41
Стан 4	126	126	95
Стан 5	78	68	12
Стан 6	123	123	103
Стан 7	144	144	43
Стан 8	1408	1408	299
Стан 9	1306	1306	455
Стан 10	505	505	419
Стан 11	40	40	39
Стан 12	141	139	63
Стан 13	1306	1306	455
Стан 14	126	126	95
Стан 15	78	68	12
Стан 16	123	123	103
Стан 17	123	123	103
Стан 18	1408	1408	299
Стан 19	1306	1306	455
Стан 20	135	107	47

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритм неінформативного пошуку LDFS та інформативного пошуку RBFS. Отримані знання застосовано на практиці. Проведено аналіз алгоритмів.

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%.