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-15, Богун Д.О
Перевірив Головченко М.М.

Київ 2022

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.

№	Задача	АНП	АІП	АЛП	Func
1	Лабіринт	LDFS	A*		H2

3. ВИКОНАННЯ

3.1 Псевдокод алгоритму

3.1.1 LDFS:

```
solveMaze_ldfs(maze,  start, end, limit):
   step  =  [(-1, 0 ),
             ( 0, -1),
             ( 1, 0 ),
             ( 0, 1 )]

   IF start is None OR end is None:
      exit
   left_to_visit = [start]
   visited_nodes = [start]
   iter = 0
   height, width = len(maze), len(maze[0])
   solution = [start]
    WHILE len(left_to_visit) > 0
          do
      iter = 0
      cur_node = left_to_visit.pop()
      IF (cur_node == end):
         break
      FOR new_pos in step
            do
         IF iter<limit:
            node_pos = (cur_node[0] + new_pos[0],cur_node[1] + new_pos[1] )
            IF (node_pos[0] > (height - 1) OR
               node_pos[0] < 0 OR
               node_pos[1] > (width -1) OR
               node_pos[1] < 0):
               continue
            IF maze[node_pos[0]][node_pos[1]] != 0 AND maze[node_pos[0]][node_pos[1]]
!= 4 AND maze[node_pos[0]][node_pos[1]] != 5:
               continue
            IF node_pos in visited_nodes:
               continue
            visited_nodes.append(node_pos)
            left_to_visit.append(node_pos)
            solution.append(node_pos)
            x,y = cur_node
            IF maze[x][y] !=4 AND maze[x][y]!=1:
               maze[x][y] = 8

            iter+=1
         ELSE : continue
             end
      end
```

3.1.2 Псевдокод алгоритму A*:

```
solveMaze(maze, cost, start, end):
    first_node = node(None, start)
    last_node = node(None, end)
    first_node.G = first_node.H = first_node.F = 0
    last_node.G = last_node.H = last_node.F = 0
    left_to_visit = []
    visited_nodes = []
    left_to_visit.append(first_node)
    iter = 0
    max_iter = (len(maze) // 2) ** 10
    step  =  [[-1, 0 ],
              [ 0, -1],
              [ 1, 0 ],
              [ 0, 1 ]]
    height, width = len(maze), len(maze[0])
    deadend = 0
    WHILE len(left_to_visit) > 0:
        iter += 1
        cur_node = left_to_visit[0]
        cur_index = 0
        FOR i, j in (left_to_visit):
              do
           IF j.F < cur_node.F:
              cur_node = j
              cur_index = i
            end
        IF iter > max_iter:
           return returnSolution( maze,  visited_nodes, start, end),visited_nodes
        left_to_visit.pop(cur_index)
        visited_nodes.append(cur_node)
        maze[cur_node.pos[0]][cur_node.pos[1]] = 8
        IF cur_node == last_node:
           return returnSolution( maze,  visited_nodes, start, end), visited_nodes
        children_list = []
        FOR new_pos in step:
              do
           node_pos = (cur_node.pos[0] + new_pos[0], cur_node.pos[1] + new_pos[1])
           IF (node_pos[0] > (height - 1) OR
              node_pos[0] < 0 OR
              node_pos[1] > (width -1) OR
              node_pos[1] < 0):
              CONTINUE
           IF maze[node_pos[0]][node_pos[1]] !=0 AND maze[node_pos[0]][node_pos[1]] != 4
                    AND maze[node_pos[0]][node_pos[1]] != 5:
              CONTINUE
           temp_node = node(cur_node, node_pos)
           children_list.append(temp_node)
            end
        FOR child_node in children_list:
```

```
  do
      IF len([child_visited FOR child_visited in visited_nodes IF  child_visited ==
      child_node]) > 0:
            CONTINUE
      child_node.H = (abs((child_node.pos[0] - last_node.pos[0])) +
            abs((child_node.pos[1] - last_node.pos[1])))
      child_node.G = cur_node.G + cost
      child_node.F = child_node.G + child_node.H
      IF len([k FOR k in left_to_visit IF child_node == k AND  child_node.G > k.G]) >
        0:
            CONTINUE
       left_to_visit.append(child_node)
end
```

3.2Програмна реалізація

3.2.1 **LDFS**

Functions_ldfs.py

```
import time
import os
def solveMaze_ldfs(maze,  start, end):
    if start is None or end is None:
        return
    left_to_visit = [start]
    visited_nodes = [start]
    iter = 0
    step  =  [(-1, 0 ), # вгору
              ( 0, -1), # ліворуч
              ( 1, 0 ), # вниз
              ( 0, 1 )] # праворуч
    height, width = len(maze), len(maze[0])
    solution = [start]
    iterations = 0
    stan_list = []
    limit = 500
    while len(left_to_visit) > 0 and iter<limit:
        stan = 0
        cur_node = left_to_visit.pop()
        if (cur_node == end):
            break
        for new_pos in step:
            node_pos = (cur_node[0] + new_pos[0],cur_node[1] + new_pos[1] )
            if (node_pos[0] > (height ) or
                node_pos[0] < 0 or
                node_pos[1] > (width ) or
                node_pos[1] < 0):
                deadend+=1
                continue
            if maze[node_pos[0]][node_pos[1]] != 0 and maze[node_pos[0]][node_pos[1]] !=
4 and maze[node_pos[0]][node_pos[1]] != 5:
                continue
            if node_pos in visited_nodes:
                continue
            visited_nodes.append(node_pos)
            maze[node_pos[0]][node_pos[1]] = 8
            printMaze(maze, start, end)
            time.sleep(0.04)
            os.system("cls")
            left_to_visit.append(node_pos)
            solution.append(node_pos)
            x,y = cur_node
    if(end not in visited_nodes):
        return False
    printMaze(maze, start, end)
    return True

def readMaze():
    with
open('C:\\Users\\zhmis\\source\\repos\\Lab2_MazeGenerator\\Lab2_MazeGenerator\\Maze.txt')
as file:
        maze_txt = file.read()
    maze_lines = maze_txt.split('\n')
    maze_array = []
    for i in maze_lines:
        maze_array.append([int(item) for item in i])
```

```
    return maze_array

def printMaze(maze, start, end):
    maze_char=[]
    i = 0
    j = 0
    for lines in maze:
        for ch in lines:
            if (i,j) == start:
                print('S', end =''),
            elif (i,j) == end:
                print('E', end =''),
            elif ch == 8 :
                print('#', end =''),
            elif ch == 1:
                print('|', end =''),
            else:
                print(' ', end =''),
            j+=1
        i+=1
        j = 0
        print('')


def findPosition(maze, num):
    i = 0
    j = 0
    for lines in maze:
        i+=1
        for ch in lines:
            j+=1
            if ch == num:
                return(i-1, j-1)
        j = 0

def tryMaze():
    try:
        maze = readMaze()
        maze_copy = readMaze()
        first_node = findPosition(maze, 4)
        last_node = findPosition(maze, 5)
        flag=solveMaze_ldfs(maze,first_node,last_node)
        if(flag==True):
            printMaze(maze,first_node,last_node)
            print("Лабіринт пройдено успішно")

        else:
            printMaze(maze,first_node,last_node)
            print("Не вдалося знайти шлях")
    except TypeError:
        print("Не вдалося знайти шлях")
        return 'Type'
```

main:

```
from functions_ldfs import *
if __name__ == '__main__':
    tryMaze()
```

3.2.2 **A star:**

Node.py

```
class node:
    def __init__(self, parent=None, position=None):
        self.G = 0
        self.H = 0
        self.F = 0
        self.parent = parent
        self.pos = position

    def __eq__(self, other):
        return self.pos == other.pos
```

functions_astar.py

```
import time
import os
import numpy as np
from node import *

def solveMaze_astar(maze, cost, start, end):
    iterations = 0
    first_node = node(None, tuple(start))
    last_node = node(None, tuple(end))
    first_node.G = first_node.H = first_node.F = 0
    last_node.G = last_node.H = last_node.F = 0
    left_to_visit = []
    visited_nodes = []
    left_to_visit.append(first_node)
    iter = 0
    max_iter = (len(maze) // 2) ** 10
    step  =  [[-1, 0 ], # вгору
              [ 0, -1], # ліворуч
              [ 1, 0 ], # вниз
              [ 0, 1 ]] # праворуч
    height, width = len(maze), len(maze[0])
    while len(left_to_visit) > 0 :
        iter += 1
        cur_node = left_to_visit[0]
        cur_index = 0
        for i, j in enumerate(left_to_visit):
            if j.F < cur_node.F:
                cur_node = j
                cur_index = i
        if cur_node.pos == end:
            return True
        if iter > max_iter:
            return False
        left_to_visit.pop(cur_index)

        visited_nodes.append(cur_node)
        if maze[cur_node.pos[0]][cur_node.pos[1]] != 4 and
maze[cur_node.pos[0]][cur_node.pos[1]] !=5:
            maze[cur_node.pos[0]][cur_node.pos[1]] = 8
            time.sleep(0.04)
            os.system('cls')
            printMaze(maze)
        children_list = []
        for new_pos in step:
```

```
            node_pos = (cur_node.pos[0] + new_pos[0], cur_node.pos[1] + new_pos[1])

            if (node_pos[0] > (height - 1) or
                node_pos[0] < 0 or
                node_pos[1] > (width -1) or
                node_pos[1] < 0):
                continue
            if maze[node_pos[0]][node_pos[1]] != 0 and maze[node_pos[0]][node_pos[1]] !=
4 and maze[node_pos[0]][node_pos[1]] != 5:
                continue

            temp_node = node(cur_node, node_pos)

            children_list.append(temp_node)
        for child_node in children_list:

            if len([child_visited for child_visited in visited_nodes if child_visited ==
child_node]) > 0:
                continue
             # еврестична функція, використовуємо манхетенську відстань
            child_node.H = (abs((child_node.pos[0] - last_node.pos[0])) +
                            abs((child_node.pos[1] - last_node.pos[1])))
            child_node.G = cur_node.G + cost

            child_node.F = child_node.G + child_node.H

            if len([k for k in left_to_visit if child_node == k and child_node.G > k.G])
> 0:
                continue

            left_to_visit.append(child_node)



def readMaze():
    with
open('C:\\Users\\zhmis\\source\\repos\\Lab2_MazeGenerator\\Lab2_MazeGenerator\\Maze.txt')
as file:
        maze_txt = file.read()
    maze_lines = maze_txt.split('\n')
    maze_array = []
    for i in maze_lines:
        maze_array.append([int(item) for item in i])
    return maze_array

def printMaze(maze):
    print("# - пройдений шлях")

    for lines in maze:
        for ch in lines:
            if ch == 4:
                print('S', end =''),
            elif ch == 5:
                print('E', end =''),
            elif ch == 8 :
                print('#', end =''),
            elif ch == 1:
                print('|', end =''),
            else:
                print(' ', end =''),
        print('')


def findPosition(maze, num):
```

```
    i = 0
    j = 0
    for lines in maze:
        i+=1
        for ch in lines:
            j+=1
            if ch == num:
                return(i-1, j-1)
        j = 0

def tryMaze():
    try:
        maze = readMaze()
        first_node = findPosition(maze, 4)
        last_node = findPosition(maze, 5)
        cost = 1
        flag = solveMaze_astar(maze,cost,first_node,last_node)
        os.system('cls')
        print("A* алгоритм")
        printMaze(maze)
        if flag == True:
            print("Лабіринт пройдено успішно")
        else:
            print("Не вдалося знайти шлях")
    except TypeError:
        print("Не вдалося знайти шлях")
        return 'Type'


import numpy as np
from functions_astar import *


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

3.3 Приклад роботи

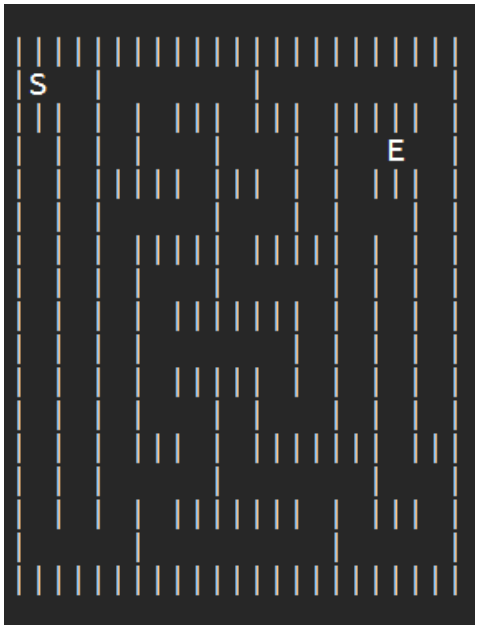

Рисунок 3.3.1 - Лабіринт

На рисунках 3.3.2 і 3.3.3 показані приклади роботи програми для різних алгоритмів пошуку.

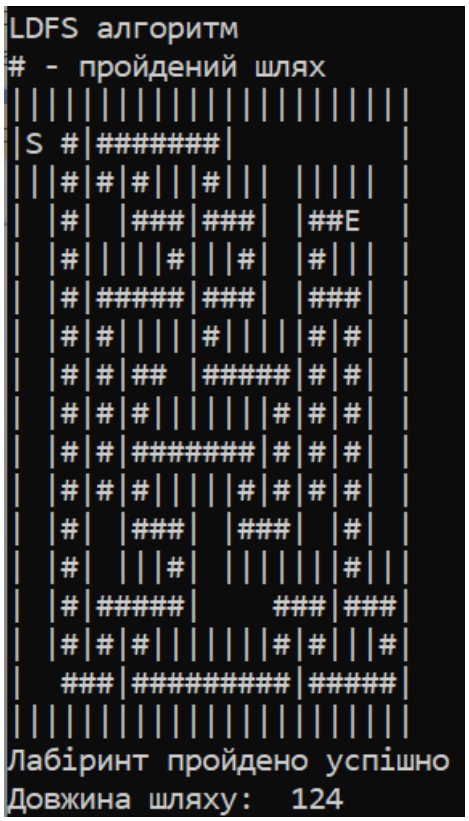

Рисунок 3.3.2 - алгоритм LDFS

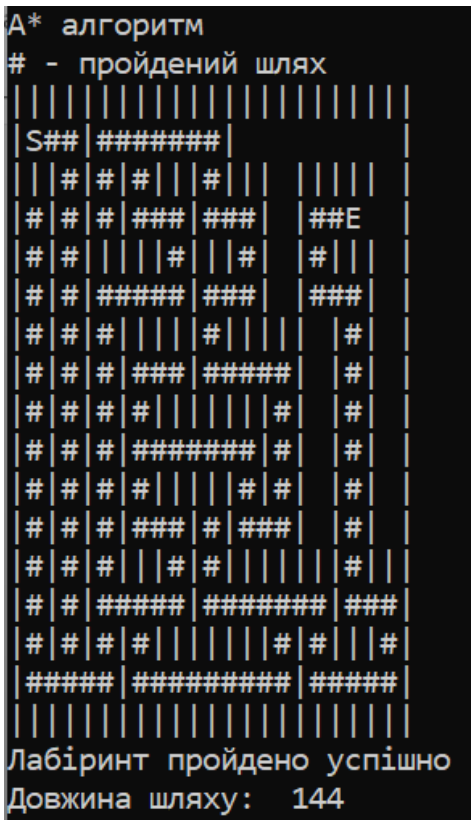

Рисунок 3.3.3 - алгоритм A*

3.4 Дослідження алгоритмів

В таблиці 3.4.1 наведені характеристики оцінювання алгоритму LDFS, задачі «Лабіринт» для 20 початкових станів.

Таблиця 3.4.1 – Характеристики оцінювання алгоритму LDFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	37	0	38	69
Стан 2	49	0	50	69
Стан 3	24	0	25	71
Стан 4	19	0	20	70
Стан 5	31	0	32	48
Стан 6	7	0	8	49
Стан 7	11	0	12	47
Стан 8	38	0	39	55
Стан 9	31	0	32	69
Стан 10	62	0	63	87
Стан 11	33	0	34	91
Стан 12	50	0	51	51
Стан 13	46	0	47	48
Стан 14	46	0	47	53
Стан 15	32	0	33	71
Стан 16	60	1	61	69
Стан 17	15	1	16	69
Стан 18	21	1	22	68
Стан 19	9	1	10	44
Стан 20	20	1	21	48

В таблиці 3.4.2 наведені характеристики оцінювання алгоритму A*, задачі «Лабіринт» для 20 початкових станів.

Таблиця 3.4.2 – Характеристики оцінювання алгоритму A*

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	37	0	36	69
Стан 2	55	0	54	69
Стан 3	24	0	23	71
Стан 4	35	0	34	70
Стан 5	18	0	17	48
Стан 6	7	0	6	49
Стан 7	15	0	14	47
Стан 8	21	0	20	55
Стан 9	31	0	30	69
Стан 10	41	0	40	87
Стан 11	65	0	64	91
Стан 12	37	0	36	51
Стан 13	25	0	24	48
Стан 14	38	0	37	53
Стан 15	34	0	33	71
Стан 16	88	1	61	68
Стан 17	16	1	15	70
Стан 18	32	1	23	49
Стан 19	16	1	11	44
Стан 20	40	1	30	48

Висновок

При виконанні даної лабораторної роботи я ознайомився з двома алгоритмами пошуку: **LDFS** – неінформативний пошук та **A*** - інформативний пошук, використовуючий еврестичну функцію манхеттенської відстані. Написав псевдокод алгоритмів, реалізував програму, яка розв’язує лабіринт двома алгоритмами та провів 20 випробувань для порівняння і перевірки ефективності алгоритмів. Згідно результатам тестування, неможна зробити однозначний висновок, який алгоритм є більш ефективним. Алгоритм LDFS видав кращий результат у серіях 2, 4, 7, 11, 15, 16, 17, 18, 19, 20 – було проведено менше ітерацій . Алгоритм A* показав кращу ефективність в серіях 1, 3, 5, 6, 8, 10, 13, 14 – було згенеровано менше станів.