Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Катедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 2 з дисципліни
«Проєктування алгоритмів»

«Неінформативний, інформативний та локальний пошук»

Виконав(ла) *ІП-11 Лесів В. І.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.М.* _____
(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

```
function LDFS(maze m, int limit)
    explored ← []
    frontier ← [[0,start]]
    dfsPath ← {}
    dSeacrh ← []
    while length(frontier) > 0 do
        lev, currCell ← frontier.pop()
        dSeacrh.append(currCell)
        if currCell = m._goal then
            break
        end if
        poss ← 0
        if currCell in explored or lev>=level then
            continue
        end if
        explored.append(currCell)
        for d in 'ESNW' do
            if m.maze_map[currCell][d] = True then
                if d = 'E' then
                    child ← (currCell[0], currCell[1] + 1)
                if d = 'W' then
                    child ← (currCell[0], currCell[1] - 1)
                if d = 'N' then
                    child ← (currCell[0] - 1, currCell[1])
                if d = 'S' then
                    child = (currCell[0] + 1, currCell[1])
                poss ← poss + 1
                if child not in explored then
                    frontier.append([lev+1,child])
                    dfsPath[child] ← currCell
            end if
        if poss > 1 then
            m.markCells.append(currCell)
    end while
    fwdPath ← {}
    cell ← m._goal
    print(dfsPath)
    if cell in dfsPath then
        while cell != start do
```

```
            fwdPath[dfsPath[cell]] ← cell
            cell ← dfsPath[cell]
    else
        print("На такій глибині шлях LDFS знайти не може")
    end if
    return dSeacrh, dfsPath, fwdPath

function h(cell1, cell2)
    x1, y1 ← cell1
    x2, y2 ← cell2
    return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)


function aStar(maze m)
    open ← PriorityQueue()
    open.put((h(start, m._goal), h(start, m._goal), start))
    aPath ← {}
    g_score ← {row: float("inf") for row in m.grid}
    g_score[start] ← 0
    f_score ← {row: float("inf") for row in m.grid}
    f_score[start] ← h(start, m._goal)
    searchPath ← [start]
    while not open.empty() do
        currCell ← open.get()[2]
        searchPath.append(currCell)
        if currCell = m._goal then
            break
        for d in 'ESNW' do
            if m.maze_map[currCell][d] = True then
                if d = 'E' then
                    childCell ← (currCell[0], currCell[1] + 1)
                if d = 'W' then
                    childCell ← (currCell[0], currCell[1] - 1)
                if d = 'N' then
                    childCell ← (currCell[0] - 1, currCell[1])
                if d = 'S' then
                    childCell = (currCell[0] + 1, currCell[1])
            end if
                temp_g_score ← g_score[currCell] + 1
                temp_f_score ← temp_g_score + h(childCell, m._goal)

                if temp_f_score < f_score[childCell] then
                    aPath[childCell] ← currCell
```

```
                        g_score[childCell] ← temp_g_score
                        f_score[childCell] ← temp_g_score + h(childCell,
m._goal)
                        open.put((f_score[childCell], h(childCell, m._goal),
childCell))
        end while
        fwdPath ← {}
        cell ← m._goal
        while cell != start do
            fwdPath[aPath[cell]] ← cell
            cell ← aPath[cell]
        end while
        return searchPath, aPath, fwdPath
```

3.2 Програмна реалізація

3.2.1 Вихідний код

Main.py

```
from dfs import LDFS
from astar import aStar
from pyamaze import maze,agent,COLOR
#from datetime import datetime

n=int(input("Введіть довжину лабіринту:"))
m=int(input("Введіть ширину лабіринту:"))
myMaze=maze(n,m)
myMaze.CreateMaze(loopPercent=50)
p=int(input("Введіть максимальну глибину для LDFS:"))
searchPath,aPath,fwdPath=aStar(myMaze)
dSearch,dfsPath,fwdDFSPath=LDFS(myMaze,p)


a=agent(myMaze,footprints=True,color=COLOR.cyan,filled=True)
b=agent(myMaze,footprints=True,color=COLOR.yellow)
a1=agent(myMaze,footprints=True,color=COLOR.red,filled=True)
b1=agent(myMaze,footprints=True,color=COLOR.green,filled=True)
myMaze.tracePath({a:dSearch},delay=50)
myMaze.tracePath({b:searchPath},delay=50)
myMaze.tracePath({b1:fwdPath},delay=50)
myMaze.tracePath({a1:fwdDFSPath},delay=50)


myMaze.run()
```

astar.py

```
from queue import PriorityQueue


def h(cell1, cell2):
    x1, y1 = cell1
    x2, y2 = cell2
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** (1 / 2)
```

```
def aStar(m, start=None):
    end=0
    if start is None:
        start = (m.rows, m.cols)
    open = PriorityQueue()
    open.put((h(start, m._goal), h(start, m._goal), start))
    aPath = {}
    g_score = {row: float("inf") for row in m.grid}
    g_score[start] = 0
    f_score = {row: float("inf") for row in m.grid}
    f_score[start] = h(start, m._goal)
    searchPath = [start]
    counter=0
    while not open.empty():
        poss = 0
        counter+=1
        currCell = open.get()[2]
        searchPath.append(currCell)
        if currCell == m._goal:
            break
        for d in 'ESNW':
            if m.maze_map[currCell][d] == True:
                if d == 'E':
                    childCell = (currCell[0], currCell[1] + 1)
                elif d == 'W':
                    childCell = (currCell[0], currCell[1] - 1)
                elif d == 'N':
                    childCell = (currCell[0] - 1, currCell[1])
                elif d == 'S':
                    childCell = (currCell[0] + 1, currCell[1])
                if childCell not in searchPath:
                    poss+=1

                temp_g_score = g_score[currCell] + 1
                temp_f_score = temp_g_score + h(childCell, m._goal)

                if temp_f_score < f_score[childCell]:
                    aPath[childCell] = currCell
                    g_score[childCell] = temp_g_score
                    f_score[childCell] = temp_g_score + h(childCell, m._goal)
                    open.put((f_score[childCell], h(childCell, m._goal),
childCell))

        if poss==0:
            end+=1

    fwdPath = {}
    cell = m._goal
    while cell != start:
        fwdPath[aPath[cell]] = cell
        cell = aPath[cell]
    print("Кількість ітерацій A* = ",counter+1)
    print("Кількість глухих кутів A* =",end)
    print('A-Star Path Length =', len(fwdPath))
    print('Всього станів у A-Star =', len(searchPath))
    return searchPath, aPath, fwdPath
```

dfs.py

```
def LDFS(m, level, start=None):
    end=0
    if start is None:
        start = (m.rows, m.cols)
    explored = []
    frontier = [[0, start]]
    dfsPath = {}
    dSeacrh = []
    counter=0
    while len(frontier) > 0:
        counter+=1
        lev, currCell = frontier.pop()
        dSeacrh.append(currCell)
        if currCell == m._goal:
            break
        poss = 0
        if currCell in explored or lev >= level:
            continue
        explored.append(currCell)
        for d in 'ESNW':
            if m.maze_map[currCell][d] == True:
                if d == 'E':
                    child = (currCell[0], currCell[1] + 1)
                if d == 'W':
                    child = (currCell[0], currCell[1] - 1)
                if d == 'N':
                    child = (currCell[0] - 1, currCell[1])
                if d == 'S':
                    child = (currCell[0] + 1, currCell[1])
                if child not in explored:
                    poss += 1
                    frontier.append([lev + 1, child])
                    dfsPath[child] = currCell
        if poss > 1:
            m.markCells.append(currCell)
        if poss==0:
            end+=1
    fwdPath = {}
    cell = m._goal
    if cell in dfsPath:
        while cell != start:
            fwdPath[dfsPath[cell]] = cell
            cell = dfsPath[cell]
    else:
        print("На такій глибині шлях LDFS знайти не може")
    print("Кількість ітерацій LDFS = ", counter)
    print("Кількість глухих кутів LDFS =",end)
    print('LDFS Path Length =', len(fwdPath))
    print('Всього станів у LDFS =', len(dSeacrh))
    return dSeacrh, dfsPath, fwdPath
```

3.2.2 Приклади роботи

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

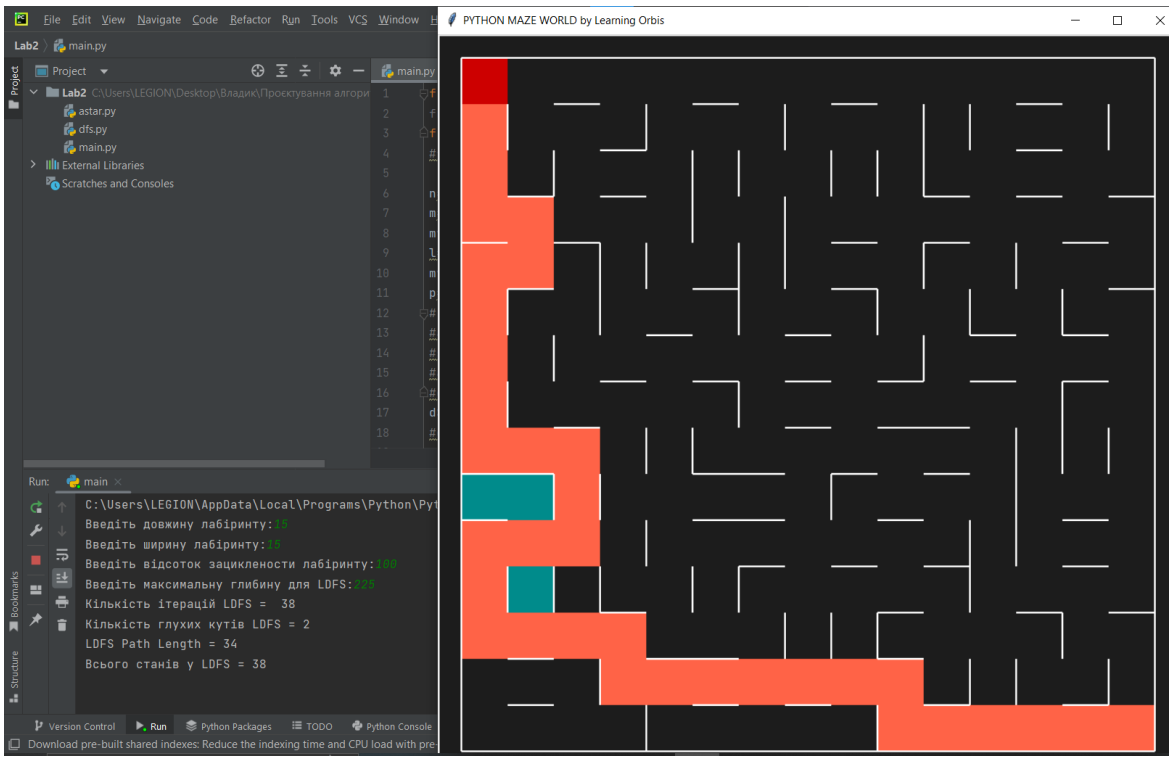

Рисунок 3.1 – Алгоритм LDFS для лабіринту 15*15

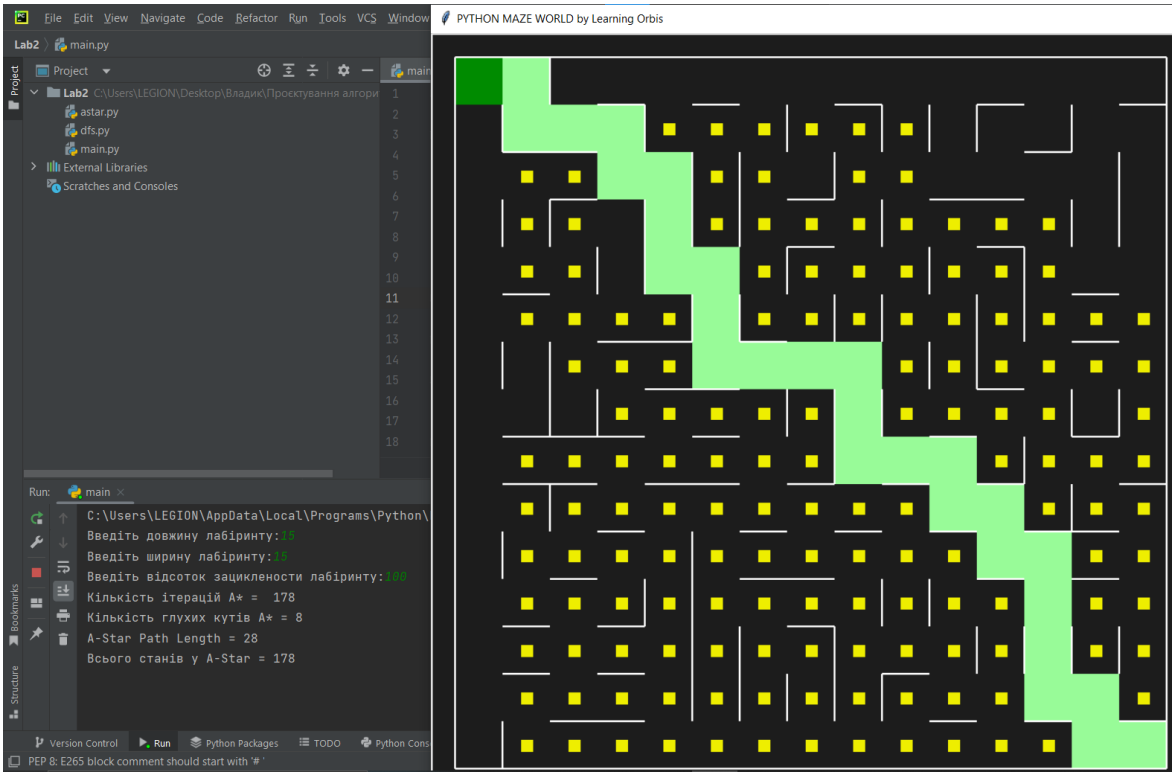

Рисунок 3.2 – Алгоритм A-Star

3.3 Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS для Лабіринту 20*20.

Початкові стани (генеруються програмою)	Всього станів	К-сть гл. кутів	Ітерації	Всього станів у пом'яті
Стан 1	55	2	52	18
Стан 2	96	8	82	38
Стан 3	58	1	56	28
Стан 4	66	3	56	40
Стан 5	74	6	44	25
Стан 6	57	3	48	24
Стан 7	75	3	68	28
Стан 8	51	1	48	23
Стан 9	54	1	52	30
Стан 10	68	2	64	35
Стан 11	51	0	50	21
Стан 12	46	1	44	23
Стан 13	46	1	42	22
Стан 14	75	3	68	33
Стан 15	64	2	60	33
Стан 16	90	4	80	34
Стан 17	64	4	42	22
Стан 18	50	1	46	22
Стан 19	80	2	74	30
Стан 20	57	3	52	22
Average	**64**	**3**	**56**	**28**

В таблиці 3.2 наведені характеристики оцінювання алгоритму A-Star, задачі Лабіринт для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання алгоритму A-Star для Лабіринту 20*20.

Початкові стани (генеруються програмою)	Всього станів	К-сть гл. кутів	Ітерації	Всього станів у пам’яті
Стан 1	224	25	38	29
Стан 2	388	54	44	34
Стан 3	257	29	38	33
Стан 4	386	53	44	33
Стан 5	288	31	40	35
Стан 6	317	37	40	26
Стан 7	311	44	42	35
Стан 8	311	35	40	27
Стан 9	266	24	40	33
Стан 10	260	27	40	30
Стан 11	259	23	40	29
Стан 12	273	34	40	34
Стан 13	238	17	38	37
Стан 14	211	16	38	36
Стан 15	378	43	42	42
Стан 16	324	39	40	33
Стан 17	227	22	38	32
Стан 18	332	27	42	30
Стан 19	245	28	38	32
Стан 20	251	31	38	30
Average	**287**	**32**	**40**	**33**

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми для пошуку шляху у довільному лабіринті від початкової точки до кінцевої. Було розроблено програмну реалізацію алгоритму неінформативного пошуку LDFS та алгоритму інформативного пошуку A-Star з використанням евристичної функції «Евклідова відстань».

LDFS - пошук з обмеженням у глибині доцільний для вирішення проблеми необмеженого шляху, яку можна виправити, наклавши обмеження на глибину області пошуку. Отож щоб уникнути статусу нескінченного циклу під час виконання коду, алгоритм пошуку розгортає якомога глибший вузол, який не перевищує заданої максимальної глибину. Даний алгоритм реалізується за допомогою структури даних черги.

Перевагою LDFS можна назвати те, що він потребує менше часу і пам’яті, аніж класичний DFS. Натомість недоліком пошуку з обмеженою глибиною є те, що цільовий вузол може не існувати в обмеженні глибини, встановленому раніше, що змусить користувача повторити подальшу ітерацію, додавши час виконання.

Алгоритм A * покроково переглядає всі шляхи, що ведуть від початкової вершини в кінцеву, поки не знайде мінімальний. При виборі вершини він враховує, крім іншого, весь пройдений до неї шлях. На початку роботи розглядаються вузли, суміжні з початковим; вибирається той з них, який має мінімальне значення f (n), після чого цей вузол розкривається. На кожному етапі алгоритм оперує з множиною шляхів з початкової точки до всіх ще не розкритих (листових) вершин графа - множиною часткових рішень, - яке розміщується в черзі з пріоритетом. Пріоритет шляху визначається за значенням f (n) = g (n) + h (n). Алгоритм продовжує свою роботу до тих пір, поки значення f (n) цільової вершини не виявиться меншим, ніж будь-яке значення в черзі, або поки все дерево не буде переглянуто. З множини рішень вибирається рішення з найменшою вартістю. Отож було застосовано структуру даних черга з пріоритетом.

У результаті розробленої програмної реалізації алгоритмів та аналізу ефективності на прикладі лабіринту 20*20, було обчислено такі значення:

1) У середньому шлях від початкової точки до кінцевої за допомогою алгоритму LDFS займав 56 клітинок, в той час як A* знаходив шлях на 40 клітинок. Отже, у середньому наочно переконуємося, що АІП A* знаходить ефективніший шлях розв'язання проблеми.
2) У середньому кількість глухих кутів в LDFS – 3 одиниці, в той час як A* - 32 клітинки. Отже, помічаємо, що в середньому LDFS менше заходить у глухі кути, аніж A*.
3) У середньому кількість станів, тобто кількість розглянутих клітинок, LDFS – 64 одиниці, A* - 287 одиниць. Отож A* аналізує набагато більший спектр різних можливих шляхів, аніж LDFS: саме тому закономірно впирається у більшу кількість глухих кутів.
4) У середньому кількість станів, що зберігається у пам'яті, для LDFS – 28, для A* - 33. Отже, алгоритм A* використовує більше пам'яті упродовж своєї ітерації.

Отже, алгоритм LDFS варто використовувати, коли евристика A* погана; якщо метою є оптимальність і евристика A* неприпустима; якщо рішення розташовані глибоко в структурі, що розглядається.

Натомість алгоритму A* варто надати перевагу, якщо структура є нескінченною або ж евристика є достатньо ефективна.

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%.