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 Ал Хадам Мурат* _____
(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·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

Функція A*:

queue = **PriorityQueue** ()

```
Поки open not queue.empty():
    curr = queue.get()
    Якщо curr == goal:
      break
    Все якщо

    Для кожного neighbor in expand повторити:
      temp_g_score = g_score[current] + 1
      temp_f_score = temp_g_score + self.h(neighbour, self.start)

      Якщо temp_f_score < f_score[neighbour]:
               g_score[neighbour] = temp_g_score
               f_score[neighbour] = temp_f_score
               queue.put((temp_f_score, self.h(neighbour,
                           self.start), neighbour))
               a_path[neighbour] = current
      Все якщо
    Все повторити
Все поки
```

Все функція

Функція LDFS

```
stack = stack()
stack.append((start, [start]))

Поки not stack.empty()
      curr, path = stack.pop()
      Якщо len(path) - 1 == limit:
          stops += 1
          continue
      Все якщо

      Якщо curr == goal:
          return
      Все якщо

      neighbours = []
      Для всіх dir in 'ESNW' повторити:
       Якщо self.m.maze_map[curr][dir] == 1:
           Якщо dir == 'E':
              neighbour = (curr[0], curr[1] + 1)
           Інакше якщо dir == 'W':
              neighbour = (curr[0], curr[1] - 1)
           Інакше якщо dir == 'N':
              neighbour = (curr[0] - 1, curr[1])
           Інакше:  # if dir == 'S':
              neighbour = (curr[0] + 1, curr[1])
             Все якщо
      Якщо neighbour not in path:
```

```
                neighbours.append((neighbour, path + [neighbour]))
            Все якщо

            stack.append(expand(m, curr, path))
```

Все поки

Все функція

3.2 Програмна реалізація

3.2.1 Вихідний код

main.py

```
from AlgoSolver import ALgoSolver
from pyamaze import maze, agent


def main():
    size = int(input("Enter size of the maze: "))
    loop_percent = int(input("Enter percent of possible loops:"))

    m = maze(size, size)
    m.CreateMaze(loopPercent=loop_percent)

    option = -1
    while option != 1 and option != 2:
        option = int(input("For A* algo pick 1,\n"
                           "For LDFS - pick 2:\n"))

    algo = ALgoSolver(m)
    if option == 1:
        algo.search_AStar()
    else:
        limit = int(input("Enter limit of depth or -1 for auto-limit: "))
        limit = limit if limit else size**2//2
        algo.search_LDFS(limit)
        print(f"Number of stops is {algo.stops}")
        if not algo.is_solvable:
            print(f"There is no solution with the {limit} depth.")

    print(f"Iterations: {algo.iterations}, unique states:
{algo.amount_of_states}")

    a = agent(m, shape='arrow', filled=True, footprints=True, color=algo.color)
    m.tracePath({a: algo.path}, delay=100)

    print(len(algo.path))

    m.run()


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

AlgoSolver.py

```
import psutil
import os
import func_timeout

from queue import PriorityQueue
from pyamaze import COLOR
from time import time


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

        self.stops = 0
        self.states = []
        self.amount_of_states = 0

        self.iterations = 0
        self.start_time = time()

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

        self.is_solvable = False
        self.path = {}

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

    def __del__(self):
        print(f"{self.name} algo finised in {time() - self.start_time}
seconds...")
        print(f"{psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2} MB
used...")

    def search_AStar(self):
        return func_timeout.func_timeout(60 * 30, self.__AStar)

    def __AStar(self):
        self.name = "A*"

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

        f_score = {
            cell: float('inf')
            for cell in self.m.grid
        }
        f_score[self.start] = 0 + self.h(self.start, self.goal)

        queue = PriorityQueue()
        queue.put((f_score[self.start], self.h(self.start, self.goal),
self.start))

        a_path = {}
        while not queue.empty():
            if psutil.Process(os.getpid()).memory_info().rss > 1024 ** 3:
                raise MemoryError("1 Gb memory exceeded")

            self.iterations += 1
            current = queue.get()[2]
```

```
            if current == self.goal:
                self.amount_of_states = len(self.states)
                break

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

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

                    temp_g_score = g_score[current] + 1
                    temp_f_score = temp_g_score + self.h(neighbour, self.start)

                    if temp_f_score < f_score[neighbour]:
                        g_score[neighbour] = temp_g_score
                        f_score[neighbour] = temp_f_score
                        queue.put((temp_f_score, self.h(neighbour, self.start),
neighbour))
                        a_path[neighbour] = current

        cell = self.goal
        while cell != self.start:
            self.path[a_path[cell]] = cell
            cell = a_path[cell]

    # manhattan distance
    @staticmethod
    def h(point_a: tuple, point_b: tuple) -> int:
        return abs(point_a[0] - point_b[0]) + abs(point_a[1] - point_b[1])

    def search_LDFS(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 memory exceeded")

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

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

            if curr == self.goal:
                self.amount_of_states = len(self.states)
                self.is_solvable = True
                break
```

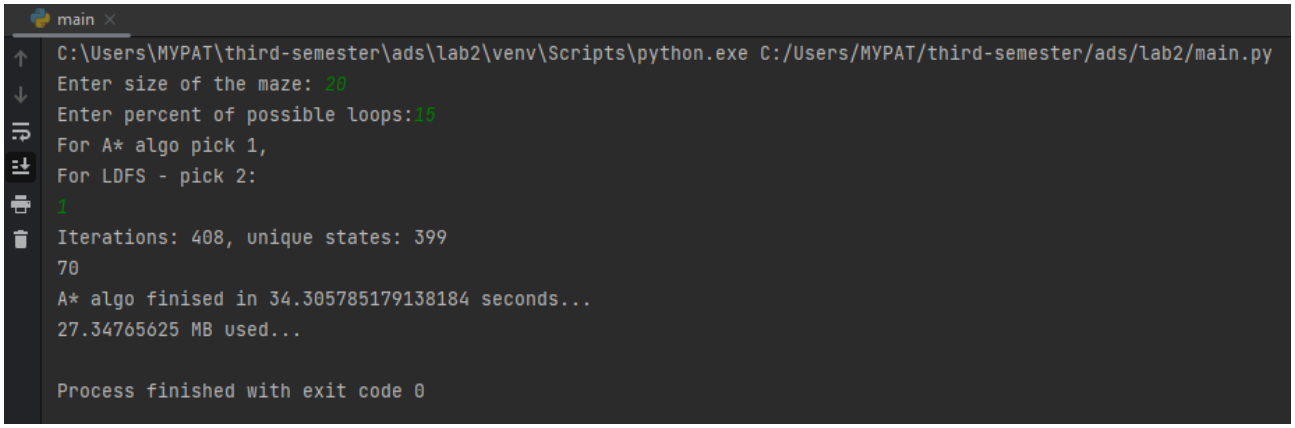
```
            neighbours = []
            for dir in 'ESNW':
                if self.m.maze_map[curr][dir] == 1:
                    if dir == 'E':
                        neighbour = (curr[0], curr[1] + 1)
                    elif dir == 'W':
                        neighbour = (curr[0], curr[1] - 1)
                    elif dir == 'N':
                        neighbour = (curr[0] - 1, curr[1])
                    else:  # if dir == 'S':
                        neighbour = (curr[0] + 1, curr[1])

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

            stack += neighbours
```

3.2.2 Приклади роботи

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



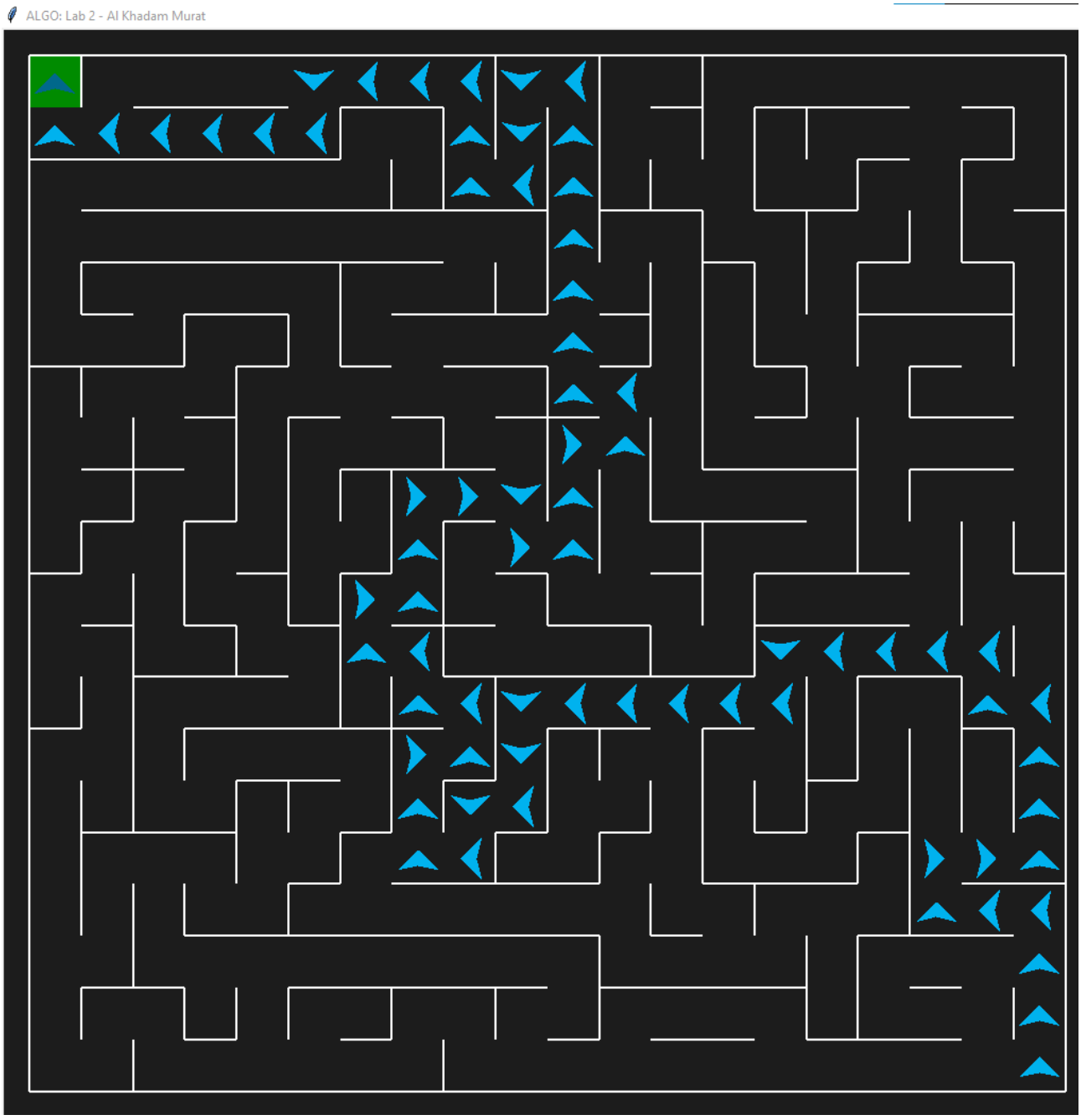


Рисунок 3.1 – Алгоритм A*

```
main
C:\Users\MYPAT\third-semester\ads\lab2\venv\Scripts\python.exe C:/Users/MYPAT/third-semester/ads/lab2/main.py
Enter size of the maze: 20
Enter percent of possible loops:20
For A* algo pick 1,
For LDFS - pick 2:
2
Enter limit of depth or -1 for auto-limit: -1
Number of stops is 0
Iterations: 96, unique states: 82
52
LDFS algo finised in 32.792749881744385 seconds...
27.39453125 MB used...

Process finished with exit code 0
```

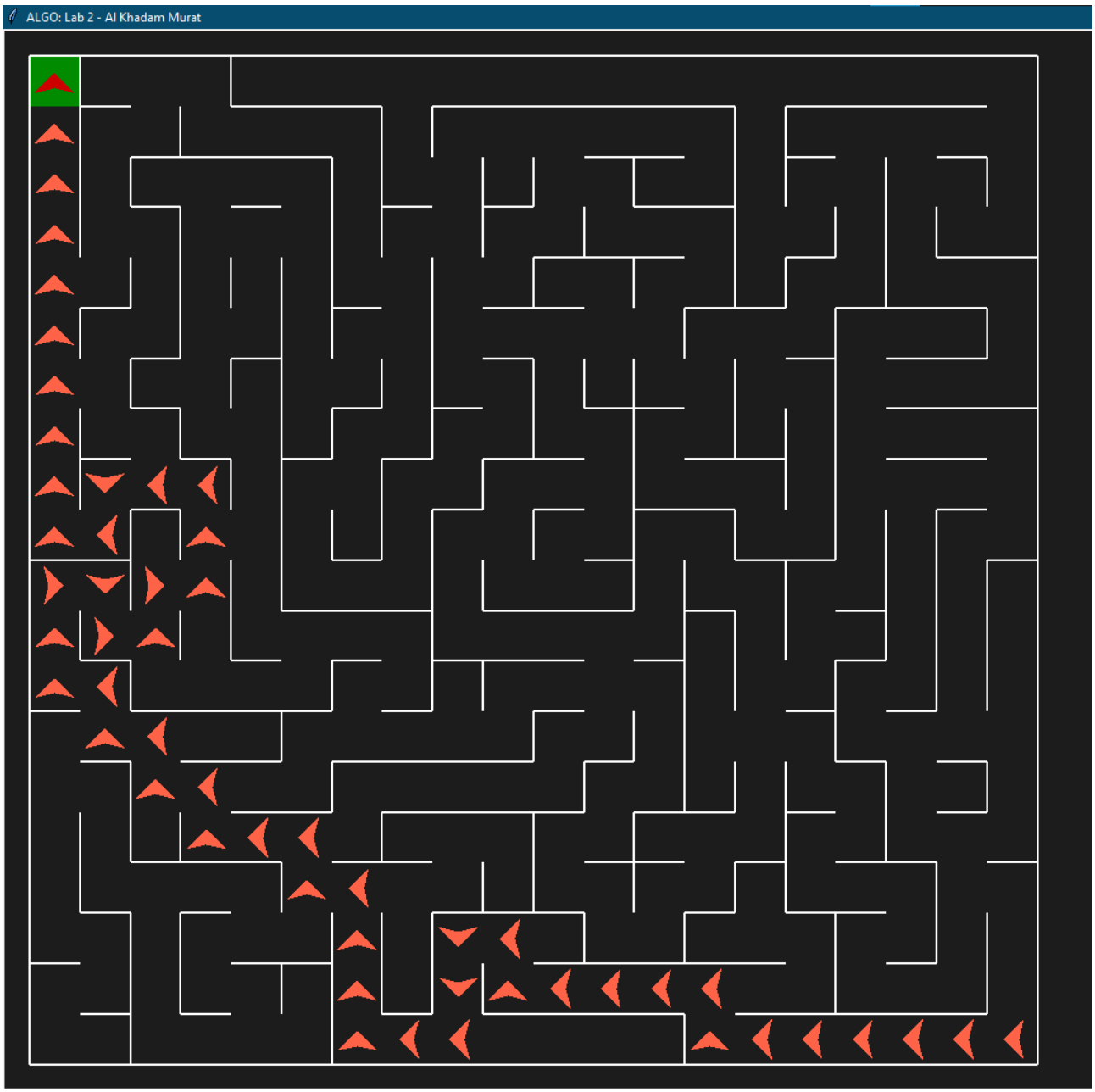


Рисунок 3.2 – Алгоритм LDFS

3.3 Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом’яті
Стан 1	2562	0	386	21
Стан 2	337	0	337	14
Стан 3	100	13	145	11
Стан 4	256	0	152	86
Стан 5	327	0	327	132
Стан 6	1542	199	135	13
Стан 7	54	4	122	13
Стан 8	410	0	399	33
Стан 9	182	0	182	26
Стан 10	71	0	71	15
Стан 11	143	0	143	21
Стан 12	415	0	351	34
Стан 13	215	0	199	15
Стан 14	15224	1044	342	36
Стан 15	15	2	9	9
Стан 16	200	0	200	27
Стан 17	183	0	183	14
Стан 18	166	1	122	24
Стан 19	356	0	356	14
Стан 20	6502	621	396	35

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

Таблиця 3.3 – Характеристики оцінювання алгоритму A*

Початкові стани	Ітерації	Всього станів	Всього станів у пом’яті
Стан 1	336	329	48
Стан 2	409	399	60
Стан 3	408	399	58
Стан 4	410	399	46
Стан 5	411	399	50
Стан 6	410	399	38
Стан 7	413	399	54
Стан 8	257	255	60
Стан 9	37	35	14
Стан 10	25	24	4
Стан 11	372	359	95
Стан 12	403	399	126
Стан 13	405	398	104
Стан 14	156	124	16
Стан 15	359	359	36
Стан 16	450	399	59
Стан 17	123	120	10
Стан 18	262	249	29
Стан 19	212	168	9
Стан 20	121	99	5

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми пошуку в глибину з обмеженням глибини та A* для задачі лабіринту, було здійснено програмну реалізацію цих алгоритмів. Було здійснено 20 експерементів для кожного із алгоритмів і зафіксовано кількість ітерацій, кількість пройдених станів та максимальну кількість станів у пам’яті, чим виступала максимальна кількість елементів у черзі.

Очевидно, що A* здійснює менше ітерацій та завжди знаходить найкоротший шлях у лабіринті, але розгортає більше станів. Алгоритм має еврестичну функцію – Манхетенську відстань – сума модулів різниць координат.

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%.