**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)**

(шифр, прізвище, ім'я, по батькові)

*ІП-13 Ал Хадам Мурат*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc81070688)

[2 ЗаВдання 4](#_Toc81070689)

[3 Виконання 8](#_Toc81070690)

[3.1 Псевдокод алгоритмів 8](#_Toc81070691)

[3.2 Програмна реалізація 8](#_Toc81070692)

[3.2.1 Вихідний код 8](#_Toc81070693)

[3.2.2 Приклади роботи 8](#_Toc81070694)

[3.3 Дослідження алгоритмів 8](#_Toc81070695)

[Висновок 11](#_Toc81070696)

[Критерії оцінювання 12](#_Toc81070697)

# Мета лабораторної роботи

Мета роботи – розглянути та дослідити алгоритми неінформативного, інформативного та локального пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

# 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 А\*.
* **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
* **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь А може стояти на одній лінії з ферзем В, проте між ними стоїть ферзь С; тому А не б’є В).
* **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 Т 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 |

# Виконання

## Псевдокод алгоритмів

**Функція** А\*:

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**

## Програмна реалізація

### Вихідний код

main.py

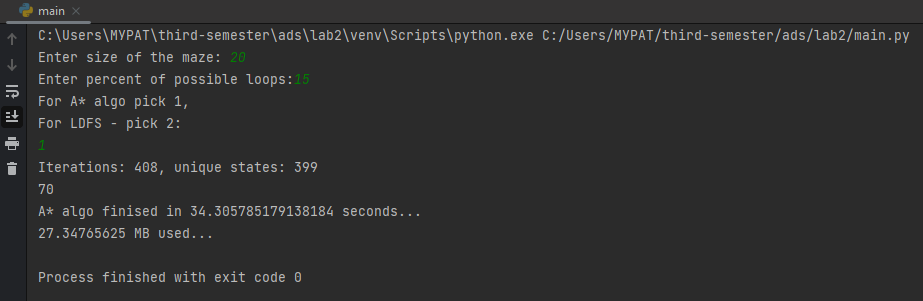
**from** AlgoSolver **import** ALgoSolver  
**from** pyamaze **import** maze, agent  
  
  
**def** main():  
 size = int(input("Enter size of the maze: "))  
 loop\_percent = int(input("Enter percent of possible loops:"))  
  
 m = maze(size, size)  
 m.CreateMaze(loopPercent=loop\_percent)  
  
 option = -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

### Приклади роботи

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



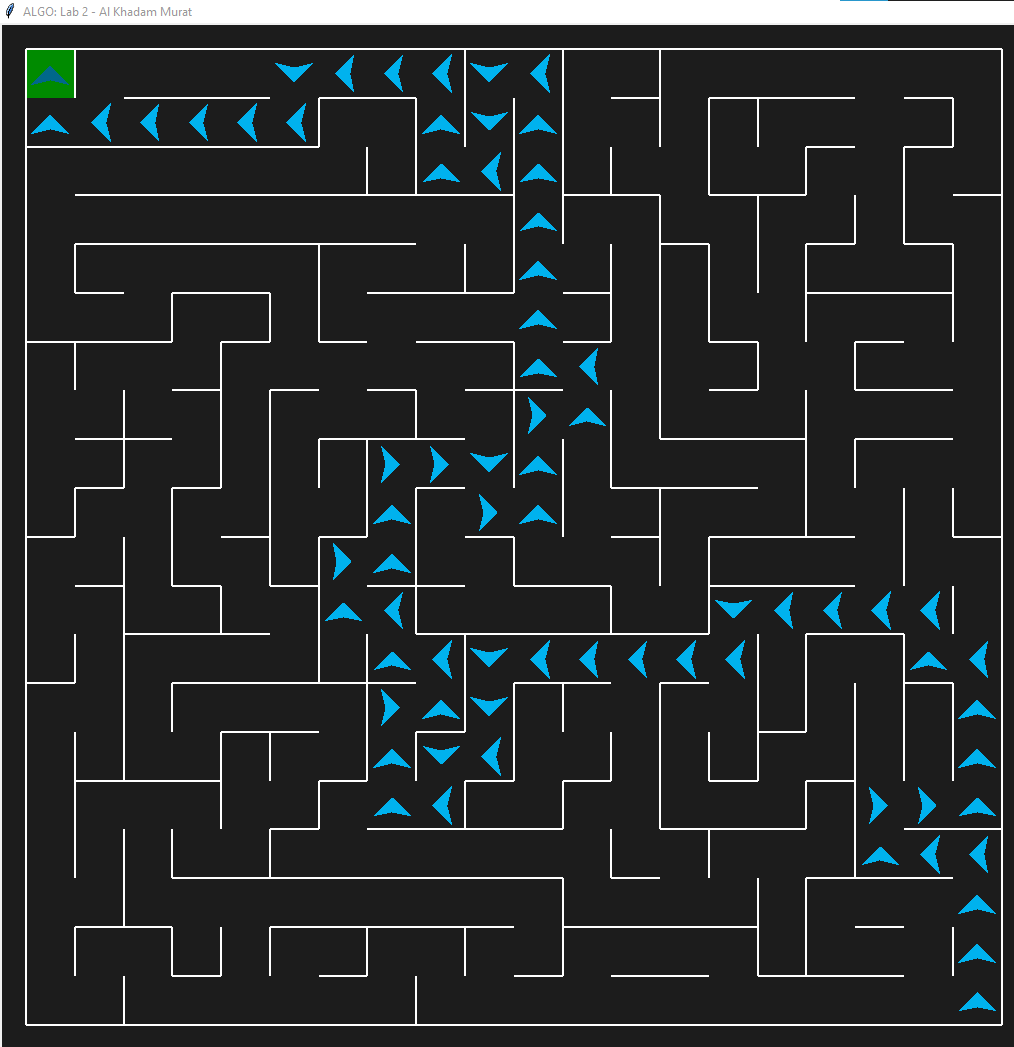
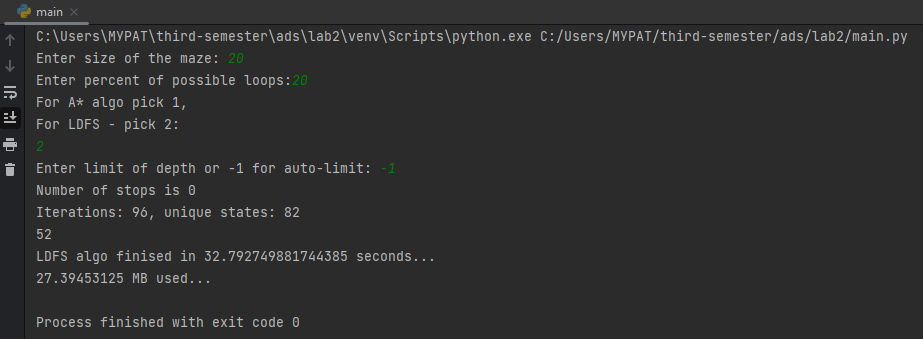


Рисунок 3.1 – Алгоритм A\*



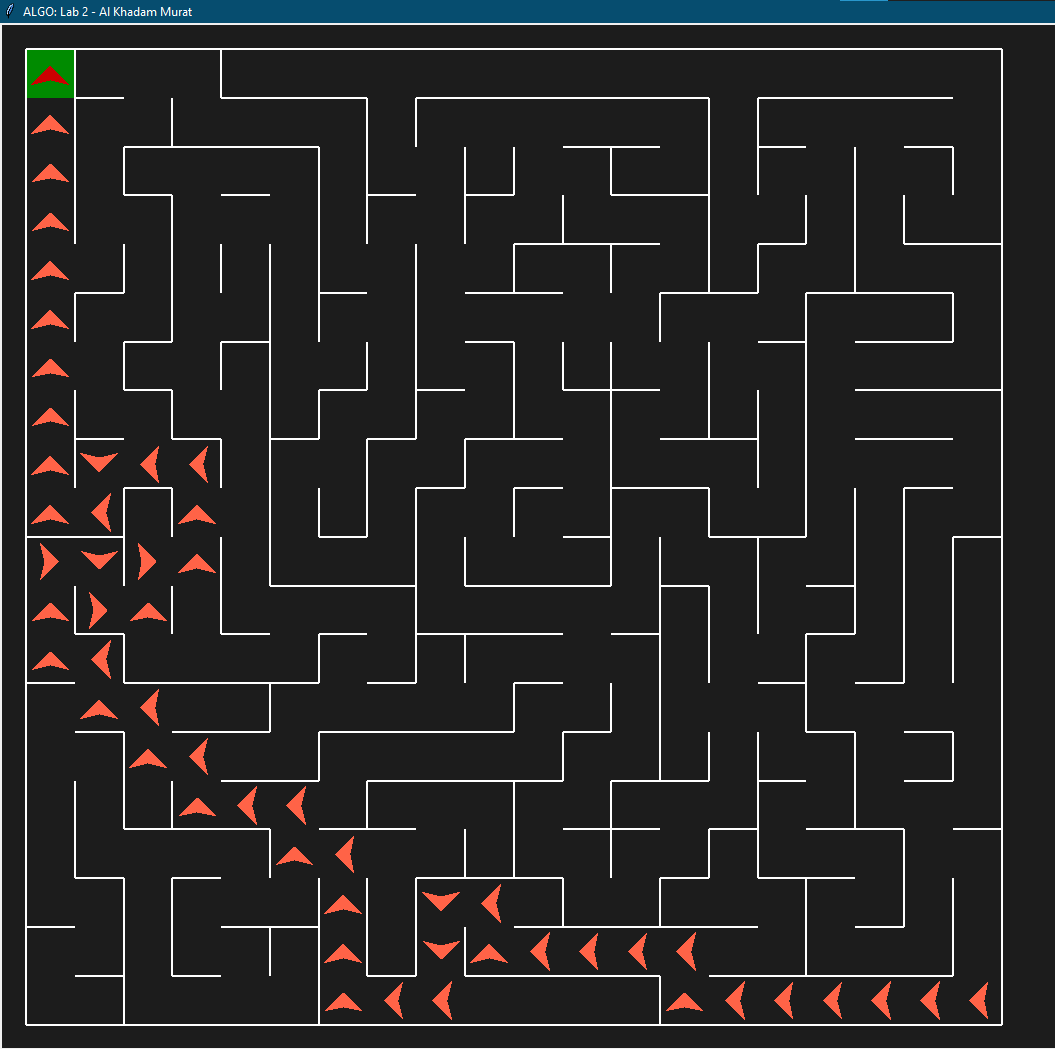


Рисунок 3.2 – Алгоритм LDFS

## 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 А\* 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 А\* 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%.