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-12 Кушнір Ганна Вікторівна* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олексій Олександрович* _____
(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

3.1.1 Псевдокод алгоритму IDS

IDS(root)

```
depth ← 0
while true do
        result ← DLS(root, depth)
        if result is a goal state
                then    return result
        end if
        depth ← depth + 1
end while
```

DLS(node, depth)

```
if depth = 0 and result is a goal state
        then    return node
        else    if depth > 0
                        then    for every child in Expand(node) do
                                        result ← DLS(child, depth – 1)
                                        if result is a Node
                                                then    return result
                                        end if
                                end for
                        else    return None
                end if
end if
```

Expand(node)

```
newNode ← node
for i from 0 to 8 do
        for j from 1 to 8 do
                child ← node whose queens are repositioned so that queen number "i" is moved "j" positions down
                add child to newNode’s children
        end for
end for
return newNode
```

3.1.2 Псевдокод алгоритму A*

A*(root)

```
Closed ← empty list
Open ← empty priority queue
Open.push(root)
g[root] ← 0
f[root] ← g[root] + H(root)
while Open.size != 0 do
        current ← Open.pop_min()
        if current is a goal state
                then    return current
        end if
        Closed.push_back(current)
        for every child in Expand(current) do
                pathCost ← g[current] + 1
                        if child is in Closed and pathCost ≥ g[child]
                                then    continue
                        end if
                        if child is not in Closed or pathCost < g[child]
                                then    add child to current's children
                                        g[child] ← pathCost
                                        f[child] ← g[child] + H(child)
                                        if child is not in Open
                                                then    Open.push(child)
                                        end if
                        end if
        end for
end while
```

H(node)

```
conflicts ← 0
for i from 0 to 8 do
        for j from i + 1 to 8 do
                if node.queens[i] = node.queens[j]
                        then    conflicts ← conflicts + 1
                                for k from i + 1 to j do
                                        if node.queens[i] = node.queens[k]
                                                then    conflicts ← conflicts – 1
                                                        break
                                        end if
                                end for
```

```
                end if
                if node.queens[i] = node.queens[j] + j - i
                        then    conflicts ← conflicts + 1
                                for k from i + 1 to j do
                                        if node.queens[i] = node.queens[k] + k - i
                                                then    conflicts ← conflicts – 1
                                                        break
                                        end if
                                end for
                end if
                if node.queens[i] = node.queens[j] - j + i
                        then    conflicts ← conflicts + 1
                                for k from i + 1 to j do
                                        if node.queens[i] = node.queens[k] - k + i
                                                then    conflicts ← conflicts – 1
                                                        break
                                        end if
                                end for
                end if
        end for
    end for
    return conflicts
```

Expand(node)

```
    newNode ← node
    for i from 0 to 8 do
        for j from 1 to 8 do
            child ← node whose queens are repositioned so that queen number "i" is moved "j" positions down
            add child to newNode’s children
        end for
    end for
    return newNode
```

3.2 Програмна реалізація

3.2.1 Вихідний код

3.2.1.1 Вихідний код програмної реалізації алгоритму IDS

Файл «Node.py»:

```
NUM = 8
```

```
class Node:
    def __init__(self, queens: list):
        self.queens = queens
        self.childs = []

    def addChild(self, i: int, j: int):
        queens = []
        for k in range(NUM):
             queens.append(self.queens[k])
        queens[i] = (queens[i] + j) % NUM
        self.childs.append(Node(queens))
        return
```

Файл «IDFS.py»:

```
from Node import *

def isAGoalState(queens: list):
    for i in range(NUM):
        for j in range(i + 1, NUM):
            if queens[i] == queens[j] or queens[i] == queens[j] + j -
i or queens[i] == queens[j] - j + i:
                return False
    return True

def expand(node: Node):
    newNode = Node(node.queens)
    for i in range (NUM):
        for j in range (1, NUM):
            newNode.addChild(i, j)
    return newNode

def IDFS(root: Node):
    depth = 0
    while True:
        result = DLS(root, depth)
        if result != None and isAGoalState(result.queens):
            return result, depth
        depth += 1

def DLS(node: Node, depth: int):
    if depth == 0 and isAGoalState(node.queens):
        return node
    elif depth > 0:
        for child in expand(node).childs:
            result = DLS(child, depth - 1)
```

```
            if result != None:
                return result
    else:
        return None
```

Файл «main.py»:

```
from IDFS import *
import time

def printBoard(queens: list):
    print('┌──┬──┬──┬──┬──┬──┬──┬──┐')
    for i in range(8):
        print('│' + '  │'* queens[i] + ' * │' + '  │'* (7 - queens[i]))
        if i != 7:
            print('├──┼──┼──┼──┼──┼──┼──┼──┤')
        else:
            print('└──┴──┴──┴──┴──┴──┴──┴──┘')

if __name__ == "__main__":
    queens = [0, 6, 3, 3, 5, 0, 4, 0]
    print('Entry positions of queens: ', queens)
    print('The input state of the checkerboard:')
    printBoard(queens)
    root = Node(queens)
    start_time = time.time()
    result, depth = IDFS(root)
    end_time = time.time()
    print('Solution: ', result.queens)
    print('The resulting target state of the checkerboard:')
    printBoard(result.queens)
    print('Iterations:', depth)
    print('Time spent: %0.5f seconds' % (end_time - start_time))
```

3.2.1.2 Вихідний код програмної реалізації алгоритму A*

Файл «Node.py»:

```
NUM = 8
B = NUM * (NUM - 1)
```

```
D = 1

class Node:
    def __init__(self, queens: list, g: int = 0):
        self.queens = queens
        self.childs : list[Node] = []
        self.h = self.countConflicts()
        self.g = g
        self.f = self.g + self.h

    def addChildByChangingParent(self, i: int, j: int):
        queens = []
        for k in range(NUM):
             queens.append(self.queens[k])
        queens[i] = (queens[i] + j) % NUM
        self.childs.append(Node(queens, self.g + D))
        return

    def addChild(self, queens: list):
        self.childs.append(Node(queens, self.g + D))

    def countConflicts(self):
        conflicts = 0
        for i in range(NUM):
            for j in range(i + 1, NUM):
                if self.queens[i] == self.queens[j]:
                    conflicts += 1
                    for k in range(i + 1, j):
                        if self.queens[i] == self.queens[k]:
                            conflicts -= 1
                            break
                if self.queens[i] == self.queens[j] + j - i:
                    conflicts += 1
                    for k in range(i + 1, j):
                        if self.queens[i] == self.queens[k] + k - i:
                            conflicts -= 1
                            break
                if self.queens[i] == self.queens[j] - j + i:
                    conflicts += 1
                    for k in range(i + 1, j):
                        if self.queens[i] == self.queens[k] - k + i:
                            conflicts -= 1
                            break
        return conflicts
```

Файл «MinHeap.py»:

```
import sys
from Node import *
class MinHeap:
    def __init__(self, maxsize):
        self.maxsize = maxsize
        self.size = 0
        self.Heap : list[Node] = [None]*(self.maxsize + 1)
        self.Heap[0] = -1 * sys.maxsize
        self.FRONT = 1

    def parent(self, pos):
        return pos//2

    def leftChild(self, pos):
        return 2 * pos

    def rightChild(self, pos):
        return (2 * pos) + 1

    def isLeaf(self, pos):
        return pos*2 > self.size

    def swap(self, fpos, spos):
        self.Heap[fpos],    self.Heap[spos]    =    self.Heap[spos],
self.Heap[fpos]

    def minHeapify(self, pos):
        if not self.isLeaf(pos):
            if (self.Heap[pos].f > self.Heap[self.leftChild(pos)].f or
               self.Heap[pos].f > self.Heap[self.rightChild(pos)].f):
                if         self.Heap[self.leftChild(pos)].f         <
self.Heap[self.rightChild(pos)].f:
                    self.swap(pos, self.leftChild(pos))
                    self.minHeapify(self.leftChild(pos))
                else:
                    self.swap(pos, self.rightChild(pos))
                    self.minHeapify(self.rightChild(pos))

    def insert(self, element: Node):
        if self.size >= self.maxsize :
            self.updateMaxsize(self.maxsize)
        self.size+= 1
        self.Heap[self.size] = element
        current = self.size
```

```
            while self.parent(current) != 0 and self.Heap[current].f <
self.Heap[self.parent(current)].f:
                self.swap(current, self.parent(current))
                current = self.parent(current)

        def minHeap(self):
            for pos in range(self.size//2, 0, -1):
                self.minHeapify(pos)

        def remove(self):
            popped = self.Heap[self.FRONT]
            self.Heap[self.FRONT] = self.Heap[self.size]
            self.size -= 1
            self.minHeapify(self.FRONT)
            return popped

        def updateMaxsize(self, add):
            self.maxsize = self.maxsize + add
            for i in range(add):
                self.Heap.append(None)

        def __getitem__(self, pos):
            return self.Heap[pos]
```

Файл «A_star.py»:

```
from MinHeap import *

def expand(node: Node):
    newNode = Node(node.queens, node.g)
    for i in range (NUM):
        for j in range (1, NUM):
            newNode.addChildByChangingParent(i, j)
    return newNode

def A_star(root: Node):
    closed = []
    opened = MinHeap(B)
    opened.insert(root)
    while opened.size != 0:
        current = opened.remove()
        if current.h == 0:
            return current
        closed.append(current)
        for child in expand(current).childs:
            pathCost = current.g + D
```

```
                if child in closed and pathCost >= child.g:
                    continue
                if not child in closed or pathCost < child.g:
                    current.addChild(child.queens)
                    if not child in opened:
                        opened.insert(child)
        return None
```

Файл «main.py»:

```
from A_star import *
import time

def printBoard(queens: list):
    print('┌───┬───┬───┬───┬───┬───┬───┬───┐')
    for i in range(8):
        print('│' + '   │' * queens[i] + ' * │' + '   │' * (7 -
queens[i]))
        if i != 7:
            print('├───┼───┼───┼───┼───┼───┼───┼───┤')
        else:
            print('└───┴───┴───┴───┴───┴───┴───┴───┘')


if __name__ == "__main__":
    queens = [0, 0, 0, 5, 7, 0, 4, 0]
    print('Entry positions of queens: ', queens)
    print('The input state of the checkerboard:')
    printBoard(queens)
    root = Node(queens)
    start_time = time.time()
    result = A_star(root)
    end_time = time.time()
    if result != None:
        print('Solution: ', result.queens)
        print('The resulting target state of the checkerboard:')
        printBoard(result.queens)
    else:
        print('Solution was not found')
    print('Time spent: %0.5f seconds' % (end_time - start_time))
```

3.2.2 Приклади роботи

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

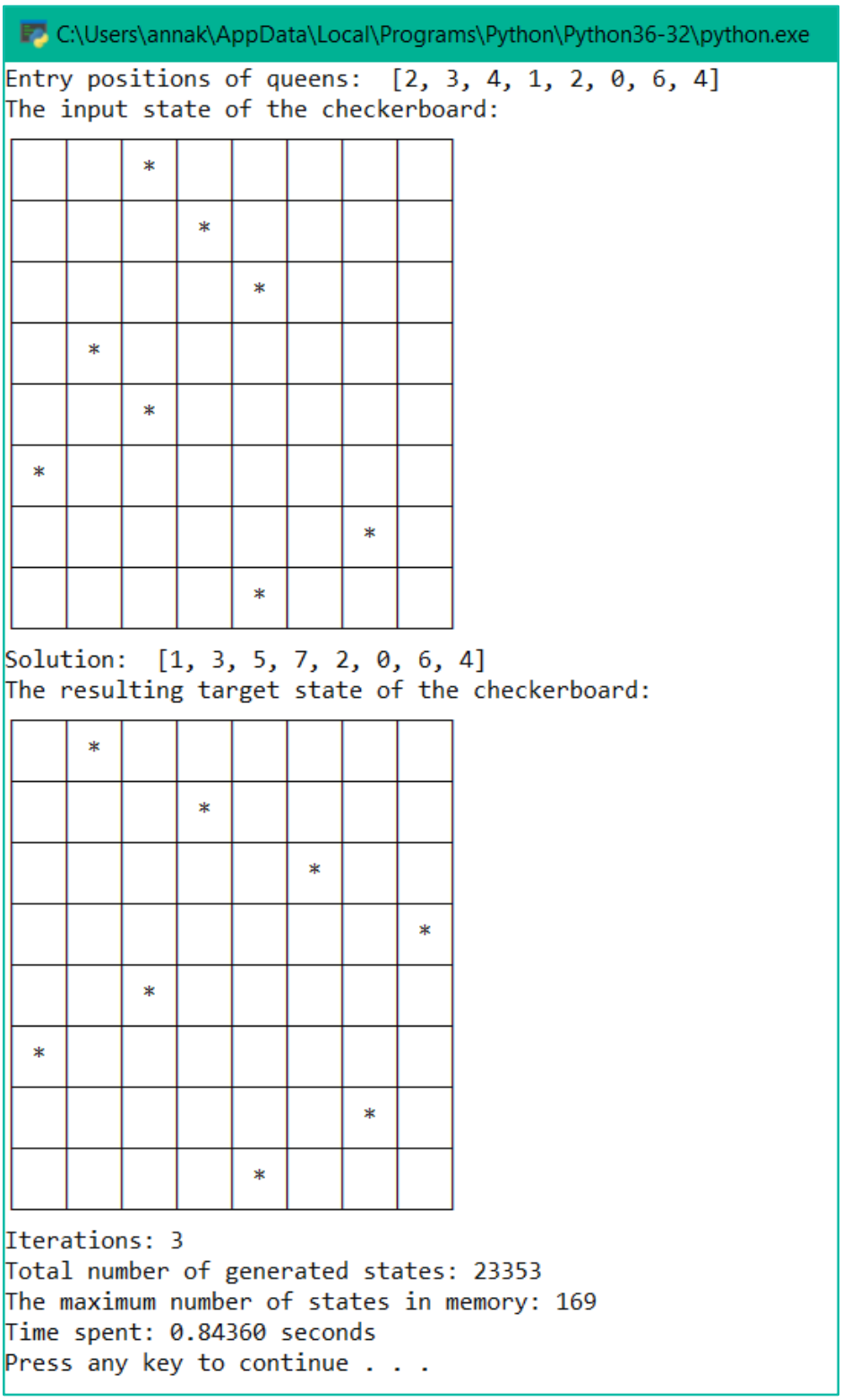

Рисунок 3.1 – Алгоритм IDS

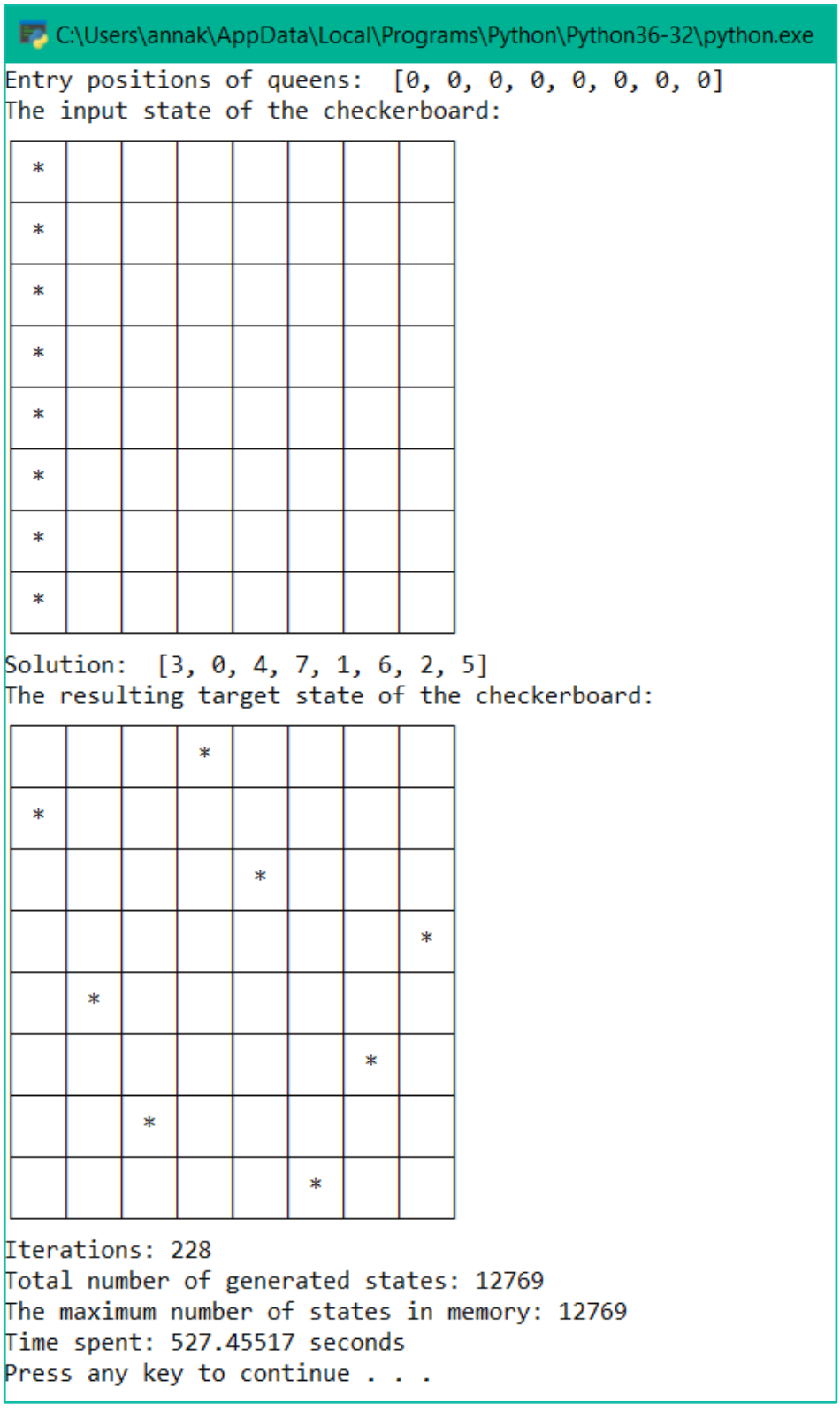

Рисунок 3.2 – Алгоритм A*

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму IDS, задачі «8 ферзів» для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму IDS

Початкові стани	Ітерації	Глухі кути	Всього станів	Максимум станів у пам’яті
Стан 1 [4, 6, 7, 3, 1, 7, 5, 2]	1	–	57	57
Стан 2 [5, 2, 0, 0, 3, 1, 6, 4]	1	–	57	57
Стан 3 [3, 0, 3, 7, 0, 6, 2, 5]	2	–	953	113
Стан 4 [2, 3, 4, 1, 2, 0, 6, 4]	3	–	23 353	169
Стан 5 [4, 0, 4, 5, 7, 3, 0, 6]	3	–	26 881	169
Стан 6 [2, 0, 6, 4, 7, 0, 0, 0]	3	–	117 601	169
Стан 7 [7, 5, 3, 6, 1, 3, 5, 2]	4	–	205 409	225
Стан 8 [5, 6, 2, 5, 5, 3, 1, 5]	4	–	209 049	225
Стан 9 [4, 0, 2, 4, 5, 2, 1, 7]	4	–	436 465	225
Стан 10 [0, 1, 4, 2, 5, 7, 5, 3]	4	–	590 241	225
Стан 11 [0, 6, 2, 1, 1, 7, 3, 5]	4	–	625 073	225
Стан 12 [0, 0, 0, 0, 0, 1, 4, 2]	4	–	927 305	225
Стан 13 [0, 2, 2, 1, 3, 4, 4, 3]	4	–	953 513	225

Продовження таблиці 3.1

Початкові стани	Ітерації	Глухі кути	Всього станів	Максимум станів у пам’яті
Стан 14 [6, 4, 2, 0, 7, 5, 3, 1]	4	–	1 102 921	225
Стан 15 [5, 3, 1, 6, 2, 7, 4, 0]	4	–	1 294 273	225
Стан 16 [2, 3, 5, 7, 4, 3, 2, 0]	4	–	1 489 769	225
Стан 17 [3, 7, 6, 3, 1, 4, 2, 6]	4	–	2 377 033	225
Стан 18 [5, 5, 1, 1, 7, 7, 2, 2]	5	–	12 416 713	281
Стан 19 [1, 4, 5, 3, 1, 3, 2, 6]	5	–	21 512 849	281
Стан 20 [0, 0, 0, 0, 0, 0, 0, 0]	–	Перевищено ліміт часу	–	–

В таблиці 3.2 наведені характеристики оцінювання алгоритму A*, задачі «8 ферзів» для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання алгоритму A*

Початкові стани	Ітерації	Глухі кути	Всього станів	Максимум станів у пам’яті
Стан 1 [4, 6, 7, 3, 1, 7, 5, 2]	1	–	57	57
Стан 2 [5, 2, 0, 0, 3, 1, 6, 4]	1	–	57	57
Стан 3 [3, 0, 3, 7, 0, 6, 2, 5]	2	–	113	113

Продовження таблиці 3.2

Початкові стани	Ітерації	Глухі кути	Всього станів	Максимум станів у пам’яті
Стан 4 [2, 3, 4, 1, 2, 0, 6, 4]	4	–	225	225
Стан 5 [4, 0, 4, 5, 7, 3, 0, 6]	4	–	225	225
Стан 6 [2, 0, 6, 4, 7, 0, 0, 0]	4	–	225	225
Стан 7 [7, 5, 3, 6, 1, 3, 5, 2]	36	–	2017	2017
Стан 8 [5, 6, 2, 5, 5, 3, 1, 5]	4	–	225	225
Стан 9 [4, 0, 2, 4, 5, 2, 1, 7]	4	–	225	225
Стан 10 [0, 1, 4, 2, 5, 7, 5, 3]	16	–	897	897
Стан 11 [0, 6, 2, 1, 1, 7, 3, 5]	12	–	673	673
Стан 12 [0, 0, 0, 0, 0, 1, 4, 2]	5	–	281	281
Стан 13 [0, 2, 2, 1, 3, 4, 4, 3]	5	–	281	281
Стан 14 [6, 4, 2, 0, 7, 5, 3, 1]	35	–	1 961	1 961
Стан 15 [5, 3, 1, 6, 2, 7, 4, 0]	247	–	13 833	13 833
Стан 16 [2, 3, 5, 7, 4, 3, 2, 0]	28	–	1 569	1 569

Продовження таблиці 3.2

Початкові стани	Ітерації	Глухі кути	Всього станів	Максимум станів у пам’яті
Стан 17 [3, 7, 6, 3, 1, 4, 2, 6]	34	–	1 905	1 905
Стан 18 [5, 5, 1, 1, 7, 7, 2, 2]	149	–	8 345	8 345
Стан 19 [1, 4, 5, 3, 1, 3, 2, 6]	–	Перевищено ліміт часу	–	–
Стан 20 [0, 0, 0, 0, 0, 0, 0, 0]	228	–	12 769	12 769

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми неінформативного, інформативного та локального пошуку. Було детально досліджено алгоритм пошуку вглиб з ітеративним заглибленням (DFS), який належить до алгоритмів неінформативного пошуку, а також алгоритм пошуку A*, який належить до алгоритмів інформативного пошуку.

На основі проведених досліджень було написано псевдокоди вищезгаданих алгоритмів. За побудованими псевдокодами було здійснено їх програмну реалізацію мовою програмування Python.

Було проведено дві серії випробувань по 20 випробувань для кожного алгоритму. За результатами експериментів можна зробити наступні висновки:

1) Обидва побудовані алгоритми є *повними*, оскільки завжди знаходять розв'язок задачі, але в деяких випадках пошук вимагає більше часу, ніж дозволено згідно з завданням лабораторної роботи (більше 30 хвилин).
2) Обидва побудовані алгоритми є *оптимальними*, оскільки при переважній більшості початкових станів цільовий стан знаходиться менш, ніж за декілька секунд.
3) *Часова складність* алгоритмів:
   - Часова складність алгоритму IDS: $O(b^d) = O(56^d)$, де b – коефіцієнт розгалуження (b = 56), d – глибина найбільш поверхневого цільового вузла ($0 \leq d \leq 7$). У випадку з алгоритмом IDS, d відповідає кількості ітерацій алгоритму.
   - Часова складність алгоритму A*: час роботи алгоритму експоненційно залежить від довжини розв'язку.
4) *Просторова складність* алгоритмів:
   - Просторова складність алгоритму IDS: $O(bd) = O(56 \times d)$, тобто цей алгоритм використовує лінійний простір.
   - Просторова складність алгоритму A*: алгоритм зберігає всі згенеровані вузли в оперативній пам'яті.

КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 30.10.2022 включно максимальний бал дорівнює – 5. Після 30.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 10%;
- програмна реалізація алгоритму – 60%;
- дослідження алгоритмів – 25%;
- висновок – 5%.