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 Карамян В. С.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов О.О.* _____
(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

```
BFS(initial_state):
    q = Queue()
    q.push(initial_state)
    while ( !q.empty() ):
        if time >= 1800:
            return False
        curr_state = q.pop()
        if curr_state.is_solution == True:
            return curr_state
        curr_depth = curr_state.get_depth()
        if curr_depth != n:
            for i from 0 to n:
                new_state = State(curr_state, curr_depth)
                new_state.move(curr_depth, i)
                q.push(new_state)
```

```
A*(initial_state):
    q = PriorityQueue()
    q.push(initial_state)
    while ( !q.empty() ):
        if time >= 1800:
            return False
        curr_state = q.pop()
        if curr_state.is_solution == True:
            return curr_state
        curr_depth = curr_state.get_depth()
        if curr_depth != n:
```

```
            for i from 0 to n:
                new_state = State(curr_state, curr_depth)
                new_state.move(curr_depth, i)
                q.push(new_state)
is_solution():
    for i from 0 to n - 1:
        for j from i +1 to n :
            dist = j –i
            if queens[i] == queens[j] | | queens[i] == queens[j] – dist | |
            queens[i] == queens[j] + dist:
                return False
    return True

F1():
    num_of _pairs = 0
    for i from 0 to n - 1:
        is_visible_row = True
        is_visible_d1 = True
        is_visible_d2 = True
        for j from i +1 to n :
            dist = j –i
            if queens[i] == queens[j] && is_visible_row == True:
                num_of_pairs += 1
                is_visible_row = False
            if queens[i] == queens[j] – dist && is_visible_d1== True:
                num_of_pairs += 1
                is_visible_d1 = False
            if queens[i] == queens[j] + dist && is_visible_d2 == True:
                num_of_pairs += 1
                is_visible_d2 = False
```

return num_of_pairs

priority():

return F1() + depth

3.2 Програмна реалізація

3.2.1 Вихідний код

main.py

```
import sys
from ChessBoard import ChessBoard

queens_input = 0
algo = 0
size = 0

while size > 20 or size < 4:
    try:
        size = int(input('Enter board size (min: 4, max: 20): '))
    except ValueError:
        sys.exit('Type number!')

while queens_input != '1' and queens_input != '2':
    queens_input = input('Do you want to input or generate queens:\n1 - input\n2
- generate\n')

while algo != '1' and algo != '2':
    algo = input('Choose the algorithm:\n1 - BFS\n2 - A star\n')

# create board
my_board = ChessBoard(size)

# create initial state
if queens_input == '1':
    init_state = my_board.input_queens()
else:
    init_state = my_board.generate_queens()

# print initial state
print('\nInitial state:')
my_board.print_state(init_state)

# solution
if algo == '1':
    solution = my_board.BFS_solution(init_state)
    print("\nBFS solution:")
else:
    solution = my_board.A_star_solution(init_state)
    print("\nA star solution:")

# print solution
if solution:
    my_board.print_state(solution[0])
    print(f'iterations: {solution[1]}\n'
          f'total states: {solution[2]}\n'
```

```
          f'states in memory: {solution[3]}\n'
          f'time: {solution[4]}')
```

state.py

```
from copy import copy


class State:
    def __init__(self, queens, depth=0):
        self._queens = copy(queens)
        self._depth = copy(depth)

    def is_solution(self):
        for i in range(len(self._queens) - 1):
            for j in range(i + 1, len(self._queens)):
                dist = j - i  # distance between columns
                # same diagonal
                if self._queens[i] == self._queens[j] or self._queens[i] - dist
== self._queens[j] or \
                        self._queens[i] + dist == self._queens[j]:
                    return False
        return True

    def move(self, row, col):
        self._queens[row] = col

    def get_depth(self):
        return self._depth

    def set_depth(self, depth):
        self._depth = depth

    def get_queens(self):
        return self._queens

    def __f1_heuristic(self):
        num_of_pairs = 0

        for i in range(len(self._queens) - 1):
            is_visible_row = True   # visibility in row
            is_visible_d1 = True    # visibility in left diagonal
            is_visible_d2 = True    # visibility in left diagonal

            for j in range(i + 1, len(self._queens)):
                dist = j - i
                if self._queens[i] == self._queens[j] and is_visible_row:
                    num_of_pairs += 1
                    is_visible_row = False
                if self._queens[i] == self._queens[j] - dist and is_visible_d1:
                    num_of_pairs += 1
                    is_visible_d1 = False
                if self._queens[i] == self._queens[j] + dist and is_visible_d2:
                    num_of_pairs += 1
                    is_visible_d2 = False
        return num_of_pairs

    def __priority(self):
        return self.__f1_heuristic() + self._depth

    def __gt__(self, other):
        return self.__priority() > other.__priority()
```

ChessBoard.py

```
import random
from queue import Queue, PriorityQueue
from termcolor import colored
from time import time
from state import State


class ChessBoard:
    def __init__(self, n):
        self.__n = n

    def input_queens(self):
        try:
            while True:
                s = input(f"Enter {self.__n} queens in format '1 1 1 1 1 1 1 1':
")
                condition = True
                queens = list(map(int, s.split()))
                for num in queens:
                    if int(num) >= self.__n:
                        condition = False

                if len(s.replace(' ', '')) == self.__n and condition:
                    break
            queens = list(map(int, s.split()))
            state = State(queens)
            return state

        except ValueError:
            print('Wrong input')
            exit()

    def generate_queens(self):
        queens = [random.randint(0, self.__n - 1) for x in range(self.__n)]
        state = State(queens, 0)
        return state

    def print_state(self, state: State):
        nums = ''.join(str(x) + ' ' * 5 if x < 9 else str(x) + ' ' * 4 for x in
range(self.__n))
        print(colored('\n        ' + nums, 'red'), end='')
        print(colored('\n    -' + '------' * self.__n, 'blue'))
        queens = state.get_queens()
        for i in range(self.__n):
            text = f'{i}   ' if i < 10 else f'{i}  '
            print(colored(text, 'red'), end='')
            print(colored('|  ', 'blue'), end='')
            for j in range(self.__n):
                if queens[j] == i:
                    print(colored('Q', 'green'), end='')
                    print(colored('  |  ', 'blue'), end='')
                else:
                    print(colored('   |  ', 'blue'), end='')
            print(colored('\n    -' + '------' * self.__n, 'blue'))

        print(f"Queens: {queens}")

    def BFS_solution(self, initial_state):
        start_time = time()
        q = Queue()
        q.put(initial_state)
```

```
        iterations = 0
        total_states = 1
        states_in_mem = 1

        while not q.empty():

            if time() - start_time >= 1800:
                print('Time limit!')
                return False

            iterations += 1
            curr_state = q.get()
            states_in_mem -= 1

            if curr_state.is_solution():
                return curr_state, iterations, total_states, states_in_mem,
time() - start_time

            curr_depth = curr_state.get_depth()
            if curr_depth != self.__n:
                for i in range(self.__n):
                    new_state = State(curr_state.get_queens(), curr_depth + 1)
                    new_state.move(curr_depth, i)  # queens[row] = col
                    q.put(new_state)
                    total_states += 1
                    states_in_mem += 1

    def A_star_solution(self, initial_state):
        start_time = time()
        q = PriorityQueue()
        q.put(initial_state)

        iterations = 0
        total_states = 1
        states_in_mem = 1

        while not q.empty():

            if time() - start_time >= 1800:
                return False

            iterations += 1
            curr_state = q.get()
            states_in_mem -= 1

            if curr_state.is_solution():
                return curr_state, iterations, total_states, states_in_mem,
time() - start_time

            curr_depth = curr_state.get_depth()
            if curr_depth != self.__n:
                for i in range(self.__n):
                    new_state = State(curr_state.get_queens(), curr_depth + 1)
                    new_state.move(curr_depth, i)  # queens[row] = col
                    q.put(new_state)
                    total_states += 1
                    states_in_mem += 1
```

3.2.2 Приклади роботи

На рисунках 3.1 - 3.4 показані приклади роботи програми для різних алгоритмів пошуку.

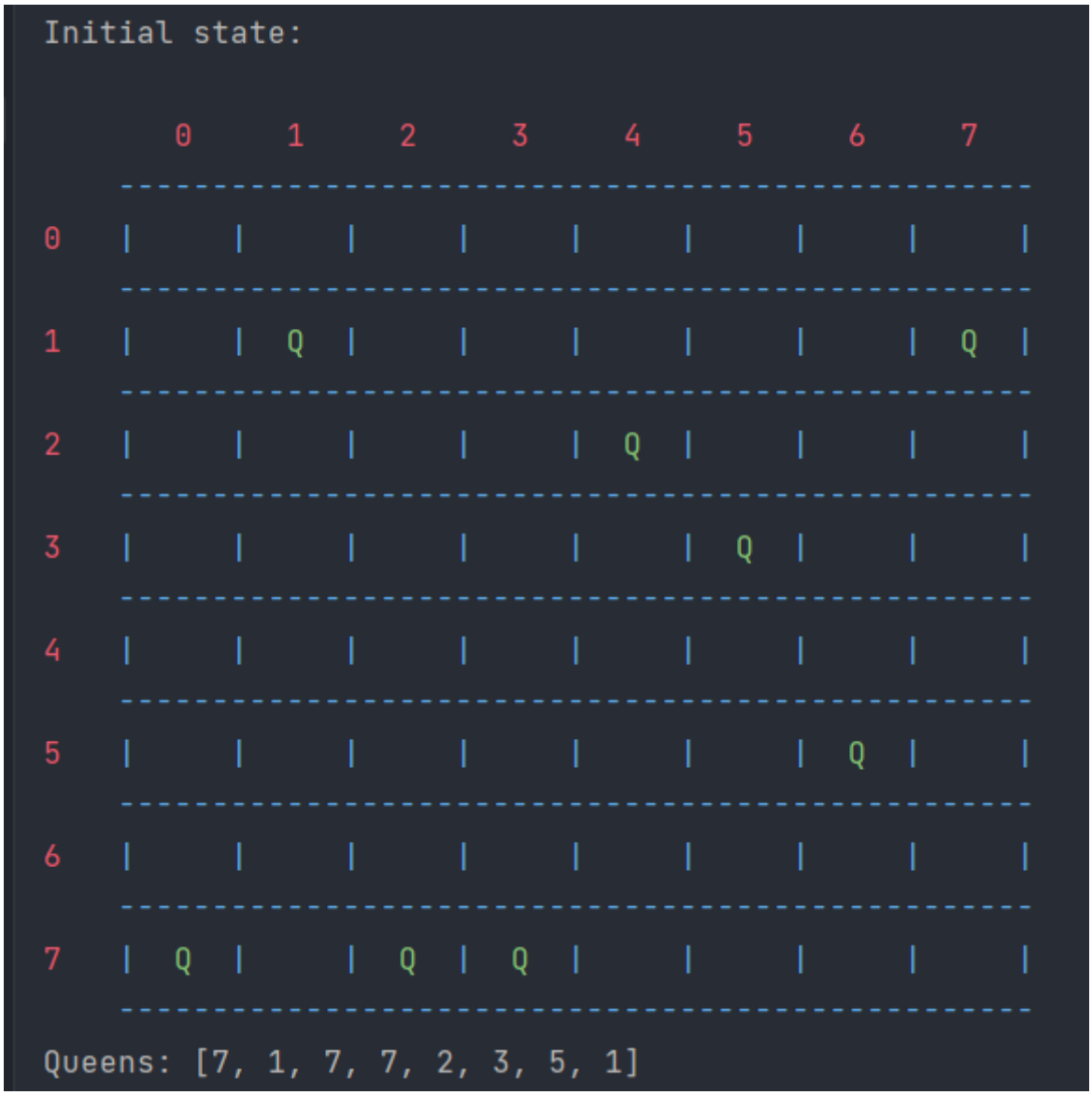

Рисунок 3.1 – Алгоритм BFS

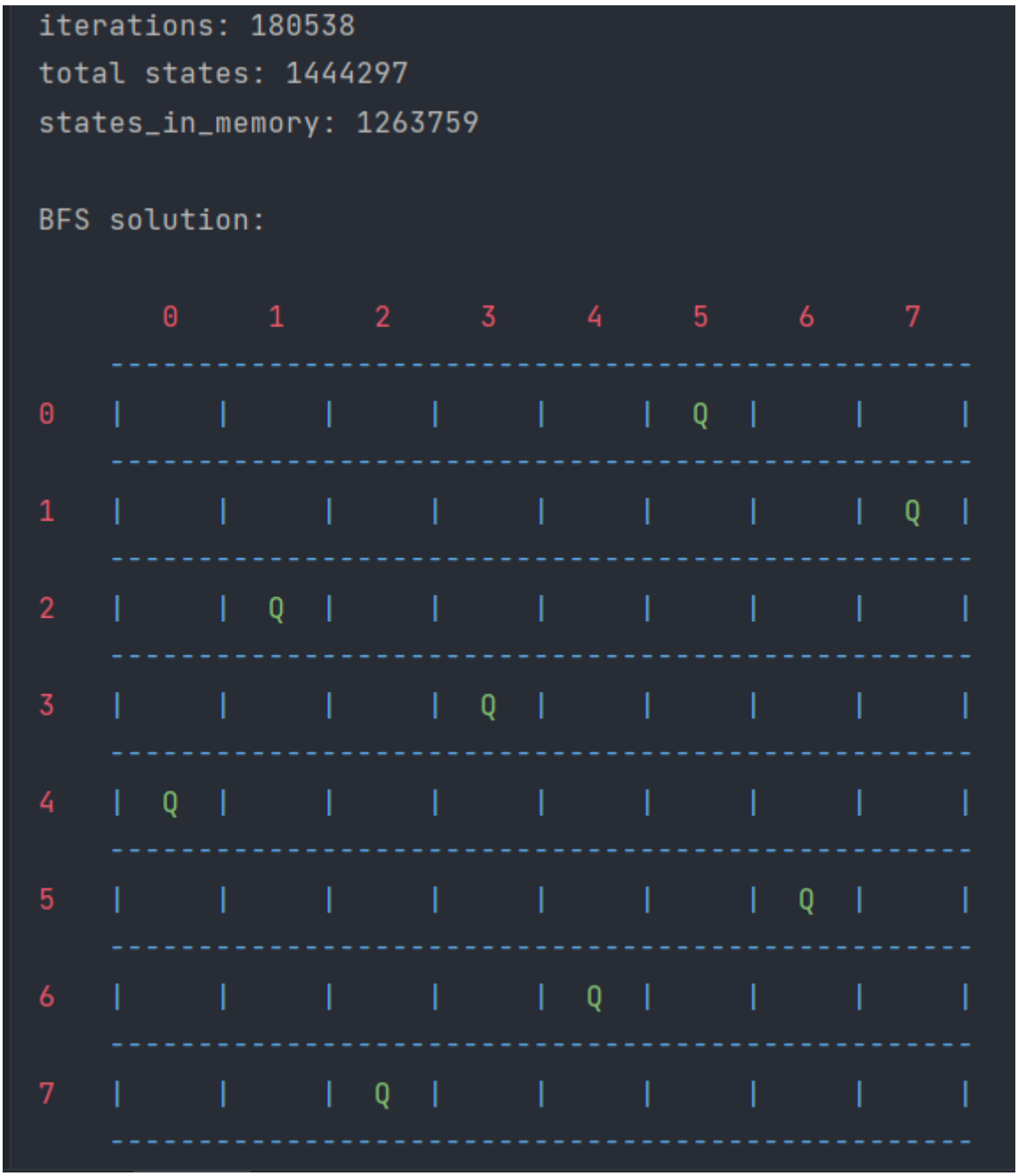

Рисунок 3.2 – Алгоритм BFS

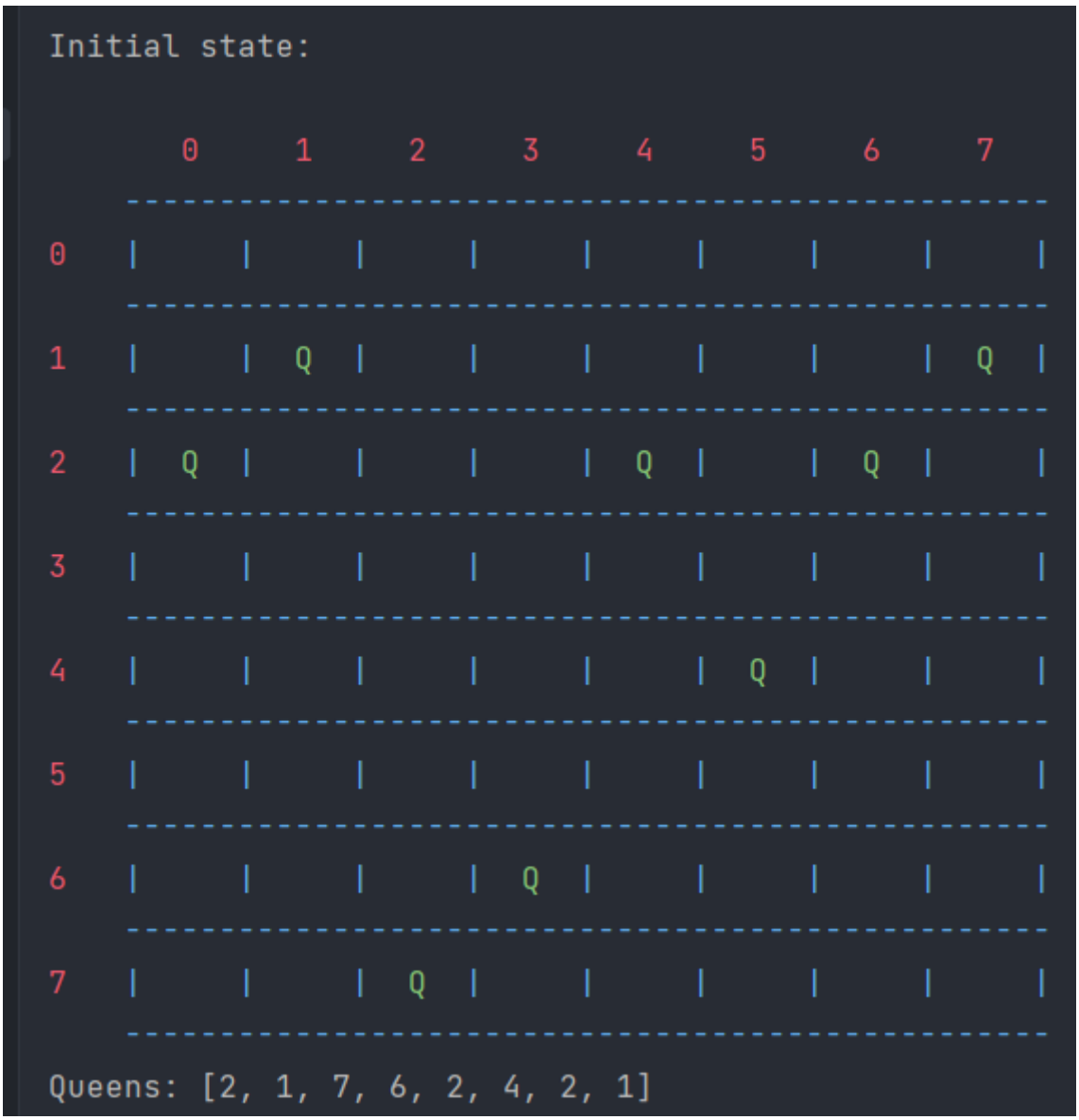

Рисунок 3.3 – Алгоритм A*

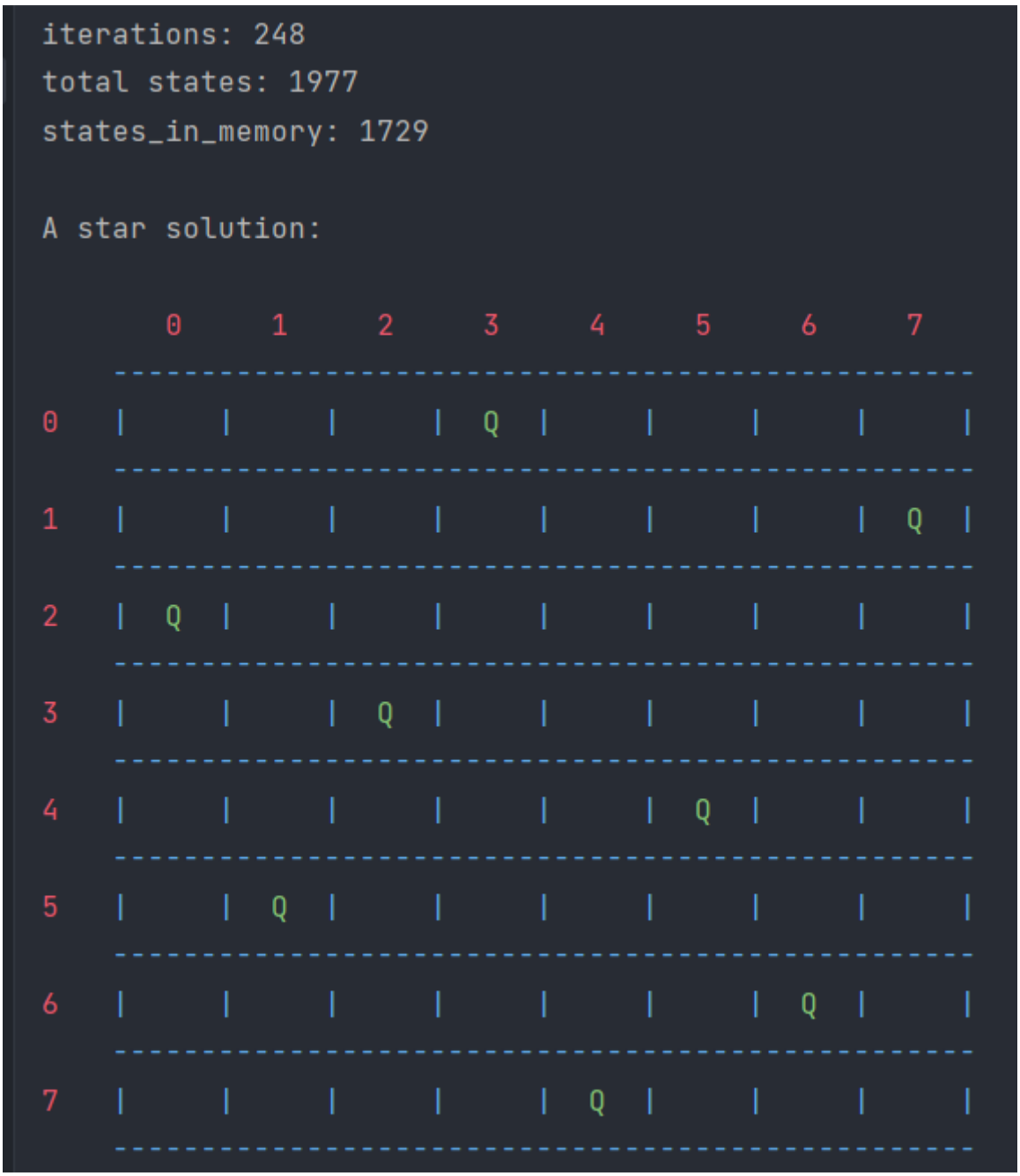

Рисунок 3.4 – Алгоритм A*

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS задачі 8-ферзів для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

Початкові стани	Ітерації	Всього станів	Всього станів у пам’яті
Стан 1: [5, 6, 2, 0, 6, 2, 6, 4]	85530	684233	598703
Стан 2: [3, 0, 7, 6, 1, 7, 6, 1]	125062	1000489	875427
Стан 3: [0, 3, 1, 5, 3, 1, 3, 4]	493539	3948305	3454766
Стан 4: [2, 7, 1, 2, 0, 4, 4, 2]	63939	511505	447566
Стан 5: [5, 0, 1, 6, 2, 7, 6, 1]	125062	1000489	875427
Стан 6: [5, 4, 5, 2, 5, 2, 1, 3]	57760	462073	404313
Стан 7: [1, 1, 3, 7, 7, 0, 4, 3]	462075	3696593	3234518
Стан 8: [4, 1, 3, 6, 2, 7, 5, 0]	2728	21817	19089
Стан 9: [5, 3, 2, 1, 4, 2, 1, 7]	211917	1695329	1483412
Стан 10: [2, 4, 5, 7, 6, 0, 7, 6]	17970	143753	125783
Стан 11: [3, 4, 4, 5, 0, 4, 4, 7]	1299852	10398809	9098957
Стан 12: [1, 2, 1, 5, 6, 0, 6, 3]	15537	124289	108752
Стан 13: [3, 2, 6, 6, 0, 6, 2, 6]	142386	1139081	996695
Стан 14: [1, 1, 5, 5, 2, 6, 1, 2]	511510	4092073	3580563
Стан 15: [3, 0, 6, 3, 7, 5, 3, 3]	462075	3696593	3234518
Стан 16: [7, 0, 7, 2, 4, 4, 5, 6]	1139084	9112665	7973581
Стан 17: [2, 6, 4, 5, 7, 7, 3, 0]	962984	7703865	6740881
Стан 18: [6, 1, 5, 4, 4, 5, 4, 3]	462075	3696593	3234518
Стан 19: [5, 4, 6, 3, 7, 2, 6, 5]	850965	6807713	5956748
Стан 20: [4, 5, 1, 6, 5, 7, 2, 5]	138264	1106105	967841
Середнє значення:	**381516**	**3052118**	**2670603**

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

Таблиця 3.3 – Характеристики оцінювання A*

Початкові стани	Ітерації	Всього станів	Всього станів у пам’яті
Стан 1: [5, 6, 2, 0, 6, 2, 6, 4]	18	137	119
Стан 2: [3, 0, 7, 6, 1, 7, 6, 1]	119	945	826
Стан 3: [0, 3, 1, 5, 3, 1, 3, 4]	326	2601	2275
Стан 4: [2, 7, 1, 2, 0, 4, 4, 2]	955	7449	6494
Стан 5: [5, 0, 1, 6, 2, 7, 6, 1]	368	2937	2569
Стан 6: [5, 4, 5, 2, 5, 2, 1, 3]	151	1201	1050
Стан 7: [1, 1, 3, 7, 7, 0, 4, 3]	289	2305	2016
Стан 8: [4, 1, 3, 6, 2, 7, 5, 0]	52	409	357
Стан 9: [5, 3, 2, 1, 4, 2, 1, 7]	430	3385	2955
Стан 10: [2, 4, 5, 7, 6, 0, 7, 6]	47	369	322
Стан 11: [3, 4, 4, 5, 0, 4, 4, 7]	544	4345	3801
Стан 12: [1, 2, 1, 5, 6, 0, 6, 3]	354	2825	2471
Стан 13: [3, 2, 6, 6, 0, 6, 2, 6]	676	5345	4669
Стан 14: [1, 1, 5, 5, 2, 6, 1, 2]	139	1105	966
Стан 15: [3, 0, 6, 3, 7, 5, 3, 3]	214	1705	1491
Стан 16: [7, 0, 7, 2, 4, 4, 5, 6]	56	441	385
Стан 17: [2, 6, 4, 5, 7, 7, 3, 0]	519	4145	3626
Стан 18: [6, 1, 5, 4, 4, 5, 4, 3]	234	1529	1295
Стан 19: [5, 4, 6, 3, 7, 2, 6, 5]	249	1985	1736
Стан 20: [4, 5, 1, 6, 5, 7, 2, 5]	22	169	147
Середнє значення	**288**	**2266**	**1979**

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто один алгоритм неінформативного пошуку (BFS) та один алгоритм інформативного пошуку (A*). Проведено порівняльний аналіз ефективності роботи цих алгоритмів. Отже, неіформативний виявився в декілька разів швидшим при будя-яких вхідних даних.

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%.