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 Кисельов Микита* _____
(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.
- **BFS** – Пошук вшир.
- **A*** – Пошук A*.
- **F2** – кількість пар ферзів, які б’ють один одного без урахування видимості.

Таблиця 2.1 – Варіанти алгоритмів

№	Задача	АНП	АІП	АЛП	Func
12	8-ферзів	BFS	A*	-	F2

Далі усі результати та розрахунки показані для розміру оперативної пам’яті половина гігабайту.

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

3.1.1 НЕІНФОРМАТИВНИЙ ПОШУК ВШИР (BFS)

1. ПОЧАТОК
2. СТВОРИТИ СПИСОК РОЗВ’ЯЗКІВ
3. ПОСТАВИТИ НА ДОШКУ У 0 РЯДОК КОРОЛЕВУ НА ВИПАДКОВИЙ СТОВПЧИК ВІД 0 ДО РОЗМІР ДОШКИ – 1
4. ЯКЩО РОЗМІР ДОШКИ МЕНШЕ 4
   4.1. ДОДАТИ У СПИСОК РОЗВЯЗКІВ ДОШКУ
5. ІНАКШЕ
   5.1. ПОВТОРИТИ ДЛЯ row ВІД 0 до -1
      5.1.1. ПОВТОРИТИ ПОКИ НЕ ДОСЯГНЕНО КІНЦЯ ДОШКИ АБО КОРОЛЕВА НА РЯДУ row ПІД АТАКОЮ
         5.1.1.1. ЗДВИНУТИ УПЕРЕД КОРОЛЕВУ НА РЯДКУ row
      5.1.2. ВСЕ ПОВТОРИТИ
      5.1.3. ЯКЩО МИ НЕ ДОСЯГЛИ КІНЦЯ ДОШКИ
      5.1.3.1. ЯКЩО РЯД НЕ ОСТАННІЙ
         5.1.3.1.1. ПОСТАВИТИ НА ДОШКУ КОРОЛЕВУ У row = row + 1 РЯДОК ТА -1 СТОВПЧИК
      5.1.3.2. ІНАКШЕ
         5.1.3.2.1. ДОДАТИ У СПИСОК РОЗВ’ЯЗКІВ ЗНАЙДЕНУ ДОШКУ
         5.1.3.2.2. ВИЙТИ З ЦИКЛУ
      5.1.4. ІНАКШЕ
      5.1.4.1. ПЕРЕЙТИ НА ПОПЕРЕДНІЙ РЯД (row = row - 1)
   5.2. ПОВЕРНУТИ СПИСОК РОЗВ'ЯЗКІВ
6. КІНЕЦЬ

3.1.2 ІНФОРМАТИВНИЙ ПОШУК A*

1. ПОЧАТОК
2. СТВОРИТИ СПИСОК РОЗВ’ЯЗКІВ
3. ЯКЩО РОЗМІР ДОШКИ МЕНШЕ 4
    3.1. ДОДАТИ У СПИСОК РОЗВЯЗКІВ ДОШКУ
4. ІНАКШЕ
    4.1. СТВОРИТИ ЧЕРГУ ДОШОК З ПРІОРИТЕТОМ (КІЛЬКІСТЬ ПАР БЕЗ УРАХУВАННЯ ВИДИМОСТІ ЯКІ БЬЮТЬ ОДИН ОДНОГО)
    4.2. ДОДАТИ У ЧЕРГУ ПОЧАТКОВУ ДОШКУ ЗГЕНЕРОВАНУ ВИПАДКОВИМ ЧИНОМ
    4.3. ПОКИ ЧЕРГА НЕ ПУСТА ПОВТОРИТИ
    4.4. ВЗЯТИ З ЧЕРГИ ПОТОЧНИЙ СТАН (ДОШКУ)
    4.5. ЯКЩО ПОТОЧНА ДОШКА Є РОЗВ'ЯЗКОМ
        4.5.1. ДОДАТИ ПОТОЧНУ ДОШКУ У СПИСОК РОЗ'ВЯЗКІВ
        4.5.2. ВИЙТИ З ЦИКЛУ
    4.6. ПРИСВОЇТИ currentDepth ГЛИБИНУ ПОТОЧНОГО СТАНУ
    4.7. ЯКЩО currentDepth НЕ ДОРІВНЮЄ РОЗМІРУ ДОШКИ
        4.7.1. ПОВТОРИТИ ДЛЯ row ВІД 0 ДО РОЗМІРУ ДОШКИ
        4.7.2. СТВОРИТИ ГЛИБИННУ КОПІЮ ДАНОГО СТАНУ (ДОШКИ)
        4.7.3. ПОСТАВИТИ КОРОЛЕВУ НА КОПІЮ У row РЯДОК ТА У КОЛОНКУ currentDepth
        4.7.4. ПРИСВОЇТИ ГЛУБИНІ ЦЬОГО СТАНУ (ГЛИБИННОЇ КОПІЇ) ЗНАЧЕННЯ currentDepth + 1
        4.7.5. ЯКЩО ЕВРЕСТИЧНА ФУНКЦІЯ ВІД ЦЬОГО СТАНУ (ГЛИБИННОЇ КОПІЇ) МЕНША НІЖ ЕВРЕСТИЧНА ФУНКЦІЯ ВІД ПОТОЧНОГО СТАНУ

4.7.5.1. ДОДАТИ У ПРИОРИТЕТНУ ЧЕРГУ ЦЕЙ СТАН (ГЛИБИННА КОПІЯ ІЗ ДОДАНОЮ КОРОЛЕВОЮ)

4.7.6. ВСЕ ПОВТОРИТИ

4.8. ВСЕ ПОВТОРИТИ

4.9. ПОВЕРНУТИ СПИСОК РОЗВ'ЯЗКІВ

5. КІНЕЦЬ

3.2 Програмна реалізація

3.2.1 Вихідний код

```
ArrayList<State> AStar() {
    ArrayList<State> solutions = new
ArrayList<>();
    long start = System.currentTimeMillis();
    if (size < MIN_SOLUTION_SIZE)
solutions.add(board);
    else {
        PriorityQueue<State> states = new
PriorityQueue<>();
        states.add(board);
        while (!states.isEmpty()) {
            State current = states.poll();
            if (current.done())
{solutions.add(current); break;}
            int currentDepth =
current.getDepth();
            if (currentDepth != size) {
                int currentGrade =
current.heuristic();
                for (int row = INIT_POS; row <
size; ++row) {
                    State node = new
State(current);
                    node.put(row,
currentDepth);
                    node.setDepth(currentDepth
+ 1);
                    if (node.heuristic() <=
currentGrade) states.add(node);
                }
            }
        }
    }
    System.out.println("A* on "  + size + "x"
+ size + " board has found solution in "
            + (System.currentTimeMillis()-
start) + "ms:");
    return solutions;
}
ArrayList<State> BFS() {
    ArrayList<State> solutions = new
ArrayList<>();
    State solution = new State(board);
    solution.put(0, new Random().nextInt(size
- 1));
    long start = System.currentTimeMillis();
    if (size < MIN_SOLUTION_SIZE)
solutions.add(solution);
    else {
        for (int row = INIT_POS; row >=
INIT_POS;) {
            do solution.forward(row);
            while (solution.at(row) < size &&
solution.attacked(row));
            if (solution.at(row) < size) {
                if (row < size -
1)solution.put(++row, INIT_POS - 1);
                else {
                    solutions.add(new
State(solution));
                    break;
                }
            } else --row;
        }
    }
    System.out.println("BFS on "  + size + "x"
+ size + " board has found solution in "
            + (System.currentTimeMillis()-
start) + "ms:");
    return solutions;
}
public class State implements
Comparable<State> {
    private final int[] board;
    private final int size;
    private int depth;
    public State(int size, int depth) {
        this.board = new int[this.size =
size];
        this.depth = depth;
        ArrayList<Integer> pos = new
ArrayList<>(size);
        for (int i = 0; i < size; ++i)
pos.add(i);
        Random random = new Random();
        for (int i = 0; i < size; ++i) put(i,
pos.remove(random.nextInt(pos.size())));
    }

    public State(State state) {
        this.board = state.board.clone();
        this.depth = state.depth;
        this.size = state.size;
    }
    void put(int row, int col) {
        board[row] = col;
    }
    boolean attacked(int row) {
        for (int i = 1; i <= row; ++i)
            if (board[row-i] == board[row] ||
board[row-i] == board[row] - i || board[row-i]
== board[row] + i)
                return true;
        return false;
    }
    boolean done() {
        for (int queen = 0; queen <
board.length; ++queen)
```

```
            for (int enemy = 0; enemy <
board.length && board[enemy] >= 0; ++enemy)
                if (enemy != queen
                        && (board[queen] ==
board[enemy]
                        || queen +
board[queen] == enemy + board[enemy]
                        || queen -
board[queen] == enemy - board[enemy]))
                    return false;
        return true;
    }
    int heuristic() {
        int conflicts = 0;
        for (int queen = 0; queen <
board.length; ++queen)
            for (int enemy = 0; enemy <
board.length && board[enemy] >= 0; ++enemy)
                if (enemy != queen
                        && (board[queen] ==
board[enemy]
                        || queen +
board[queen] == enemy + board[enemy]
                        || queen -
board[queen] == enemy - board[enemy]))
                    ++conflicts;
        return conflicts >> 1;
    }
    public int at(int row) {
        return board[row];
    }
    public void forward(int row) {
        ++board[row];
    }
    public void setDepth(int depth) {
        this.depth = depth;
    }
    public int getDepth() {
        return depth;
    }
    @Override public int compareTo(State o) {
        return
Integer.compare(this.heuristic(),
o.heuristic());
    }
    @Override public String toString() {
        return  depth + "d" +
Arrays.toString(board) + "b";
    }
}
```

3.2.2 Приклади роботи

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

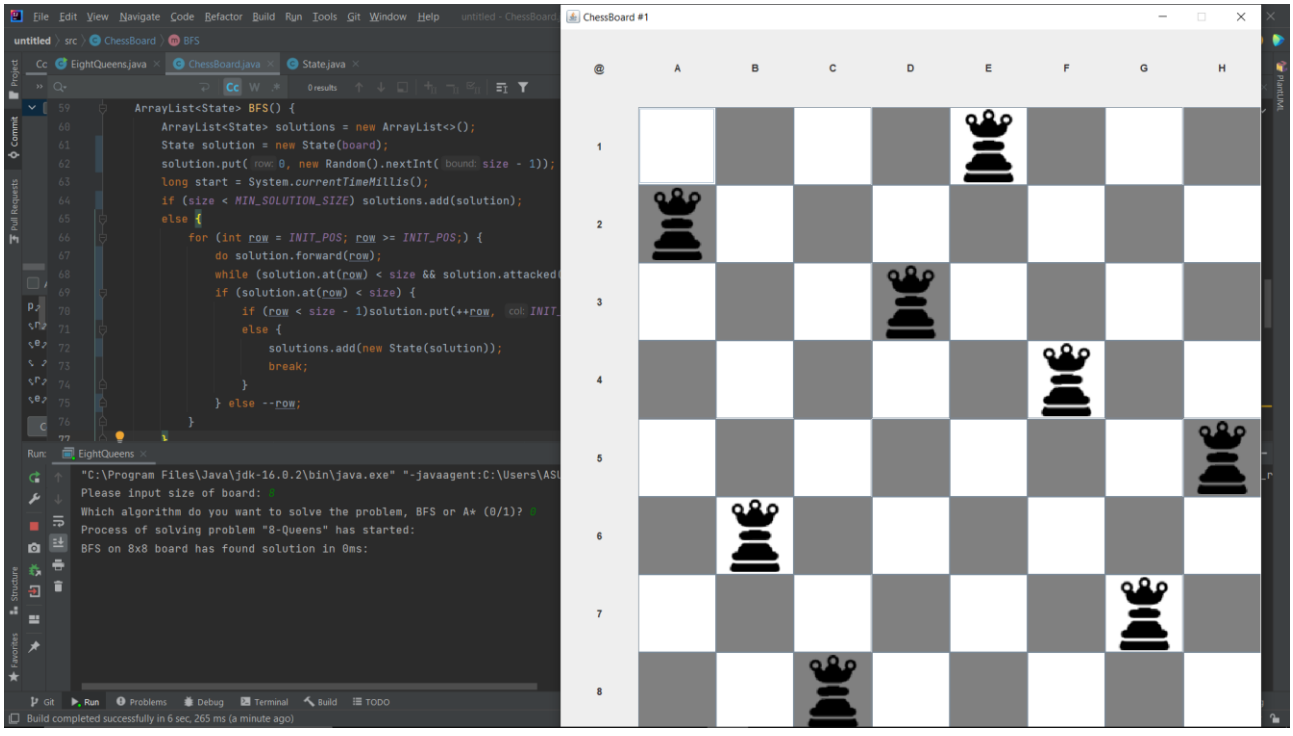

Рисунок 3.1 – Алгоритм пошуку вшир (BFS)

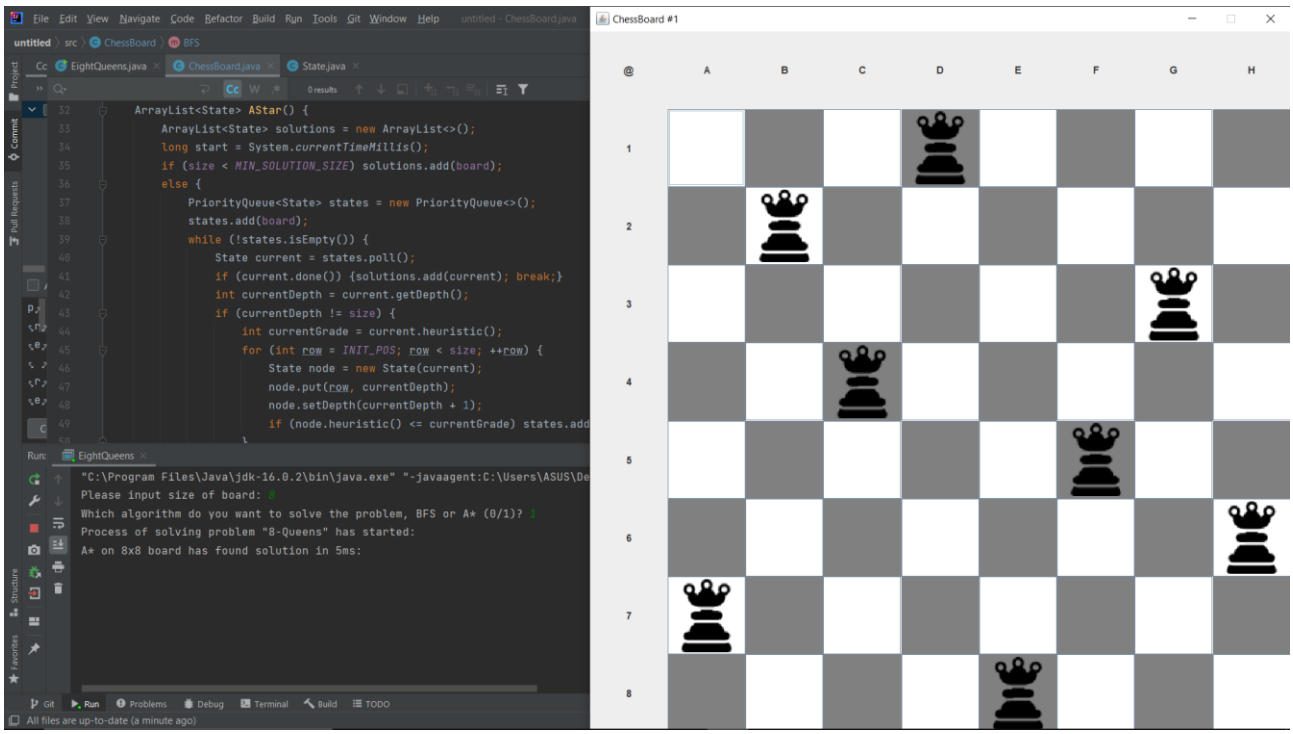

Рисунок 3.2 – Алгоритм пошуку A*

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму пошуку вшир (BFS), задачі 8-ферзів для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму пошуку вшир (BFS)

Початкові стани	**Ітерації**	**К-сть гл. кутів**	**Всього станів**	**Всього станів у пам'яті**
Стан 1 0d[5, 4, 1, 6, 2, 3, 7, 0]b	89	0	1	1
Стан 2 0d[4, 3, 5, 7, 1, 6, 2, 0]b	9	0	1	1
Стан 3 0d[4, 5, 0, 1, 2, 3, 6, 7]b	31	0	1	1
Стан 4 0d[5, 2, 3, 4, 6, 0, 7, 1]b	47	0	1	1

Початкові стани	**Ітерації**	**К-сть гл. кутів**	**Всього станів**	**Всього станів у пам’яті**
Стан 5 0d[1, 0, 7, 6, 5, 4, 3, 2]b	89	0	1	1
Стан 6 0d[1, 3, 2, 5, 7, 4, 0, 6]b	29	0	1	1
Стан 7 0d[3, 6, 7, 5, 4, 1, 0, 2]b	45	0	1	1
Стан 8 0d[0, 2, 5, 7, 3, 4, 6, 1]b	31	0	1	1
Стан 9 0d[6, 1, 4, 5, 3, 2, 7, 0]b	9	0	1	1
Стан 10 0d[6, 3, 0, 5, 7, 4, 1, 2]b	29	0	1	1
Стан 11 0d[0, 6, 3, 2, 5, 7, 1, 4]b	31	0	1	1
Стан 12 0d[3, 7, 2, 4, 1, 5, 6, 0]b	9	0	1	1
Стан 13 0d[2, 1, 7, 5, 4, 6, 0, 3]b	89	0	1	1
Стан 14 0d[4, 7, 3, 1, 0, 5, 6, 2]b	9	0	1	1
Стан 15 0d[6, 2, 4, 1, 7, 5, 3, 0]b	45	0	1	1
Стан 16 0d[1, 6, 3, 4, 0, 5, 7, 2]b	9	0	1	1
Стан 17 0d[5, 2, 4, 7, 0, 1, 3, 6]b	9	0	1	1

Початкові стани	**Ітерації**	**К-сть гл. кутів**	**Всього станів**	**Всього станів у пам'яті**
Стан 18 0d[1, 6, 4, 5, 2, 7, 0, 3]b	9	0	1	1
Стан 19 0d[3, 2, 0, 4, 6, 7, 5, 1]b	89	0	1	1
Стан 20 0d[3, 5, 1, 2, 6, 0, 7, 4]b	79	0	1	1
Середнє значення	**39**	**0**	**1**	**1**

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

Таблиця 3.2 – Характеристики оцінювання алгоритму пошуку A*

Початкові стани	**Ітерації**	**К-сть гл. кутів**	**Всього станів**	**Всього станів у пам'яті**
Стан 1 0d[0, 6, 5, 1, 7, 3, 2, 4]b	7	0	55	7
Стан 2 0d[2, 5, 4, 0, 7, 6, 3, 1]b	1422	0	5302	1422
Стан 3 0d[5, 7, 6, 0, 1, 3, 2, 4]b	13	0	109	13
Стан 4 0d[1, 6, 4, 3, 5, 7, 2, 0]b	93	0	573	93
Стан 5 0d[4, 5, 6, 0, 2, 3, 7, 1]b	747	0	2547	747
Стан 6 0d[4, 1, 3, 7, 6, 5, 0, 2]b	11	0	91	11
Стан 7 0d[5, 3, 7, 4, 2, 0, 1, 6]b	668	0	3068	668

Початкові стани	**Ітерації**	**К-сть гл. кутів**	**Всього станів**	**Всього станів у пам'яті**
Стан 8 0d[3, 0, 7, 5, 2, 6, 1, 4]b	7	0	55	7
Стан 9 0d[0, 1, 2, 5, 7, 6, 3, 4]b	7117	0	25637	7117
Стан 10 0d[3, 6, 7, 5, 1, 2, 4, 0]b	2398	0	9030	2398
Стан 11 0d[1, 6, 0, 7, 4, 3, 2, 5]b	5	0	37	5
Стан 12 0d[4, 1, 6, 0, 3, 2, 5, 7]b	33	0	201	33
Стан 13 0d[3, 1, 7, 2, 4, 6, 0, 5]b	24	0	184	24
Стан 14 0d[0, 2, 7, 4, 1, 3, 5, 6]b	142	0	846	142
Стан 15 0d[5, 0, 4, 2, 6, 7, 3, 1]b	217	0	929	217
Стан 16 0d[4, 3, 1, 2, 6, 5, 0, 7]b	435	0	1651	435
Стан 17 0d[0, 7, 5, 4, 2, 1, 6, 3]b	22	0	158	22
Стан 18 0d[7, 2, 3, 1, 4, 5, 0, 6]b	12	0	76	12
Стан 19 0d[7, 3, 2, 1, 5, 6, 4, 0]b	1142	0	4518	1142
Стан 20 0d[4, 1, 7, 2, 6, 5, 0, 3]b	274	0	1362	274
Середнє значення	**739**	**0**	**2821**	**739**

ВИСНОВОК

При виконанні даної лабораторної роботи Я розглянув наступні алгоритми: алгоритм неінформативного пошуку – пошук вшир (BFS), а також алгоритм інформативного пошуку – пошук A*.

У ході даної роботи дослідив ці алгоритми та провів експерименти по 20 серій кожного, їх результати наведено нижче.

1. Для алгоритму BFS
    1.1. Середня кількість ітерацій: 39
    1.2. Середня кількість станів: 1
    1.3. Середня кількість станів у пам’яті: 1
    1.4. Середня кількість разів, коли алгоритм заходив у глухий кут: 0
2. Для алгоритму A*
    2.1. Середня кількість ітерацій: 739
    2.2. Середня кількість станів: 2821
    2.3. Середня кількість станів у пам’яті: 739
    2.4. Середня кількість разів, коли алгоритм заходив у глухий кут: 0

Порівнюючи алгоритми можна зробити висновок, що алгоритм A* потребує значно більше ітерацій у порівнянні з BFS, у середньому на 700 більше. Теж саме можна сказати про кількість станів, що зберігається у пам’яті. Також A* генерує набагато більше станів ніж BFS, а саме на 2820 станів. Кількість глухих кутів в обох алгоритмах дорівнює нулю, тому що у даній задачі (8-ферзів) завжди можна знайти розв’язок, навіть коли дійшли до кінця дошки, повернувшись на крок назад.

Але перелічені мінуси алгоритму неоптимізованого A* компенсуються його швидкістю на значеннях, більших за 12. Він працює значно швидше, ніж BFS. Результати залежності часу у мс від розміру дошки у клітинках обох алгоритмів наведені на рисунку 3.4, червоним показано алгоритм неінформативного пошуку BFS, та зеленим – алгоритм інформативного пошуку A*.

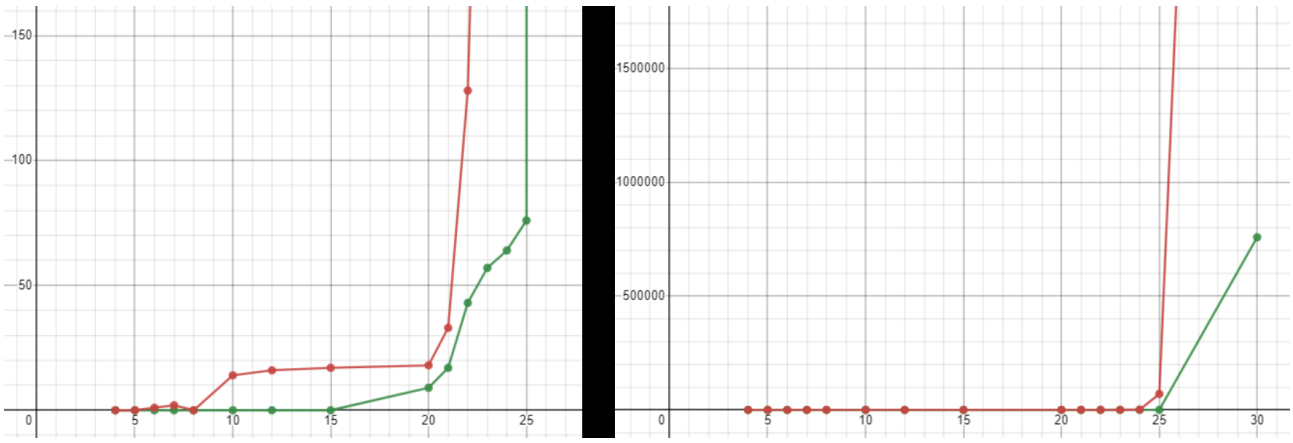

Рисунок 3.4 – графік залежності часу роботи алгоритмів BFS та A* від розміру дошки.

Таким чином, Я хочу зробити висновок, що алгоритм BFS краще використовувати, коли розмір дошки менший за 12, бо він працює швидше ніж A* на цих розмірах та використовує менше пам’яті. Але, якщо розмір дошки більше або рівний 12 та є доступ до хоча б пів гігабайту оперативної пам’яті, то краще використовувати алгоритм інформативного пошуку A*.

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%.