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-14 Качмар А.Д.* _____
(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

Алгоритм **(LDFS)**

```
deapthLimitedSearch (limit, parentNode)
  recursiveDls(parentNode, limit)
end


recursiveDls (limit, node)
  isCutoff = false
  if (isSolution(node))
    solution = node
    return SearchResult(node)
  if (node.depth = limit)
    isCutoff = true
    return SearchResult(cutoff)
  for i to generateSuccessor(node).length
    result =  recursiveDls (limit, successor)
    if (result.fail and result.message = cutoff)
      isCutoff = true
    if (result.success)
      return SearchResult(successor)
  end for
  if (isCutoff)
    return SearchResult(cutoff)
  else
    return SearchResult(failure)
end
```

```
generateSuccessor(node)
  checkedStates.add(node)
  successors = []
  for I to 8
    currentQueenPos = getCurrentPos(state.position,i)
    if (currentQueenPos not null)
      otherQueens = getOtherQueens(I)
      for J to 7
        positions.add(otherQueens)
        positions.add(QueenPositon(I,J))
        if (positions is not checked )
          successors.add(positions)
      end for
    end if
  end for
end
```

Алгоритм **(A*)**

```
aStarSearch(node)
  open = priorityQueue(node)
  closed = priorityQueue()
  while (open not empty)
     currentNode = open.extractMin
     if (currentNode.prositions is solution)
        return currentNode.prositions
     closed.add(currentNode)
     for successor to successors
        if (closed not contains successor)
           open.add(successor)
     end for
  end while
  return SearchResult(failure)

countFunctionCost(positions)
   cost = 0
   for pos1 to positions
      for pos2 to positions
         hasNoPosition = getPosBetween(pos1, pos2)
         if (pos1 not equal pos2 AND hasNoPosition)
            cost++
      end for
   end for
   return cost
```

3.2 Програмна реалізація

3.2.1 Вихідний код

Алгоритм **(LDFS)**

```
public SearchResult depthLimitedSearch() {
    time = System.currentTimeMillis();
    SearchResult result = recursiveDls(parentNode);
    time = System.currentTimeMillis() - time;
    return result;
}

private SearchResult recursiveDls(GameNode currentNode) {
    iterations++;
    boolean isCutoff = false;
    if (validatePositions(currentNode.getPositions())) {
        solution = currentNode;
        return new SearchResult(currentNode, isSuccess: true);
    }
    if (Objects.equals(currentNode.getDepth(), MAX_DEPTH)) {
        return new SearchResult(CUTOFF_MESSAGE, isSuccess: false);
    }
    List<GameNode> successors = generateSuccessors(currentNode);
    states += successors.size();
    for (GameNode successor : successors) {
        SearchResult result = recursiveDls(successor);
        if (!result.isSuccess() && result.getMessage().equals(CUTOFF_MESSAGE)) {
            isCutoff = true;
        }
        if (result.isSuccess()) {
            result.getSolutionPath().add(successor);
            result.setSolution(solution);
            return result;
        }
    }
    return new SearchResult(isCutoff ? CUTOFF_MESSAGE : "failure", isSuccess: false);
}
```

```java
private List<GameNode> generateSuccessors(GameNode currentState) {
    List<GameNode> successors = new ArrayList<>();
    checkedStates.add(currentState.getPositions());

    for (int currentCol = 0; currentCol < QUEENS; currentCol++) {
        QueenPosition takenPositions = getCurrentPosition(currentState.getPositions(), currentCol);
        if (takenPositions != null) {
            List<QueenPosition> otherQueens = retainAllNotEmptyPositions(currentState.getPositions(), currentCol);
            List<Integer> rows = IntStream.rangeClosed(0, 7).boxed().collect(Collectors.toList());
            rows.remove(takenPositions.getyPos());

            for (Integer row : rows) {
                List<QueenPosition> positions = new ArrayList<>(otherQueens);
                positions.add(new QueenPosition(currentCol, row));
                if (!isStateChecked(positions)) {
                    successors.add(new GameNode(positions, depth: currentState.getDepth() + 1));
                }
            }
        }
    }
    if (successors.isEmpty()) {
        fails++;
    }
    return successors;
}
```

```java
private boolean isStateChecked(List<QueenPosition> currentState) {
    return checkedStates.stream().anyMatch(positions -> positions.equals(currentState));
}
```

Алгоритм **(A*)**

```
public SearchResult aStarSearch() {
    long start = System.currentTimeMillis();
    PriorityQueue<GameNode> open = new PriorityQueue<>(new GameNodeComparator());
    PriorityQueue<GameNode> closed = new PriorityQueue<>(new GameNodeComparator());
    open.add(parentNode);
    while (!open.isEmpty()) {
        iterations++;
        GameNode currentNode = open.poll();
        if (validatePositions(currentNode.getPositions())) {
            time = System.currentTimeMillis() - start;
            return new SearchResult(currentNode, isSuccess: true);
        }
        closed.add(currentNode);
        memStates = closed.size();
        List<GameNode> successors = generateSuccessors(currentNode);
        states += successors.size();
        for (GameNode successor : successors) {
            if (!closed.contains(successor)) {
                open.add(successor);
            } else {
                fails++;
            }
        }
    }
    time = System.currentTimeMillis() - start;
    return new SearchResult( message: "failure", isSuccess: false);
}
```

```
public class GameNodeComparator implements Comparator<GameNode> {
    @Override
    public int compare(GameNode o1, GameNode o2) {
        return o1.getTotalCost().compareTo(o2.getTotalCost());
    }
}
```

```java
public Integer countFunctionCost() {
    Map<QueenPosition, QueenPosition> positionMap = new HashMap<>();
    this.positions.forEach(pos1 -> this.positions.forEach(pos2 -> {
        boolean isDuplicate = positionMap.containsKey(pos1) && positionMap.get(pos1).equals(pos2) ||
                positionMap.containsKey(pos2) && positionMap.get(pos2).equals(pos1);
        if (isConflict(this.positions, pos1, pos2) && !pos1.equals(pos2) && !isDuplicate) {
            positionMap.put(pos1, pos2);
        }
    }));
    return positionMap.size();
}
```

3.2.2 Приклади роботи

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

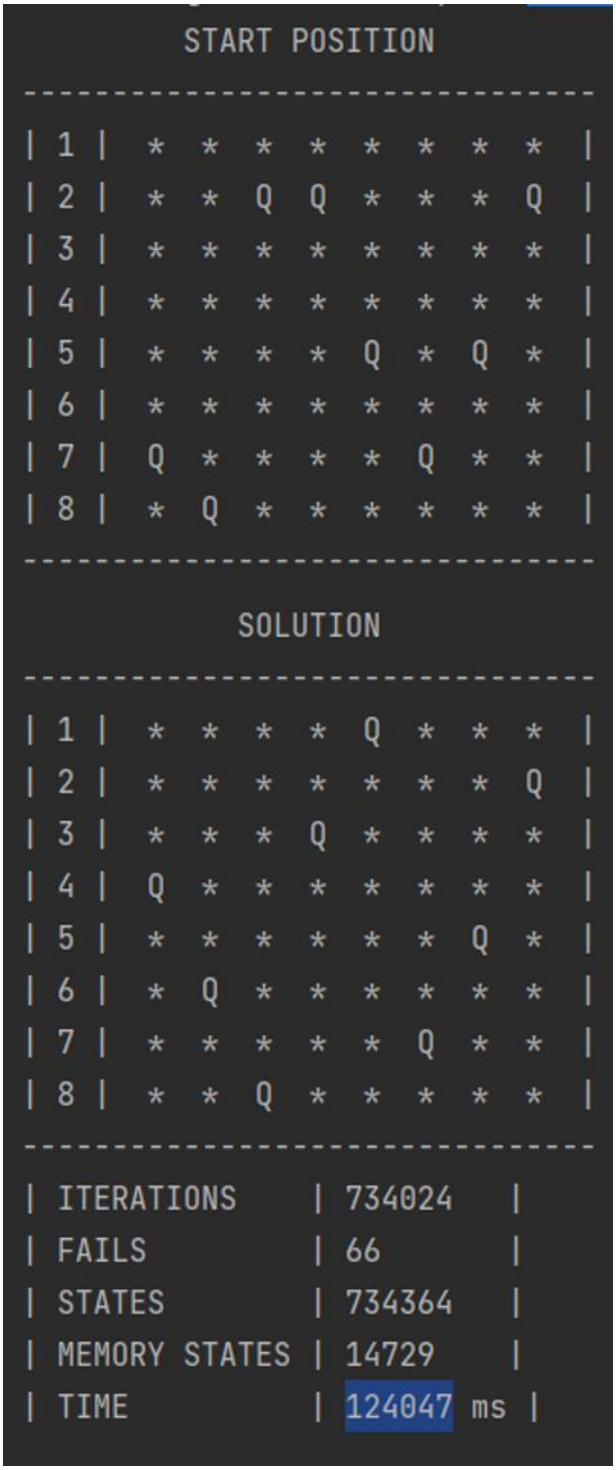

Рисунок 3.1 – Алгоритм LDFS

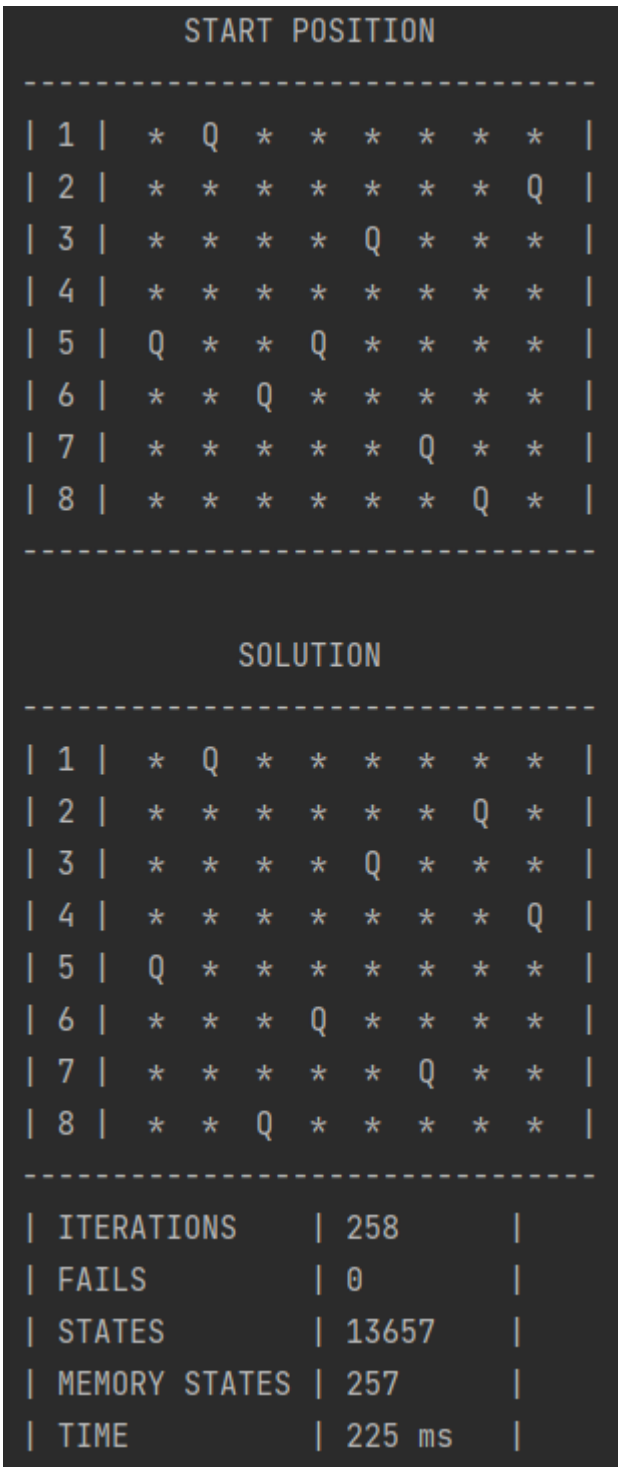

Рисунок 3.2 – Алгоритм A*

3.3 Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання алгоритму **LDFS**

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом'яті
Стан 1	647837	55	647933	13004
Стан 2	260058	13	260187	5198
Стан 3	1378361	101	1378451	28054
Стан 4	1145670	97	1145783	23221
Стан 5	2345252	208	2345251	49419
Стан 6	13649	0	13817	261
Стан 7	1199787	98	1199870	24351
Стан 8	1464574	164	1464844	29921
Стан 9	748369	66	748674	15005
Стан 10	527510	65	527826	10580
Стан 11	9129	0	9495	180
Стан 12	831517	67	831972	16677
Стан 13	158743	10	159068	3164
Стан 14	2815079	334	2815387	58077
Стан 15	1017231	103	1017510	20530
Стан 16	133692	5	134049	2675
Стан 17	1564505	171	1564797	32048
Стан 18	292334	19	292668	5845
Стан 19	266741	19	267075	5347
Стан 20	11673	0	12029	228

В таблиці 3.2 наведені характеристики оцінювання алгоритму Назва алгоритму, задачі Назва задачі для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму **A***

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом’яті
Стан 1	21	0	1108	20
Стан 2	120	0	6536	119
Стан 3	52	0	2780	51
Стан 4	105	0	5824	104
Стан 5	576	0	30459	575
Стан 6	45	0	2408	44
Стан 7	111	0	6027	110
Стан 8	98	0	5356	97
Стан 9	85	0	4651	84
Стан 10	197	0	10787	196
Стан 11	679	0	36781	678
Стан 12	317	0	17211	316
Стан 13	157	0	8574	156
Стан 14	14	0	719	13
Стан 15	27	0	1443	26
Стан 16	67	0	3640	66
Стан 17	46	0	2490	45
Стан 18	104	0	5591	103
Стан 19	14	0	725	13
Стан 20	258	0	13657	257

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто та досліджено алгоритми неінформативного, інформативного та локального пошуку. В ході виконання лабораторної роботи було вирішено задачу про вісім ферзів з використанням алгоритмів LDFS та A*. На основі отриманих результатів було проведено порівняльний аналіз ефективності використання алгоритмів. Отже було досліджено алгоритми неінформативного, інформативного та локального пошуку.

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%.