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»

„ Проектування і аналіз алгоритмів зовнішнього сортування”

Виконав студент: ІП-13 Паламарчук Олександр Олександрович

Київ 2022

ЗМІСТ

МЕТА 3
ЗАВДАННЯ 3
ВИКОНАННЯ 4
- Псевдокод алгоритмів 4

LDFS 4
RBFS 5
- Програмна реалізація 6

LDFS 7
RBFS 9
- Приклад роботи 11

LDFS 12
RBFS 15
- Дослідження алгоритмів 17

LDFS 17
RBFS 19
ВИСНОВОК 23

ВАРІАНТ 21

МЕТА

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).

ВИКОНАННЯ

- Псевдокод алгоритмів

LDFS

АЛГОРИТМ ldfs(problem, limit)

etpPoint присвоїти координати пустої клітинки

root присвоїти значення початкового вузла

ПОВЕРНУТИ recursiveLdfs(root, limit)

КІНЕЦЬ ldfs

АЛГОРИТМ recursiveLdfs(node, limit)

ПРИСВОЇТИ cutoff_occurred значення false

ЯКЩО поточний стан є рішенням проблеми, ТО повернути результат.

ІНАКШЕ ЯКЩО максимальна глибина досягнута, ТО повернути індикатор невдачі cutoff

ІНАКШЕ ЦИКЛ для кожного successor поточного вузла

ПРИСВОЇТИ result значення рекурсивного виклику методу recursiveLdfs(successor, limit)

ЯКЩО result дорівнює cutoff, ТО ПРИСВОЇТИ cutoff значення true

ІНАКШЕ ЯКЩО result НЕ дорівнює failure, ТО ПОВЕРНУТИ result

КІНЕЦЬ ЦИКЛУ

ЯКЩО cutoff_occurred?, ТО ПОВЕРНУТИ cutoff

ІНАКШЕ ПОВЕРНУТИ індикатор невдачі failure

КІНЕЦЬ recursiveLdfs

RBFS

АЛГОРИТМ rbfs(problem)

etpPoint присвоїти координати пустої клітинки

root присвоїти значення початкового вузла

ПОВЕРНУТИ recursiveRbfs(root, ∞)

КІНЕЦЬ rbfs

АЛГОРИТМ recursiveRbfs(node, f_limit)

ЯКЩО поточний стан є рішенням проблеми, ТО повернути результат

ПРИСВОЇТИ successors множину наслідників

ЯКЩО множина наслідників пуста, ТО ПОВЕРНУТИ failure, ∞

ЦИКЛ проходу по кожному successor

ПРИСВОЇТИ у поле вартість successor значення max(g(successor) + h(successor), вартість node)

КІНЕЦЬ ЦИКЛУ

ЦИКЛ

ПРИСВОЇТИ best вузол з найменшим f-значенням в множині successors

ЯКЩО вартість вуза best більше ніж f_limit, ТО повернути failure, вартість вузла best

ПРИСВОЇТИ alternative вузол другий після найменшого значення в множині successors

ПРИСВОЇТИ result і варість вузла best значення RBFS(problem, best, min(f_limit, alternative))

ЯКЩО result НЕ дорівнює failure, ТО ПОВЕРНУТИ result

КІНЕЦЬ ЦИКЛУ

КІНЕЦЬ recursiveRbfs

- Програмна реалізація

LDFS

```java
package org.example.algorithm;

import lombok.Getter;
import org.example.node.Node;
import org.example.parser.Parser;
import org.example.utils.Statistic;

import java.awt.*;
import java.util.List;

import static org.example.node.Indicator.*;
import static org.example.utils.Utils.*;

@Getter
public class LimitDepthFirstSearch {

    private final Statistic statistic;
    private static final int[][] goal;

    static {
        goal = Parser.getGoalState();
    }

    {
        statistic = new Statistic();
    }

    public static void main(String[] args) {
        int[][] problem = generateProblem();
        printExecutionTimeOf(() -> {
            var ldfs = new LimitDepthFirstSearch();
            handleResult(ldfs.search(problem, 25));
            ldfs.getStatistic().printStatistic();
        });
    }
```

```
public Result search(int[][] problem, int limit) {
    if (notSolvable(problem))
        return Result.of(NOT_SOLVABLE, null);

    Point eptTile = getEmptyTileCoordinates(problem);
    Node root = new Node(problem, eptTile.x, eptTile.y, 0, null, null);

    statistic.addUniqueState(root);
    Result result = recursiveSearch(root, limit, System.nanoTime());
    statistic.addUniqueStateInMemory(root);

    if (result.isTerminated())
        statistic.setAlgorithmTerminated(true);

    return result;
}

private Result recursiveSearch(Node node, int limit, long start) {
    statistic.incrementNumberOfIteration();
    if (timeOut(start) || memoryLimitIsReached())
        return Result.of(TERMINATED, null);

    boolean cutoffOccurred = false;
    if (node.isSolution(goal))
        return Result.of(SOLUTION, node);

    if (node.depthIsReached(limit))
        return Result.of(CUTOFF, null);

    List<Node> successors = node.getSuccessors();
    statistic.addUniqueStates(successors);

    for (Node successor : successors) {
        Result result = recursiveSearch(successor, limit, start);

        if (result.cutoff())
```

```
                cutoffOccurred = true;
            else if (result.hasSolution() || result.isTerminated()) {
                statistic.addUniqueStatesInMemory(successors);
                return result;
            }
        }

        if (cutoffOccurred)
            return Result.of(CUTOFF, null);
        else
            return Result.of(FAILURE, null);
    }
}
```

RBFS

```
package org.example.algorithm;

import lombok.Getter;
import org.example.node.Node;
import org.example.parser.Parser;
import org.example.utils.Statistic;
import org.example.utils.Utils;

import java.awt.*;
import java.util.Comparator;
import java.util.List;

import static java.lang.Math.max;
import static java.lang.Math.min;
import static org.example.node.Indicator.*;
import static org.example.utils.Utils.*;

@Getter
public class RecursiveBestFirstSearch {
    private final Statistic statistic;
    private static final int[][] goal;
```

```
static {
    goal = Parser.getGoalState();
}

{
    statistic = new Statistic();
}

public static void main(String[] args) {
    int[][] problem = Utils.generateProblem();
    printExecutionTimeOf(() -> {
        var rbfs = new RecursiveBestFirstSearch();
        handleResult(rbfs.search(problem));
        rbfs.getStatistic().printStatistic();
    });
}

public Result search(int[][] problem) {
    if (notSolvable(problem))
        return Result.of(0, NOT_SOLVABLE, null);

    Point eptTile = getEmptyTileCoordinates(problem);
    Node root = new Node(problem, eptTile.x, eptTile.y, 0, null, null);

    statistic.addUniqueState(root);
    Result result = recursiveSearch(root, Integer.MAX_VALUE, System.nanoTime());
    statistic.addUniqueStateInMemory(root);

    if (result.isTerminated())
        statistic.setAlgorithmTerminated(true);

    return result;
}

private Result recursiveSearch(Node node, int fLimit, long start) {
    statistic.incrementNumberOfIteration();
    if (timeOut(start) || memoryLimitIsReached())
```

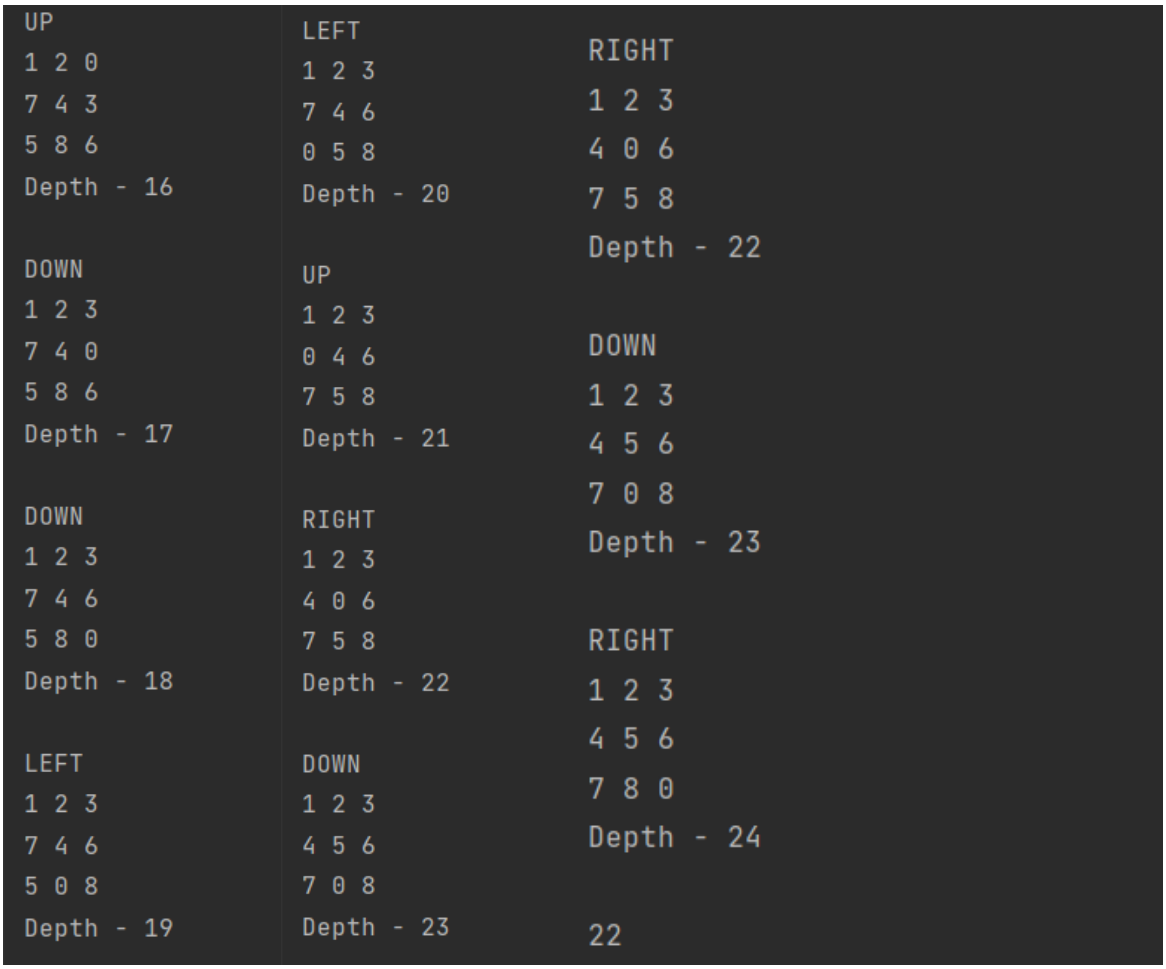
```
            return Result.of(Integer.MAX_VALUE, TERMINATED, null);

        if (node.isSolution(goal))
            return Result.of(fLimit, SOLUTION, node);

        List<Node> successors = node.getSuccessors();
        statistic.addUniqueStates(successors);

        if (successors.isEmpty())
            return Result.of(Integer.MAX_VALUE, FAILURE, null);

        for (Node successor : successors)
            successor.setF(max(successor.misplaced(goal) + successor.getDepth(),
node.getF()));

        while (true) {
            successors.sort(Comparator.comparing(Node::getF));

            Node best = successors.get(0);
            if (best.getF() > fLimit)
                return Result.of(best.getF(), FAILURE, null);

            Node alt = successors.get(1);

            Result result = recursiveSearch(best, min(alt.getF(), fLimit), start);
            best.setF(result.getFBest());

            if (result.hasSolution() || result.isTerminated()) {
                statistic.addUniqueStatesInMemory(successors);
                return result;
            }
        }
    }
}
```

- Приклад роботи

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

LDFS

```
INITIAL
0 1 2
7 4 3
5 8 6
Depth - 0

RIGHT
1 0 2
7 4 3
5 8 6
Depth - 1

RIGHT
1 2 0
7 4 3
5 8 6
Depth - 2

DOWN
1 2 3
7 4 0
5 8 6
Depth - 3

UP
1 2 0
7 4 3
5 8 6
Depth - 4

DOWN
1 2 3
7 4 0
5 8 6
Depth - 5

UP
1 2 0
7 4 3
5 8 6
Depth - 6

DOWN
1 2 3
7 4 0
5 8 6
Depth - 7

UP
1 2 0
7 4 3
5 8 6
Depth - 8

DOWN
1 2 3
7 4 0
5 8 6
Depth - 9

UP
1 2 0
7 4 3
5 8 6
Depth - 10

DOWN
1 2 3
7 4 0
5 8 6
Depth - 11

UP
1 2 0
7 4 3
5 8 6
Depth - 12

DOWN
1 2 3
7 4 0
5 8 6
Depth - 13

UP
1 2 0
7 4 3
5 8 6
Depth - 14

DOWN
1 2 3
7 4 0
5 8 6
Depth - 15
```



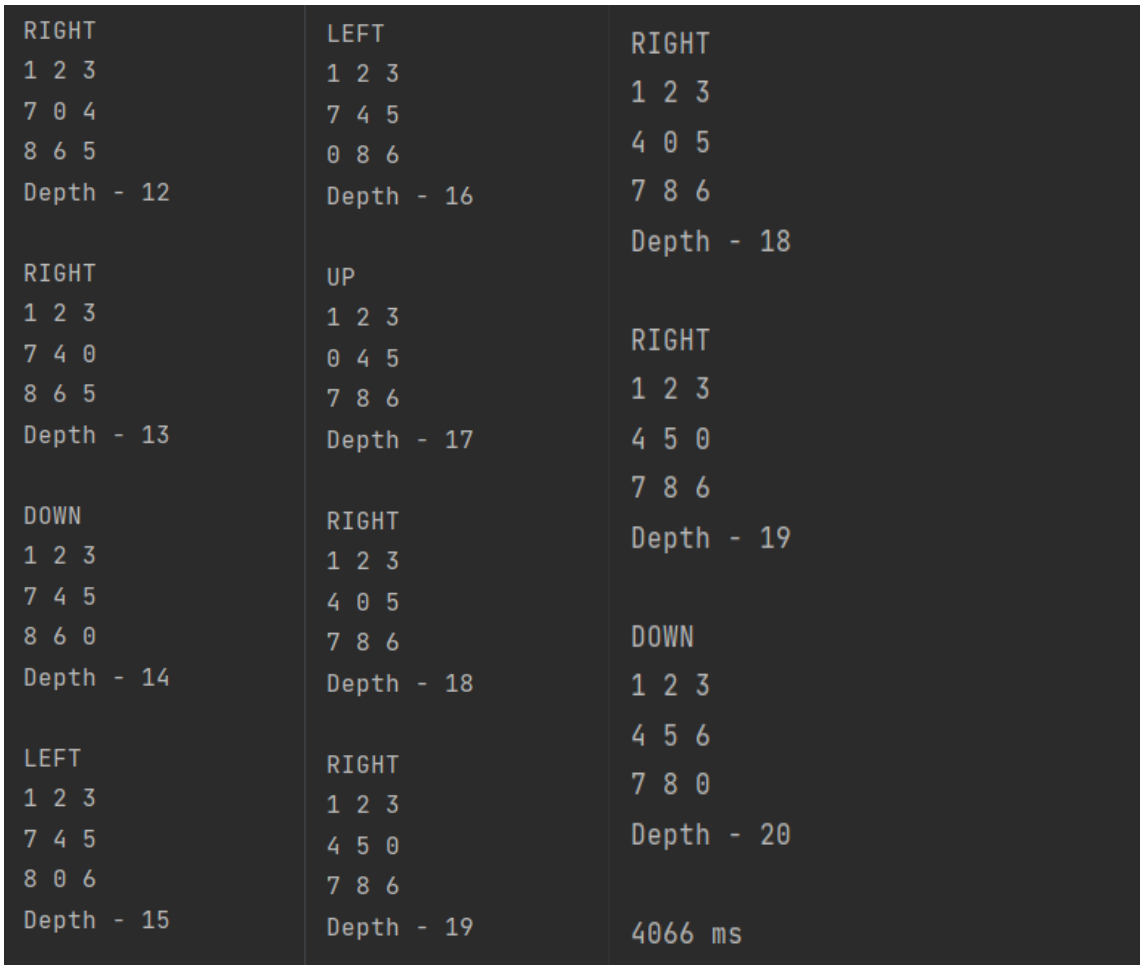
Рисунки 3.1 – Алгоритм LDFS

RBFS

```
INITIAL
7 8 3
2 0 4
6 1 5
Depth - 0

UP
7 0 3
2 8 4
6 1 5
Depth - 1

LEFT
0 7 3
2 8 4
6 1 5
Depth - 2

DOWN
2 7 3
0 8 4
6 1 5
Depth - 3
```

```
RIGHT
2 7 3
8 0 4
6 1 5
Depth - 4

DOWN
2 7 3
8 1 4
6 0 5
Depth - 5

LEFT
2 7 3
8 1 4
0 6 5
Depth - 6

UP
2 7 3
0 1 4
8 6 5
Depth - 7
```

```
RIGHT
2 7 3
1 0 4
8 6 5
Depth - 8

UP
2 0 3
1 7 4
8 6 5
Depth - 9

LEFT
0 2 3
1 7 4
8 6 5
Depth - 10

DOWN
1 2 3
0 7 4
8 6 5
Depth - 11
```



Рисунки 3.2 – Алгоритм RBFS

- Дослідження алгоритмів

LDFS

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

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS з обмеженням глибини 40

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
2 5 8 1 0 4 3 6 7	5840361865	1	78667	27
1 4 5 3 0 2 7 6 8	1397140113	0	51681	45
3 2 6 8 4 7 5 0 1	4729456177	1	71912	23
5 6 7 2 1 4 3 8 0	4357579525	1	76884	24
5 6 1 4 0 3 8 7 2	244176486	0	29457	43
8 3 2	2419091987	0	59512	42

4 1 0 6 5 7				
0 6 5 3 8 1 4 2 7	4055424173	0	71821	46
5 6 2 1 7 0 4 3 8	82345509	0	18404	35
2 5 8 1 0 4 3 6 7	5840361865	0	17603	37
7 3 6 8 2 4 0 1 5	4566402008	1	76884	22
6 1 2 4 3 7 8 0 5	4820939326	1	71912	26
5 2 7 0 6 4 3 8 1	4510947873	1	71912	33
1 6 8 5 2 3 4 0 7	13518233	0	9598	39
2 1 7 4 6 5 8 0 3	99709758	0	20903	40
3 8 4	1044090754	0	49053	45

2 1 6 5 0 7				
1 2 8 6 0 4 7 5 3	2487672807	0	65633	48
6 5 7 8 3 2 1 0 4	4851693914	1	71912	38
2 0 6 8 3 4 5 7 1	4585600475	1	71912	13
2 3 7 5 4 1 8 0 6	184913719	0	26792	44
5 4 7 1 3 8 0 2 6	4586663607	1	76884	19

RBFS

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

Таблиця 3.3 – Характеристики оцінювання алгоритму RBFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
3 7 8 6 4 5 1 0 2	8608709	0	12333	47

4 8 1 7 2 0 3 6 5	2997880	0	7073	42
7 3 5 8 6 0 1 2 4	2447038	0	6516	44
2 6 3 1 4 0 7 8 5	51708	0	645	31
2 3 0 1 7 8 4 5 6	22926	0	779	30
3 0 6 8 7 4 5 2 1	1148005863	0	66497	55
8 3 0 2 7 1 5 6 4	71019689	0	26436	46
3 0 5 2 8 4 7 6 1	674977	0	3264	37
2 3 8 5 4 1 6 7 0	6979495	0	10144	45
5 1 7 2 0 4 8 6 3	14343462	0	12995	47

4 1 3 7 8 0 2 5 6	236737	0	1548	32
5 8 3 6 1 2 7 0 4	1674872362	1	58148	25
3 2 7 8 6 0 4 5 1	249752123	0	30741	52
1 4 2 0 6 3 8 5 7	5193523	0	6731	42
0 5 2 4 8 1 6 3 7	8745343	0	10724	43
5 8 0 4 2 6 7 1 3	35741964	0	11178	43
3 8 2 1 0 5 4 6 7	266332	0	2498	37
6 5 0 8 1 7 3 2 4	1618415625	0	80840	54
7 4 6 5 2 8 3 1 0	1079622	0	4271	40

6 1 3 7 4 0 5 8 2	4290929	0	6443	40

ВИСНОВОК

На лабораторній роботі було розглянуто алгоритм неінформативного пошуку LDFS (пошук з обжеменням глибини), а також алгоритм інформативного пошуку RBFS (рекурсивний пошук за першим найкращим). У ході даної роботи було досліджено ці алгоритми, а також проведено експерименти, в результаті яких було знайдено такі середні значення:

- Для алгоритму LDFS
  - Середня кількість ітерацій: 3035904508,7
  - Середня кількість станів: 54466,8
  - Середня кількість станів у пам’яті: 34,45
  - Середня кількість разів, коли алгоритм заходив у глухий кут: 0,45
- Для алгоритму RBFS
  - Середня кількість ітерацій: 242687315,35
  - Середня кількість станів: 17990,2
  - Середня кількість станів у пам’яті: 41,6
  - Середня кількість разів, коли алгоритм заходив у глухий кут: 0,05

Порівнюючи алгоритми можемо дійти висновку, що алгоритм LDFS потребує значно більше ітерацій у порівнянні з RBFS. Різниця становить 2 793 217 193,35 одиниць. Також перший генерує набагато більше станів у порівнянні з другим. Різниця становить 36 476,6 одиниць. Однак середня кількість станів, що зберігається у пам’ять у алгоритма LDFS трохи краще. Різниця становить 7,15 одиниць. Оскільки алгоритм неінформативного пошуку є неповним (при умові, що ліміт глибини є меншим ніж мінімальна глибина рішення), то середня кількість разів, коли LDFS заходив у глухий кут

є досить великою і становить 0,45, в той час як показник для алгоритму інформативного пошуку становить лише 0,05. Отже, можна зробити висновок, що алгоритм інформативного пошуку (RBFS) є набагато ефективніше ніж алгоритм (LDFS).