

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-311 Химич Володимир Леонідович

(шифр, прізвище, ім'я, по батькові)



Перевірів

Головченко М.М.

(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1	МЕТА ЛАБОРАТОРНОЇ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ	8
3.1	ПСЕВДОКОД АЛГОРИТМІВ	8
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ	8
3.2.1	<i>Вихідний код</i>	8
3.2.2	<i>Приклади роботи</i>	8
3.3	ДОСЛІДЖЕННЯ АЛГОРИТМІВ	8
	ВИСНОВОК	11
	КРИТЕРІЇ ОЦІНЮВАННЯ	12

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 **АП**, 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 відрізнятися 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 – переміщуючи пластинки по коробці досягти впорядкування 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 А може стояти на одній лінії з ферзем В, проте між ними стоїть ферзь С; тому А не б'є В).

- **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: $T = 1000 - k \cdot t$, 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	АП	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

2 2	8-puzzle	LDFS	RBFS		H2
2 3	8-puzzle	BFS	A*		H1
2 4	8-puzzle	BFS	A*		H2
2 5	8-puzzle	BFS	RBFS		H1
2 6	8-puzzle	BFS	RBFS		H2
2 7	Лабіринт	BFS	A*		H3
2 8	8-puzzle	IDS	A*		H2
2 9	8-puzzle	IDS	RBFS		H1
3 0	8-puzzle	IDS	RBFS		H2
3 1	COLOR			HILL	MRV
3 2	COLOR			ANNEAL	MRV
3 3	COLOR			BEAM	MRV
3 4	COLOR			HILL	DGR
3 5	COLOR			ANNEAL	DGR

3	COLOR			BEAM	DGR
6					

3 ВИКОНАННЯ

1.1 Псевдокод алгоритмів

Limit Depth First Search

Function Depth-Limited-Search (problem, limit) returns рішення result або індикатор невдачі failure\cutoff

*Return Recursive-DLS(Make-Node(Initial-State[problem]),
Problem, limit)*

Function Recursive-DLS(node, problem, limit) returns рішення result або індикатор невдачі failure\cutoff

cutoff_occurred? ← неправдиве значення
if Goal-Test[problem](State[node]) then return Solution(node)
else if Depth[node] = limit then return індикатор невдачі cutoff
else for each спадкоємець successor in Expand(node, problem) do
result ← Recursive-DLS(successor, problem, limit)
if result = cutoff then cutoff_occured? ← правдиве значення
else if result != failure then return рішення result
if cutoff_occurred?
Then return індикатор невдачі cutoff
Else return індикатор невдачі failure

Recursive Best First Search

function Recursive-Best-First-Search(problem) **returns** рішення result
або індикатор невдачі failure
RBFS(problem, Make-Node(Initial-State[problem]), ∞)

function RBFS(problem, node, f_limit) **returns** рішення result
або індикатор невдачі failure і нова межа f-вартості f_limit
if Goal-Test[problem](State[node]) **then return** узел node
successors ← Expand(node, problem)
if множина вузлів спадкоємців successors пуста
 then return failure, ∞
for each s **in** successors **do**
 f[s] ← max(g(s)+h(s), f[node])
repeat
 best ← вузол з найменшим f-значенням у множині successors
 if f[best] > f_limit **then return** failure, f[best]
 alternative ← наступне після найменшого f-значення
 у множині successors
 result, f[best] ← RBFS(problem, best,
 min(f_limit, alternative))
if result ≠ failure **then return** result

1.2 Програмна реалізація

1.2.1 Вихідний код

Limit Depth First Search

```
package secondLab.algorithms;
import lombok.Getter;
import secondLab.entity.Node;
import secondLab.entity.Result;
import secondLab.entity.Statistic;

import java.util.Iterator;

import static secondLab.entity.ResultCodes.*;
import static secondLab.util.Utility.*;

@Getter
public class LimitDepthFirstSearch {
    private static final byte MAX_DEPTH = 8;
    private static Statistic statistic;
    private static int iterations;
    private static long startTime;

    public static Statistic runLimitDepthFirstSearch() {
        statistic = new Statistic();
        iterations = 0;

        byte[] problem = createProblem();
        Node root = new Node(problem, (byte) 0);
        statistic.setInitialStateNode(root);

        var limitDepthFirstSearch = new LimitDepthFirstSearch();
        startTime = System.nanoTime();
        Result result = limitDepthFirstSearch.search(problem, MAX_DEPTH);

        statistic.setResult(result);
        statistic.setIterations(iterations);
        statistic.setConsumedTime(startTime - System.nanoTime());
        return statistic;
    }
}
```

```

public Result search(byte[] problem, int limit) {
    Result result = recursiveSearch(new Node(problem, (byte) 0), limit);
    if (result.isSolution()) {
        return result;
    } else {
        return Result.of(FAILURE, null);
    }
}

private Result recursiveSearch(Node parent, int limit) {
    if (timeOut(startTime) || memoryLimitReached()){
        return Result.of(TERMINATED, null);
    }
    iterations++;

    boolean cutoffOccurred = false;
    if (parent.isSolution()) {
        return Result.of(SOLUTION, parent);
    } else if (parent.getDepth() == limit) {
        statistic.incrementEndMeet();
        return Result.of(CUT_OFF, null);
    } else {
        for (Iterator<Node> iterator = createChildren(parent, statistic).iterator();
iterator.hasNext(); ) {
            Node child = iterator.next();
            Result result = recursiveSearch(child, limit);

            if (result.isCutOff()) {
                cutoffOccurred = true;
            } else if (!result.isFailure()) {
                return result;
            }

            statistic.decrementChildrenInMemory();
            iterator.remove();
        }
    }

    if (cutoffOccurred)
        return Result.of(CUT_OFF, null);
}

```

```

        else
            return Result.of(FAILURE, null);
    }
}

```

Recursive Best First Search

```

package secondLab.algorithms;

import secondLab.entity.Node;
import secondLab.entity.Result;
import secondLab.entity.Statistic;

import java.util.Comparator;
import java.util.LinkedList;

import static java.lang.Math.min;
import static secondLab.entity.ResultCodes.FAILURE;
import static secondLab.entity.ResultCodes.SOLUTION;
import static secondLab.util.Utility.*;

public class RecursiveBestFirstSearch {
    private static long startTime;
    private static Statistic statistic;

    public static Statistic runRecursiveBestFirstSearch() {
        statistic = new Statistic();

        byte[] problem = createProblem();
        Node root = new Node(problem, (byte) 0);

        statistic.setInitialStateNode(root);

        RecursiveBestFirstSearch recursiveBestFirstSearch = new
RecursiveBestFirstSearch();

        startTime = System.nanoTime();
        Result result = recursiveBestFirstSearch.search(root, 10000);

        statistic.setConsumedTime(startTime - System.nanoTime());
        statistic.setResult(result);
        return statistic;
    }
}

```

```

}

public Result search(Node parent, int bestStepValue) {

    Result result = recursiveSearch(parent, bestStepValue);
    if (result.isSolution()) {
        return result;
    } else {
        return Result.of(FAILURE, null);
    }
}

private Result recursiveSearch(Node parent, int bestStepValue) {
    statistic.incrementIteration();

    if (parent.isSolution())
        return Result.of(SOLUTION, parent);

    LinkedList<Node> children = createChildren(parent, statistic);

    while (true) {
        children.sort(Comparator.comparing(Node::getStepValue));

        Node best = children.get(0);

        if (best.getStepValue() > bestStepValue) {
            statistic.incrementEndMeet();
            return Result.of(best.getStepValue(), FAILURE, null);
        }

        Node alternative = children.get(1);

        Result result = recursiveSearch(best, min(bestStepValue,
alternative.getStepValue()));
        best.setStepValue(result.getBestF());

        if (!result.isFailure()) {
            return result;
        }
    }
}
}

```

Решту **допоміжного коду** можна глянути в репозиторію із лабораторними роботами:

<https://github.com/vovik541/algorithms-third-session/tree/master/src/main/java/secondLab>

1.2.2 Приклади роботи

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

Рисунок 3.1 – Алгоритм **Limit Depth First Search**

```
LDFS: beginning state
□ □ Q □ □ □ □ □ □
□ □ □ □ □ □ □ Q □
□ □ □ □ □ □ □ Q □
□ □ Q □ □ □ □ □ □
□ □ Q □ □ □ □ □ □
□ □ □ □ Q □ □ □ □
□ Q □ □ □ □ □ □ □
□ □ □ Q □ □ □ □ □

After search run:
□ □ □ □ □ Q □ □ □
□ □ □ Q □ □ □ □ □
□ □ □ □ □ □ □ Q □
□ □ Q □ □ □ □ □ □
□ □ □ □ □ □ □ Q □
□ □ □ □ Q □ □ □ □
□ Q □ □ □ □ □ □ □
□ □ □ Q □ □ □ □ □

Iterations: 1279146
States created: 1279488
States in memory: 351
The end node met: 1256297
Worked for 0.287 seconds

Process finished with exit code 0
```

Рисунок 3.2 – Алгоритм **Recursive Best First Search**

```
RBFS: beginning state
□ □ Q □ □ □ □ □ □
□ □ □ □ □ □ Q □
□ □ □ □ □ □ Q □
□ □ □ Q □ □ □ □
□ □ □ Q □ □ □ □
□ □ □ □ □ Q □ □
□ Q □ □ □ □ □ □
□ □ □ □ □ □ Q □

After search run:
□ □ □ □ Q □ □ □
□ □ Q □ □ □ □ □
□ □ □ □ □ □ Q □
□ □ □ Q □ □ □ □
□ □ □ □ □ Q □ □
□ □ □ □ □ □ Q □
□ Q □ □ □ □ □ □
□ □ □ □ □ Q □ □

Iterations: 101
States created: 5600
States in memory: 5600
The end node met: 95
Worked for 0.008 seconds

Process finished with exit code 0
```

1.3 Дослідження алгоритмів

Алгоритм RBFS працює швидше, ніж LDFS. Він створює менше станів і виконує роботу за значно меншу кількість ітерацій (в сотні тисяч, а не в мільйони разів).

Переваги Depth Limited Search

1. Пошук з обмеженою глибиною є більш ефективним, ніж DFS, використовує менше часу та пам'яті.
2. Якщо рішення існує, DFS гарантує, що воно буде знайдено за кінцевий проміжок часу.
3. Щоб усунути недоліки DFS, ми встановлюємо обмеження глибини та неодноразово запускаємо нашу техніку пошуку в дереві пошуку.
4. DLS має застосування в теорії графів, які можна порівняти з DFS.

Недоліки пошуку з обмеженою глибиною

1. Щоб цей метод працював, він повинен мати обмеження глибини.
2. Якщо цільовий вузол не існує в межах вибраного обмеження глибини, користувач буде змушений повторити повторення, збільшуючи час виконання.
3. Якщо цільовий вузол не існує в межах зазначеного ліміту, його не буде виявлено.

Часова складність LDFS $O(b^l)$

Переваги Recursive Best First Search

1. Ефективніше, ніж IDA*
2. Це оптимальний алгоритм, якщо $h(n)$ допустимо
3. Просторова складність дорівнює $O(bd)$.

Недоліки Recursive Best First Search

1. Страждає від надмірної регенерації вузлів.
2. *Його складність у часі важко охарактеризувати*, оскільки вона залежить від точності $h(n)$ і того, як часто найкращий шлях змінюється під час розширення вузлів.

В таблиці 3.1 наведені характеристики оцінювання алгоритму **Limit Depth First Search** задачі **8 ферзів** для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання **Limit Depth First Search**

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом'яті
Стан 1 [2, 6, 4, 4, 6, 4, 0, 2]	24237399	23804582	24237696	306
Стан 2 [2, 1, 0, 4, 6, 4, 3, 1]	12953678	12722356	12953976	307
Стан 3 [0, 2, 6, 7, 1, 7, 1, 3]	49473	48582	49840	376
Стан 4 [4, 1, 6, 2, 1, 0, 3, 4]	458993	450790	459312	328
Стан 5 [0, 4, 4, 7, 4, 3, 6, 0]	13835	13580	14224	398
Стан 6 [4, 4, 7, 2, 2, 6, 1, 5]	1239	1209	1624	394
Стан 7 [5, 3, 7, 0, 6, 5, 5, 4]	21526299	21141894	21526624	334
Стан 8 [1, 6, 5, 2, 3, 6, 7, 7]	11727047	11517629	11727352	314
Стан 9 [2, 5, 2, 3, 4, 1, 6, 7]	113622358	111593381	113622656	307
Стан 10 [3, 3, 4, 5, 3, 4, 2, 1]	123811107	121600188	123811408	310
Стан 11 [6, 3, 4, 2, 4, 7, 5, 3]	22406817	22006688	22407168	360

Стан 12 [4, 2, 2, 1, 3, 6, 2, 5]	578998	568651	579376	387
Стан 13 [5, 2, 6, 1, 2, 0, 4, 3]	640823	629373	641144	330
Стан 14 [6, 3, 5, 7, 0, 1, 4, 1]	11493031	11287791	11493384	362
Стан 15 [3, 5, 0, 5, 5, 4, 0, 6]	2068623	2031677	2068920	306
Стан 16 [7, 6, 6, 7, 3, 5, 7, 6]	3472694	3410675	3473008	323
Стан 17 [0, 3, 0, 3, 6, 2, 3, 3]	104021467	102163935	104021736	278
Стан 18 [0, 0, 4, 1, 4, 7, 1, 6]	11499545	11294189	11499880	344
Стан 19 [6, 4, 3, 3, 2, 4, 7, 4]	32059391	31486895	32059720	338
Стан 20 [4, 2, 4, 0, 5, 1, 3, 5]	21506912	21122853	21507248	345
Середнє	25907474	25444836	25907807	328

В таблиці 3.2 наведені характеристики оцінювання алгоритму **Recursive Best First Search** задачі **8 ферзів** для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання **Recursive Best First Search**

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом'яті
Стан 1 [6, 0, 2, 3, 7, 0, 1, 0]	8	2	392	392

Стан 2 [6, 2, 6, 5, 5, 2, 5, 5]	181	174	10080	10080
Стан 3 [1, 1, 5, 7, 7, 0, 4, 3]	34	28	1848	1848
Стан 4 [6, 6, 5, 6, 7, 5, 4, 0]	9	2	448	448
Стан 5 [1, 7, 0, 2, 5, 6, 0, 0]	141	135	7840	7840
Стан 6 [1, 1, 4, 1, 5, 4, 6, 4]	24	18	1288	1288
Стан 7 [0, 2, 7, 3, 2, 4, 5, 6]	18	11	952	952
Стан 8 [7, 3, 3, 7, 5, 7, 0, 7]	9	3	448	448
Стан 9 [3, 7, 6, 1, 1, 7, 6, 0]	19	13	1008	1008
Стан 10 [3, 6, 2, 7, 1, 6, 3, 4]	4	0	168	168
Стан 11 [0, 1, 6, 0, 2, 0, 7, 0]	8	3	392	392
Стан 12 [2, 3, 0, 6, 5, 7, 2, 2]	6	0	280	280
Стан 13 [5, 2, 7, 7, 3, 7, 3, 6]	4	0	168	168
Стан 14 [2, 5, 7, 6, 6, 4, 4, 0]	119	113	6608	6608
Стан 15 [5, 3, 1, 3, 6, 6, 0, 5]	13	8	672	672

Стан 16 [2, 6, 2, 4, 2, 1, 0, 0]	6	0	280	280
Стан 17 [6, 1, 1, 6, 1, 7, 7, 2]	51	45	2800	2800
Стан 18 [3, 0, 5, 2, 5, 6, 7, 7]	6	0	280	280
Стан 19 [4, 2, 5, 3, 2, 6, 2, 4]	8	2	392	392
Стан 20 [4, 6, 1, 4, 4, 2, 2, 2]	7	2	336	336
Середнє	25	22	1827	1827

Висновок

При виконанні даної лабораторної роботи було розглянуто інформативні та не інформативні алгоритми. Було імплементовано Limit Depth First Search та Recursive Best First Search, передбачено збір статистики, обмежень у використанні пам'яті та часі виконання алгоритмів. Було порівняно та протестовано різні алгоритми по багатьом критеріям. Також, в ході виконання роботи, я більш детально ознайомився із роботою у пам'яті, оптимізував алгоритми у найкращий доступний спосіб. Із загальних висновків було прийнято до уваги те, на скільки суттєвою є різниця в часі пошуку рішення різними алгоритмами. Також було вирішено n-повну задачу 8-ферзів.

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%.