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 Пархомчук Ілля Вікторович* _____
(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
22	8-puzzle	LDFS	RBFS		H2

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

LDFS(стан s, глибина)

1. **Якщо** s є кінцевим, **то**
   1.1. **Повернути** успіх
2. **Якщо** глибина > максимум, **то**
   2.1. **Повернути** невдача
3. **for** і **від** 0 **до** кількість можливих ходів з стану s
   3.1. next = s;
   3.2. зробити хід і у стані next
   3.3. **Якщо** LDFS(next, глибина + 1) == успіх, **то**
      3.3.1. **Повернути** успіх
4. **Повернути** невдача

RBFS(стан s, ціна стану s, максимальна ціна)

1. **Якщо** s є кінцевим, то
   1.1. **Повернути** успіх
2. **for** і **від** 0 **до** кількість можливих ходів з стану s
   2.1. next = s;
   2.2. зробити хід і у стані next
   2.3. додати next у можливі стани
   2.4. ціна next = h() + g()
3. **while** true
   3.1. best = стан з найменшою ціною в множині можливих станів
   3.2. **Якщо** ціна best > максимальна ціна, **то**
      3.2.1. ціна стану s = ціна стану best
      3.2.2. **Повернути** невдача
   3.3. **Якщо** RBFS(best, ціна best, min(максимальна ціна, ціна другого найменшого стану)) == успіх, **то**
      3.3.1. **Повернути** успіх

3.2 Програмна реалізація

3.2.1 Вихідний код

General.h

```
#pragma once
#include <forward_list>

class State;

enum class Result
{
	SUCCES,
	FAIL,
	LIMIT
};

struct Token
```

```
{
	int pos_x;
	int pos_y;
};

struct info
{
	int64_t state_count = 1;
	int64_t dead_end = 0;
	int64_t iterations = 0;
	int states_in_memory = 1;
	int depth = 0;
	std::forward_list<State> history;
};
```

State.h

```
class State
{
	friend Result RecursiveLDFS(info& i);
	friend int h(State& s);
	friend Result RBSF(info& info, int limit, int* prev_best);

public:
	static constexpr int TOKEN_COUNT = 9;

	enum class Move
	{
		UP,
		DOWN,
		LEFT,
		RIGHT
	};

private:
	Move Moves[4];
	int moves_avail;

	int empty_pos;
	char field[TOKEN_COUNT];

	void CalcMovesAvail();
	bool IsSolvable();
public:
	State();
	State(const State&) = default;
	~State() = default;
	bool operator==(const State& A) const;

	void PrintState();
	void PrintMovesAvail();
	void EditState();

	bool IsFinal();
	void DoMove(Move m);
};
```

State.cpp

```
#include "State.h"
#include <vector>
#include <iostream>
#include <random>

using std::vector;
using std::cout;
using std::cin;
```

```
State::State()
{
	vector<int> UNUM(TOKEN_COUNT);

	do
	{
		UNUM.resize(TOKEN_COUNT);
		for (int i = 0; i < TOKEN_COUNT - 1; i++)
		{
			UNUM[i] = i;
		}
		UNUM[TOKEN_COUNT - 1] = ' ' - '1';

		for (int i = 0; i < TOKEN_COUNT; i++)
		{
			int ind = rand() % UNUM.size();
			field[i] = '1' + UNUM[ind];
			UNUM.erase(UNUM.begin() + ind);
		}
	} while (!IsSolvable());

	empty_pos = std::find(field, field + TOKEN_COUNT, ' ') - field;
	CalcMovesAvail();
}

bool State::operator==(const State& A) const
{
	for (size_t i = 0; i < TOKEN_COUNT; i++)
	{
		if (this->field[i] != A.field[i])
			return false;
	}
	return true;
}

void State::CalcMovesAvail()
{
	moves_avail = 0;
	size_t val = empty_pos + 1;
	size_t cmp = val % 3;
	if (val < 7)
		Moves[moves_avail++] = Move::UP;
	if (2 == cmp)
	{
		Moves[moves_avail++] = Move::LEFT;
		Moves[moves_avail++] = Move::RIGHT;
	}
	else
		Moves[moves_avail++] = cmp ? Move::LEFT : Move::RIGHT;

	if (3 < val)
		Moves[moves_avail++] = Move::DOWN;

}

bool State::IsSolvable()
{
	int inv_count = 0;
	for (size_t i = 0; i < TOKEN_COUNT-1; i++)
	{
		for (size_t k = i+1; k < TOKEN_COUNT; k++)
		{
			if (field[i] > field[k] &&
				field[i] != ' ' &&
				field[k] != ' ')
				++inv_count;
		}
```

```
        }
        return !(inv_count % 2);
}

void State::PrintState()
{
        const int BORDER_LEN = 2 + TOKEN_COUNT;
        for (int i = 0; i < BORDER_LEN; i++)
                cout << "*";
        cout << '\n';
        for (size_t i = 0; i < 3; i++)
        {
                cout << '*';
                for (size_t k = 0; k < 3; k++)
                {
                        cout << '|' << field[i * 3 + k] << '|';
                }
                cout << "*\n";

        }

        for (int i = 0; i < BORDER_LEN; i++)
                cout << "*";

        cout << '\n';
}

void State::PrintMovesAvail()
{
         auto move_name = [](State::Move m) {
                switch (m)
                {
                case State::Move::UP: return "UP";
                case State::Move::DOWN: return "DOWN";
                case State::Move::LEFT: return "LEFT";
                case State::Move::RIGHT: return "RIGHT";
                }
        };

        cout << "Moves available:" << moves_avail << '\n';
        for (size_t i = 0; i < moves_avail; i++)
        {
                cout << move_name(Moves[i]) << '\n';
        }
}

void State::EditState()
{
        auto is_correct = [this](char field[TOKEN_COUNT]) {
                using std::find;
                char* field_end = field + this->TOKEN_COUNT;
                if (field_end == find(field, field_end, ' '))
                        return false;
                for (size_t i = 0; i < this->TOKEN_COUNT - 1; i++)
                {
                        if (field_end == find(field, field_end, '1' + i))
                                return false;
                }
                return true;
        };
        cout << "Enter the matrix:\n";

        do
        {
                for (size_t i = 0; i < TOKEN_COUNT; ++i)
                {
                        field[i] = cin.get();
                        if (cin.peek() == '\n')
```

```
                        cin.ignore();
                }

                PrintState();

                if (!is_correct(field))
                {
                        cout << "Data entered incorrect. Try again\n";
                }
                else if (!IsSolvable())
                {
                        cout << "Entered State is unsolvable. Try again\n";
                }
                else
                {
                        empty_pos = std::find(field, field + TOKEN_COUNT, ' ') - field;
                        CalcMovesAvail();
                        break;
                }
        } while (true);
}

bool State::IsFinal()
{
        if (field[0] == '1' &&
                field[1] == '2' &&
                field[2] == '3' &&
                field[3] == '4' &&
                field[4] == '5' &&
                field[5] == '6' &&
                field[6] == '7' &&
                field[7] == '8')
                return true;

        return false;
}

void State::DoMove(Move m)
{
        switch (m)
        {
        case State::Move::UP:
                std::swap(field[empty_pos], field[empty_pos + 3]);
                empty_pos += 3;
                break;
        case State::Move::DOWN:
                std::swap(field[empty_pos], field[empty_pos - 3]);
                empty_pos -= 3;
                break;
        case State::Move::LEFT:
                std::swap(field[empty_pos], field[empty_pos + 1]);
                ++empty_pos;
                break;
        case State::Move::RIGHT:
                std::swap(field[empty_pos], field[empty_pos - 1]);
                --empty_pos;
                break;
        }
        CalcMovesAvail();
}
```

SearchAlgs.cpp

```
#include "SearchAlgs.h"
#include <iostream>
#include "State.h"
#include <algorithm>
#include <map>
#include <vector>
```

```
using std::cout;

void PrintResults(Result res, info& inf)
{
	switch (res)
	{
	case Result::FAIL: cout << "Fail\n"; break;
	case Result::SUCCES: cout << "\n\nSucces\n"; break;
	}
	std::vector<State> order(inf.history.begin(), inf.history.end());

	for (auto s = order.rbegin(); s < order.rend(); ++s)
	{
		(*s).PrintState();
	}
	cout << "State count: " << inf.state_count << '\n';
	cout << "States in memory: " << inf.states_in_memory << '\n';
	cout << "Iterations: " << inf.iterations << '\n';
	cout << "Dead ends: " << inf.dead_end << '\n';
	cout << "Depth: " << inf.depth << '\n';
}

void LimitedDFS(State s)
{
	info inf;
	inf.history.push_front(s);

	PrintResults(RecursiveLDFS(inf),inf);
}


Result RecursiveLDFS(info& info)
{
	++info.iterations;
	auto& top = info.history.front();
	if (top.IsFinal())
		return Result::SUCCES;
	else if (info.depth == LDFS_DEPTH_LIMIT)
	{
		++info.dead_end;
		return Result::LIMIT;
	}

	for (size_t i = 0; i < top.moves_avail; i++)
	{
		info.history.push_front(top);
		++info.depth;
		++info.state_count;
		++info.states_in_memory;
		info.history.front().DoMove(top.Moves[i]);

		if (RecursiveLDFS(info) != Result::SUCCES)
		{
			info.history.pop_front();
			--info.states_in_memory;
			--info.depth;
		}
		else
			return Result::SUCCES;
	}

	return Result::FAIL;

}
```

```
void RecursiveBestFS(State s)
{
	info inf;
	inf.history.push_front(s);

	PrintResults(RBSF(inf, std::numeric_limits<int>::max(), nullptr),inf);
}

int h(State& s)
{
	static std::map<char, Token> TableFin
	{ {'1',{0,0} },
	{'2',{1,0} },
	{'3',{2,0} },
	{'4',{0,1} },
	{'5',{1,1} },
	{'6',{2,1} },
	{'7',{0,2} },
	{'8',{1,2} },
	{' ',{2,2} } };

	auto get_token_pos = [](int linear_pos) {
		Token res;
		res.pos_x = linear_pos % 3;
		res.pos_y = linear_pos / 3;
		return res;
	};
	auto manhatten_lengt = [](const Token cur, const Token fin) -> int {
		return abs(cur.pos_x - fin.pos_x) + abs(cur.pos_y - fin.pos_y);
	};

	int res = 0;

	for (size_t k = 0; k < State::TOKEN_COUNT; k++)
	{
		if (s.field[k] == ' ')
			continue;
		res += manhatten_lengt(get_token_pos(k), TableFin[s.field[k]]);
	}
	return res;
}

Result RBSF(info& info, int limit, int* prev_best)
{
	typedef std::pair<State, int > successor_cost;

	++info.iterations;
	auto& top = info.history.front();

	if (top.IsFinal())
		return Result::SUCCES;

	successor_cost succesors[4];
	int s_count = 0;
	for (size_t i = 0; i < top.moves_avail; i++)
	{
		succesors[s_count].first = top;
		succesors[s_count].first.DoMove(top.Moves[i]);
		if (std::find(info.history.begin(), info.history.end(),
succesors[s_count].first) != info.history.end())
		{
			continue;
		}
		succesors[s_count].second = h(succesors[s_count].first) + info.depth;
```

```
			++s_count;
		}
		info.state_count += s_count;
		info.states_in_memory += s_count;
		if (0 == s_count)
		{
			*prev_best = std::numeric_limits<int>::max();
			++info.dead_end;
			return Result::FAIL;
		}
		if (1 == s_count)
		{
			info.history.push_front(succesors[0].first);
			++info.depth;
			auto res = RBSF(info, limit, prev_best);
			if (res == Result::SUCCES)
				return res;

			--info.depth;
			--info.states_in_memory;
			info.history.pop_front();
			return res;
		}
		while (true)
		{
			std::sort(succesors, succesors + s_count,
				[](const successor_cost A, const successor_cost B)
				{ return A.second < B.second; });

			successor_cost& best = succesors[0];
			if (best.second > limit)
			{
				*prev_best = best.second;
				++info.dead_end;
				info.states_in_memory -= s_count;
				return Result::FAIL;
			}

			info.history.push_front(best.first);
			++info.depth;

			if (RBSF(info, std::min(limit, succesors[1].second), &best.second) ==
Result::SUCCES)
			{
				return Result::SUCCES;
			}
			--info.depth;
			info.history.pop_front();
		}
}
```

3.2.2 Приклади роботи

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

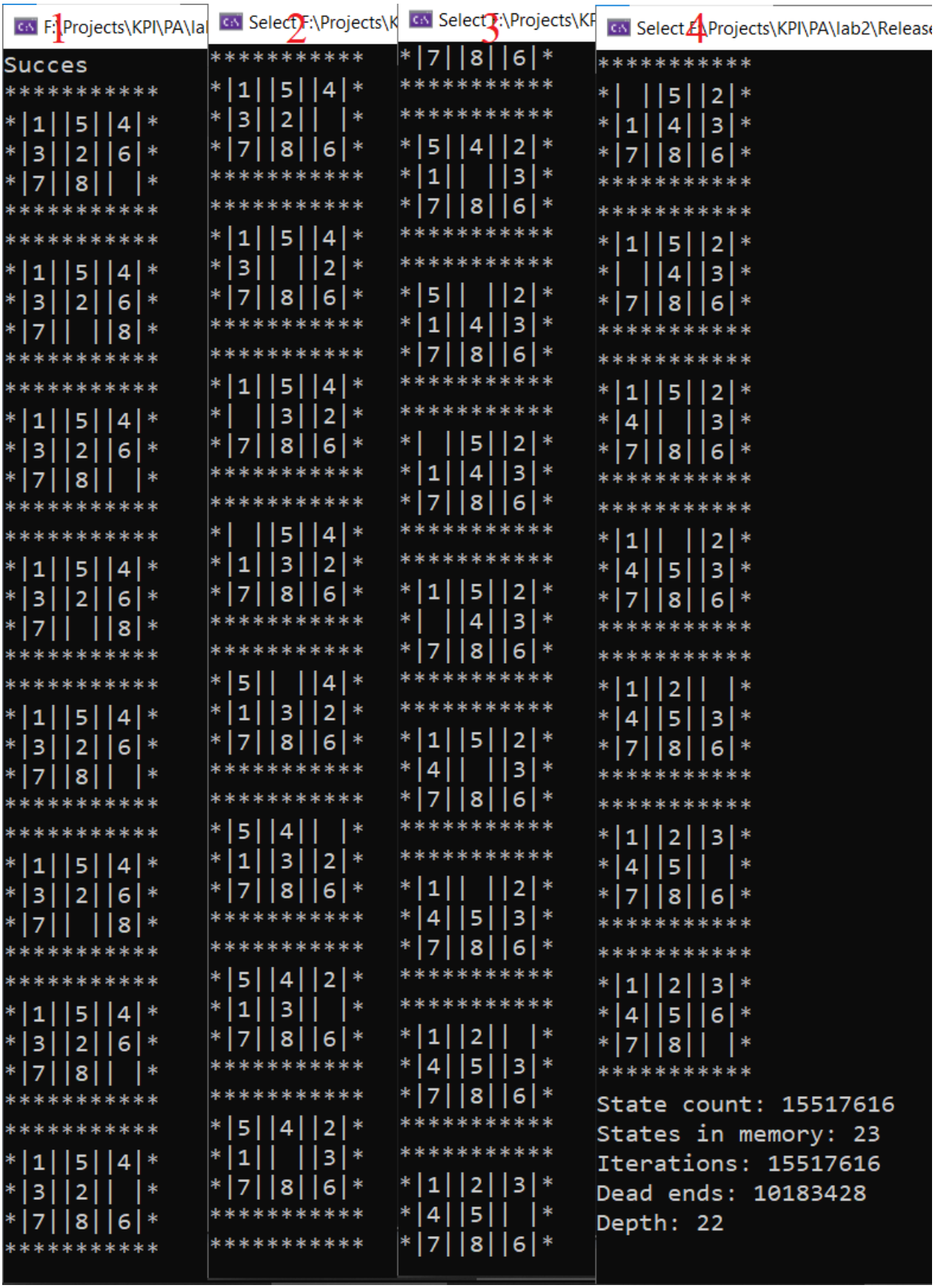

Рисунок 3.1 – Алгоритм LDFS

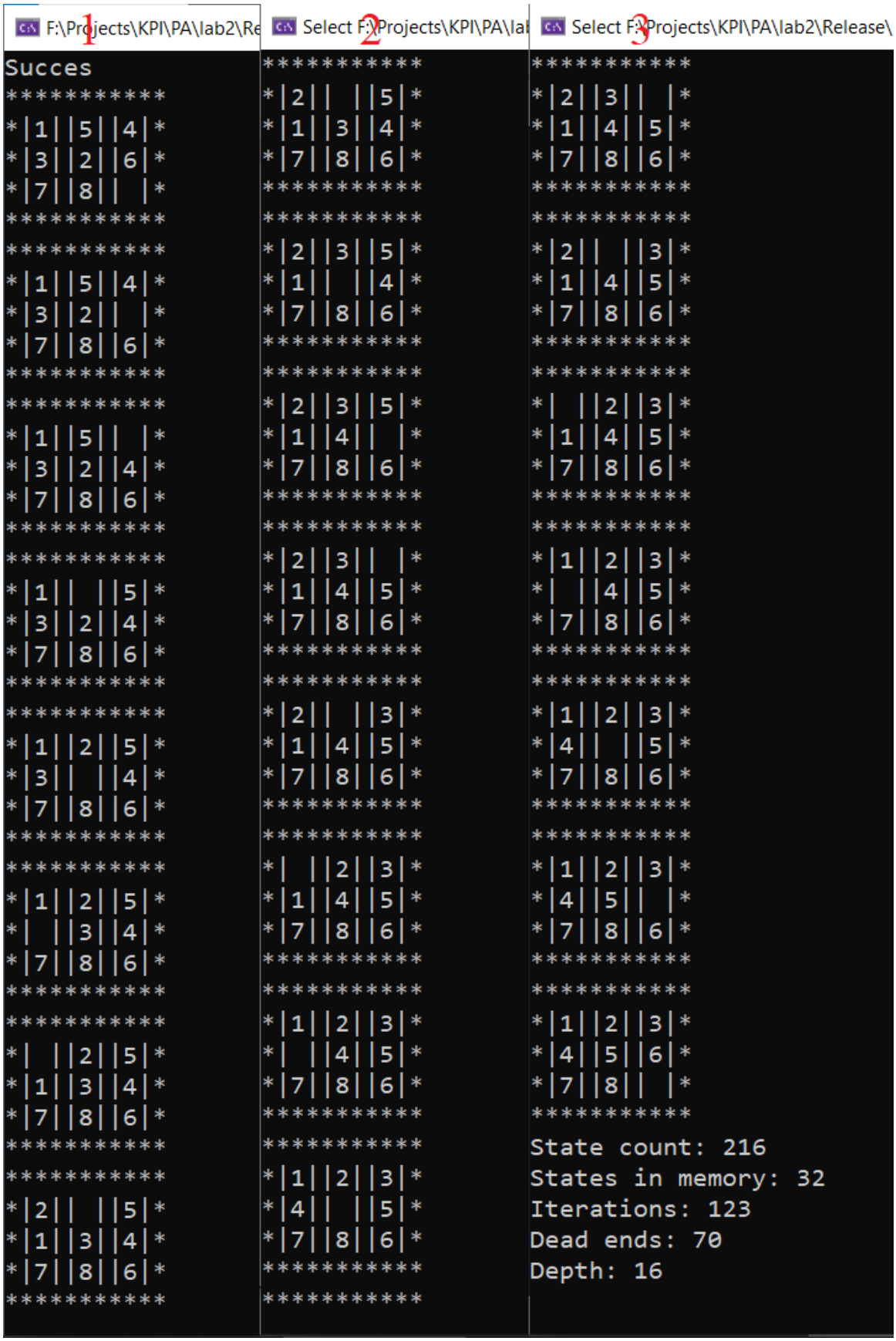

Рисунок 3.2 – Алгоритм RBFS

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму LBFS, задачі 8-puzzle для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму LBFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
|3||5||4| |2||7||8| | ||1||6|	57236288	37561305	57236288	23
|7||5||4| |2|| ||1| |8||3||6|	4259607397	2795367348	4259607397	23
|4||2||5| | ||6||1| |3||7||8|	9817068105	6442450944	9817068105	23
|5||2||3| |4||1||7| |6||8|| |	259742862	170456244	259742862	23
|8||7||2| | ||6||3| |4||1||5|	4159965845	2647250984	4159965845	23
|4||3||6| |2||7||8| |5|| ||1|	52031472	33110928	52031472	23
|5||6||2| |3||1||4| | ||7||8|	9817068105	6442450944	9817068105	23
|4||3||7| |1||2||5| |8||6|| |	1042387664	684066897	1042387664	23
|4||8||5| | ||2||7| |1||3||6|	9817068105	6442450944	9817068105	23

|7|| ||3| |2||5||8| |1||4||6|	3324984569	2115899264	3324984569	23
|7|| ||3| |1||4||5| |6||8||2|	339638705	216133712	339638705	23
|1||7||2| |8||4||6| |3||5|| |	251327045	164933364	251327045	23
|5||6||1| |4||2||3| |7|| ||8|	12183840	7753344	12183840	23
|8||1||5| |7||4||3| | ||6||2|	2075467228	1362025359	2075467228	23
|4|| ||2| |7||6||3| |5||8||1|	1779320890	1132295104	1779320890	23
| ||6||7| |1||3||5| |2||8||4|	9817068105	6442450944	9817068105	23
|5||4||6| |8||1||3| | ||7||2|	9235930528	6061079400	9235930528	23
|7||6||5| |3|| ||4| |2||1||8|	9672977785	6347891664	9672977785	23
|5||6||4| |2||8||3| | ||1||7|	9817068105	6442450944	9817068105	23

В таблиці 3.2 наведені характеристики оцінювання алгоритму RBFS задачі 8-puzzle для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання RBFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
|3||5||4| |2||7||8| | ||1||6|	84	38	141	35
|7||5||4| |2|| ||1| |8||3||6|	596	358	1025	38
|4||2||5| | ||6||1| |3||7||8|	3897	2492	6784	45
|5||2||3| |4||1||7| |6||8|| |	1052	650	1809	38
|8||7||2| | ||6||3| |4||1||5|	2134	1333	3679	44
|4||3||6| |2||7||8| |5|| ||1|	194	110	335	34
|5||6||2| |3||1||4| | ||7||8|	8735	5564	15048	48
|4||3||7| |1||2||5| |8||6|| |	933	578	1606	38
|4||8||5| | ||2||7| |1||3||6|	7261	4526	12421	50

|7|| ||3| |2||5||8| |1||4||6|	2036	1304	3545	40
|7|| ||3| |1||4||5| |6||8||2|	870	532	1483	42
|1||7||2| |8||4||6| |3||5|| |	1082	666	1855	38
|5||6||1| |4||2||3| |7|| ||8|	133	74	234	31
|8||1||5| |7||4||3| | ||6||2|	1089	703	1911	40
|4|| ||2| |7||6||3| |5||8||1|	990	608	1691	37
| ||6||7| |1||3||5| |2||8||4|	8749	5533	15006	49
|5||4||6| |8||1||3| | ||7||2|	6546	4116	11240	43
|7||6||5| |3|| ||4| |2||1||8|	621	374	1054	41
|5||6||4| |2||8||3| | ||1||7|	12229	7655	20915	48

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто два алгоритму пошуку. Неінформативний (LDFS) та інформативний (RBFS). Мною була написана реалізація цих алгоритмів та проведено серію тестувань. На основі отриманих даних можна сказати, що інформативний пошук краще на декілька порядків.

LDFS завжди використовує однакову кількість пам’яті, яка залежить від вибраної границі. Так, обравши границю занадто великою, алгоритм здійснював би зайві ітерації, а обравши границю меншою, ніж глибина, на якій є рішення, рішення ніколи не буде знайдено, що було продемонстровано у випробуваннях. Для значення, що обмежує глибину за 22, алгоритм має здійснити 9817068105 ітерацій щоб взнати, що рішення не можна досягти.

В свою чергу RBFS завжди знаходить оптимальне рішення (якщо воно ж) за відносно невелику кількість ітерацій. Евристична функція, що в ньому була використана – це сума манхетенської відстані та глибини рекурсії.

Слід зазначити, що якщо б ми не перевіряли, чи може задача бути вирішеною, то LBFS відпрацював би сталу кількість ітерацій, а RBFS працював би нескінченно.

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%.