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-22 Присяжний А. О.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе І. Е.* _____
(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: $T = 1000 - k \cdot t$, 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	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

LDFS

Depth-Limited-Search(problem, limit) {

    return Recursive-DLS(Make-Node(problem), limit);

}

Recursive-DLS(node, limit) {

    cutoff_occurred = false;

    if (node.is_solution())

        return node;

    if (node.depth == limit)

        return індикатор невдачі cutoff;

    for each спадкоємець *successor* in node.expand() **do**

        result = Recursive-DLS(successor, limit);

        if (result == cutoff)

            cutoff_occured = true;

        else if (result != failure)

            return рішення result;

    if (cutoff_occurred)

        return індикатор невдачі cutoff;

    return індикатор невдачі failure;

}

A*

A_Star(problem) {

    priority_queue **open**, елементи розташовані в порядку зростання евристичної функції;

array **close**;
initial_node = Make-Node(problem);
додати *initial_node* в *open*;

while (*open* не пуста) **do**
current = перший елемент *open*;
if (current.is_solution())
return current;
додати *current* в *close*;
for each спадкоємець *successor* in current.expand() **do**
if (*close* не містить *successor*)
додати *successor* в *open*;
}

3.2 Програмна реалізація

3.2.1 Вихідний код

functions.h

```
#pragma once

#include <string>
#include <iostream>
using namespace std;

int input_positive_number_in_range(int low, int top);
bool is_number(const string& input);
void display(int** matrix, int size);

void copy(int **dest, int** src, int size);
```

MemoryHandler.h

```
#pragma once

class MemoryHandler {
public:
    int** allocate_memory(int amount_of_rows, int amount_of_cols);
```

```
    void delete_memory(int **from, int amount_of_rows);
};
```

PuzzleCreator.h

```
#pragma once

#include <ctime>
#include <random>
#include "MemoryHandler.h"
using namespace std;


class PuzzleCreator {
    int** allocate_memory_for_puzzle(int puzzle_size);
    void fill_puzzle_with_zeros(int **puzzle, int puzzle_size);

    int generate_coords(int **puzzle, int puzzle_size);
    int generate_number_in_range(int low, int top);
public:
    int** create_puzzle(int puzzle_size);
};
```

PuzzleSolver.h

```
#pragma once

#include <queue>
#include <vector>
#include <stack>
#include "MemoryHandler.h"
#include "functions.h"

class PuzzleSolver {
    class Node {
        #define INITIAL 0
        #define UP 1
        #define DOWN 2
        #define LEFT 3
        #define RIGHT 4

        struct Coords {
            int row, col;
        };
```

```
        int** state;
        int size;
        Coords empty_cell_coords;
        int depth;
        int action;
        Node* parent;

        Coords find_coords_of_empty_cell() const;

        bool is_empty_cell_in_top_row() const;
        bool is_empty_cell_in_bottom_row() const;
        bool is_empty_cell_in_left_col() const;
        bool is_empty_cell_in_right_col() const;

        Node move_up();
        Node move_down();
        Node move_left();
        Node move_right();
        void swap(int &first_number, int &second_number);
    public:
        Node() = default;
        Node(int **puzzle, int puzzle_size, int depth, int action, Node*
parent);
        Node(Node&& obj);
        Node(const Node& obj);
        Node& operator=(Node&& obj);

        int get_amount_of_successors() const;

        Node* expand();
        bool is_solution() const;

        int** get_state();
        int get_depth() const;
        int get_size() const;
        int manhattan_distance() const;

        bool operator==(const Node& obj) const;

        void display_path();

        ~Node();
    };
```

```
        bool is_solvable(int **puzzle, int size) const;
        int* create_elements_array(int **puzzle, int size) const;
        int count_inversions(int *elements, int size) const;

        Node* _LDFS_solve(Node &node, int limit, bool &is_cutoff, bool
&is_failure);
        Node* _A_star(Node& node);

        bool contains(vector<Node*> &container, const Node& obj) const;
    public:
        int** LDFS_solve(int** puzzle, int puzzle_size, int limit);
        int** A_star(int** puzzle, int puzzle_size);
    };
```

functions.cpp

```
    #include "functions.h"


    int input_positive_number_in_range(int low, int top) {
        int number;
        string input;
        bool repeat;
        do {
            repeat = false;
            getline(cin, input);
            if (!is_number(input) || (input[0] == '0' && input.length() != 1))
{
                cout << "Invalid data, input positive integer number, please."
<< endl;
                repeat = true;
                continue;
            }

            number = stoi(input);
            if (number < low || number > top) {
                cout << "Number must be from " << low << " to " << top <<
endl;
                repeat = true;
            }
        } while (repeat);
        return number;
    }
```

```
bool is_number(const string& input) {
    for (char ch : input) {
        if (!isdigit(ch))
            return false;
    }
    return true;
}
void display(int** matrix, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j)
            cout << matrix[i][j] << " ";
        cout << endl;
    }
    cout << "--------------------------" << endl;
}

void copy(int** dest, int** src, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j)
            dest[i][j] = src[i][j];
    }
}
```

Lab2.cpp

```
#include "PuzzleCreator.h"
#include "PuzzleSolver.h"
#include <ctime>
#include <iomanip>

int main() {
    const int PUZZLE_SIZE = 3;
    const int LIMIT = 26;
    PuzzleCreator creator;
    PuzzleSolver solver;
    int run;
    clock_t beginning_of_searching, end_of_searching;

    do {
        int** puzzle = creator.create_puzzle(PUZZLE_SIZE);

        cout << "Puzzle:" << endl;
        display(puzzle, PUZZLE_SIZE);
```

```
            try {
                beginning_of_searching = clock();
                int** result = solver.A_star(puzzle, PUZZLE_SIZE);
                end_of_searching = clock();

                cout << "A* solution:" << endl;
                display(result, PUZZLE_SIZE);
                cout << "A* searching took " << fixed << double(end_of_searching - beginning_of_searching) / CLOCKS_PER_SEC << setprecision(5) << " seconds" << endl;
            }
            catch (const char* er) {
                cout << er << endl;
                end_of_searching = clock();
                cout << "A* searching took " << fixed << double(end_of_searching - beginning_of_searching) / CLOCKS_PER_SEC << setprecision(5) << " seconds" << endl;
            }

            try {
                beginning_of_searching = clock();
                int **result = solver.LDFS_solve(puzzle, PUZZLE_SIZE, LIMIT);
                end_of_searching = clock();

                cout<<"LDFS solution:"<<endl;
                display(result, PUZZLE_SIZE);
                cout << "LDFS searching took " << fixed << double(end_of_searching - beginning_of_searching) / CLOCKS_PER_SEC << setprecision(5) << " seconds" << endl;
            } catch (const char *er) {
                cout << er << endl;
                end_of_searching = clock();
                cout << "LDFS searching took " << fixed << double(end_of_searching - beginning_of_searching) / CLOCKS_PER_SEC << setprecision(5) << " seconds" << endl;
            }

            cout << "Enter 0 to stop program or 1 to run it again" << endl;
            run = input_positive_number_in_range(0, 1);
        } while (run);
        return 0;
    }
```

MemoryHandler.cpp

```
#include "MemoryHandler.h"

int **MemoryHandler::allocate_memory(int amount_of_rows, int amount_of_cols) {
    int **matrix = new int* [amount_of_rows];
    for(int i = 0; i < amount_of_rows; ++i)
        matrix[i] = new int[amount_of_cols];
    return matrix;
}

void MemoryHandler::delete_memory(int **from, int amount_of_rows) {
    for(int i = 0; i < amount_of_rows; ++i)
        delete[] from[i];
    delete[] from;
}
```

PuzzleCreator.cpp

```
#include "PuzzleCreator.h"

int **PuzzleCreator::create_puzzle(int puzzle_size) {
    srand(time(nullptr));

    int **puzzle = allocate_memory_for_puzzle(puzzle_size);
    fill_puzzle_with_zeros(puzzle, puzzle_size);

    for(int i = 1; i < puzzle_size*puzzle_size; ++i) {
        int coord = generate_coords(puzzle, puzzle_size);
        puzzle[coord/3][coord%3] = i;
    }

    return puzzle;
}
int** PuzzleCreator::allocate_memory_for_puzzle(int puzzle_size) {
    MemoryHandler handler;
    return handler.allocate_memory(puzzle_size, puzzle_size);
}
void PuzzleCreator::fill_puzzle_with_zeros(int **puzzle, int puzzle_size) {
    for(int i = 0; i < puzzle_size; ++i) {
        for(int j = 0; j < puzzle_size; ++j)
            puzzle[i][j] = 0;
    }
```

```
}
int PuzzleCreator::generate_coords(int **puzzle, int puzzle_size) {
    int coord;

    do {
        coord = generate_number_in_range(0, puzzle_size*puzzle_size-1);
    } while(puzzle[coord/3][coord%3] != 0);

    return coord;
}
int PuzzleCreator::generate_number_in_range(int low, int top) {
    return low + rand() % (top-low+1);
}
```

PuzzleSolver.cpp

```
#include "PuzzleSolver.h"

PuzzleSolver::Node::Node(int **puzzle, int puzzle_size, int depth, int action, Node* parent) : depth(depth), action(action), parent(parent) {
    size = puzzle_size;

    MemoryHandler handler;
    state = handler.allocate_memory(size, size);

    copy(state, puzzle, size);

    empty_cell_coords = find_coords_of_empty_cell();
}
PuzzleSolver::Node::Node(const PuzzleSolver::Node &obj) {
    this->size = obj.size;
    this->empty_cell_coords = obj.empty_cell_coords;
    this->depth = obj.depth;
    this->action = obj.action;
    this->parent = obj.parent;

    MemoryHandler handler;
    state = handler.allocate_memory(size, size);

    copy(state, obj.state, size);
}

PuzzleSolver::Node::Coords PuzzleSolver::Node::find_coords_of_empty_cell() const {
```

```
        Coords empty_cell_coords;

        for(int i = 0; i < size; ++i) {
            for(int j = 0; j < size; ++j) {
                if(state[i][j] == 0) {
                    empty_cell_coords.row = i;
                    empty_cell_coords.col = j;
                    return empty_cell_coords;
                }
            }
        }

        empty_cell_coords.row = -1;
        empty_cell_coords.col = -1;
        return empty_cell_coords;
    }
    PuzzleSolver::Node::Node(PuzzleSolver::Node &&obj) {
        this->state = obj.state;
        obj.state = nullptr;

        this->size = obj.size;
        this->empty_cell_coords = obj.empty_cell_coords;
        this->depth = obj.depth;
        this->action = obj.action;
        this->parent = obj.parent;
    }
    PuzzleSolver::Node&       PuzzleSolver::Node::operator=(PuzzleSolver::Node
&&obj) {
        this->state = obj.state;
        obj.state = nullptr;

        this->size = obj.size;
        this->empty_cell_coords = obj.empty_cell_coords;
        this->depth = obj.depth;
        this->action = obj.action;
        this->parent = obj.parent;

        return *this;
    }


    PuzzleSolver::Node* PuzzleSolver::Node::expand() {
        int amount_of_successors = get_amount_of_successors();
```

```
    Node* successors = new Node[amount_of_successors];
    int i = 0;

    if(!is_empty_cell_in_top_row() && action!=UP)
        successors[i++] = move_down();
    if(!is_empty_cell_in_bottom_row() && action!=DOWN)
        successors[i++] = move_up();
    if(!is_empty_cell_in_left_col() && action!=RIGHT)
        successors[i++] = move_right();
    if(!is_empty_cell_in_right_col() && action!=LEFT)
        successors[i] = move_left();

    return successors;
}
int PuzzleSolver::Node::get_amount_of_successors() const {
    int amount_of_successors = 4;

    if(is_empty_cell_in_top_row() || action==UP)
        --amount_of_successors;
    if(is_empty_cell_in_bottom_row() || action==DOWN)
        --amount_of_successors;
    if(is_empty_cell_in_left_col() || action==RIGHT)
        --amount_of_successors;
    if(is_empty_cell_in_right_col() || action==LEFT)
        --amount_of_successors;

    return amount_of_successors;
}
bool PuzzleSolver::Node::is_empty_cell_in_top_row() const {
    return empty_cell_coords.row == 0;
}
bool PuzzleSolver::Node::is_empty_cell_in_bottom_row() const {
    return empty_cell_coords.row == size-1;
}
bool PuzzleSolver::Node::is_empty_cell_in_left_col() const {
    return empty_cell_coords.col == 0;
}
bool PuzzleSolver::Node::is_empty_cell_in_right_col() const {
    return empty_cell_coords.col == size-1;
}

PuzzleSolver::Node PuzzleSolver::Node::move_up() {
    Node new_node(state, size, depth+1, UP, this);
```

```
    swap(new_node.state[empty_cell_coords.row][empty_cell_coords.col],
         new_node.state[empty_cell_coords.row+1][empty_cell_coords.col]);
    ++new_node.empty_cell_coords.row;

    return new_node;
}
PuzzleSolver::Node PuzzleSolver::Node::move_down() {
    Node new_node(state, size, depth+1, DOWN, this);

    swap(new_node.state[empty_cell_coords.row][empty_cell_coords.col],
         new_node.state[empty_cell_coords.row-1][empty_cell_coords.col]);
    --new_node.empty_cell_coords.row;

    return new_node;
}
PuzzleSolver::Node PuzzleSolver::Node::move_left() {
    Node new_node(state, size,  depth+1, LEFT, this);

    swap(new_node.state[empty_cell_coords.row][empty_cell_coords.col],
         new_node.state[empty_cell_coords.row][empty_cell_coords.col+1]);
    ++new_node.empty_cell_coords.col;

    return new_node;
}
PuzzleSolver::Node PuzzleSolver::Node::move_right() {
    Node new_node(state, size,  depth+1, RIGHT, this);

    swap(new_node.state[empty_cell_coords.row][empty_cell_coords.col],
         new_node.state[empty_cell_coords.row][empty_cell_coords.col-1]);
    --new_node.empty_cell_coords.col;

    return new_node;
}
void PuzzleSolver::Node::swap(int &first_number, int &second_number) {
    int temp = first_number;
    first_number = second_number;
    second_number = temp;
}

bool PuzzleSolver::Node::is_solution() const {
    for(int i = 0; i < size; ++i) {
        for(int j = 0; j < size; ++j) {
```

```
                if(state[i][j] != i * size + j)
                    return false;
            }
        }
        return true;
    }

    int **PuzzleSolver::Node::get_state() {
        return state;
    }
    int PuzzleSolver::Node::get_depth() const {
        return depth;
    }
    int PuzzleSolver::Node::get_size() const {
        return size;
    }
    int PuzzleSolver::Node::manhattan_distance() const {
        int distance = 0;

        for(int i = 0; i < size; ++i) {
            for(int j = 0; j < size; ++j)
                distance += abs(state[i][j]/size - i) + abs(state[i][j]%size -
j);
        }

        return distance;
    }

    bool PuzzleSolver::Node::operator==(const PuzzleSolver::Node &obj) const {
        for(int i = 0; i < size; ++i) {
            for(int j = 0; j < size; ++j) {
                if(state[i][j] != obj.state[i][j])
                    return false;
            }
        }
        return true;
    }

    void PuzzleSolver::Node::display_path() {
        stack<Node*> path;

        Node* current = this;
        path.push(current);
```

```
            while(current->parent != nullptr) {
                current = current->parent;
                path.push(current);

            }

            while (!path.empty()) {
                current = path.top();
                path.pop();
                display(current->get_state(), current->get_size());
            }
        }
        PuzzleSolver::Node::~Node() {
            if(state) {
                MemoryHandler handler;
                handler.delete_memory(state, size);
            }
        }


        int **PuzzleSolver::LDFS_solve(int **puzzle, int puzzle_size, int limit) {
            if(!is_solvable(puzzle, puzzle_size))
                throw "Puzzle doesn't have solution";

            Node initial_state(puzzle, puzzle_size, 1, INITIAL, nullptr);

            bool is_cutoff = false;
            bool is_failure = false;
            Node   *result   =   _LDFS_solve(initial_state,   limit,   is_cutoff,
is_failure);
            if (result) {
                result->display_path();

                int** result_state;
                MemoryHandler handler;
                result_state = handler.allocate_memory(result->get_size(), result-
>get_size());
                copy(result_state, result->get_state(), result->get_size());

                return result_state;
            }

            if(is_failure)
```

```
            throw "Failure";
        if(is_cutoff)
            throw "Cutoff";
    }

    PuzzleSolver::Node* PuzzleSolver::_LDFS_solve(Node &node, int limit, bool
&is_cutoff, bool &is_failure) {
        bool cutoff_occurred = false;
        is_cutoff = false;
        is_failure = false;
        if (node.is_solution())
            return &node;


        if(node.get_depth() >= limit) {
            is_cutoff = true;
            return nullptr;
        }

        Node *successors = node.expand();
        int amount_of_successors = node.get_amount_of_successors();
        Node *result;

        for(int i = 0; i < amount_of_successors; ++i) {
            result = _LDFS_solve(successors[i], limit, is_cutoff, is_failure);

            if(is_cutoff)
                cutoff_occurred = true;
            else if(!is_failure)
                return result;
        }
        delete[] successors;

        if(cutoff_occurred) {
            is_cutoff = true;
            return nullptr;
        }

        is_failure = true;
        return nullptr;
    }

    bool PuzzleSolver::is_solvable(int **puzzle, int size) const {
```

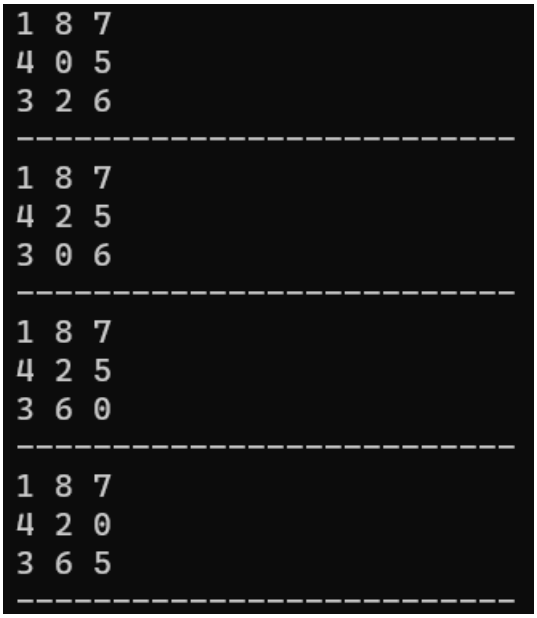
```
    int *elements = create_elements_array(puzzle, size);
    int inversions = count_inversions(elements, size*size-1);

    delete[] elements;
    return (inversions % 2 == 0);
}
int *PuzzleSolver::create_elements_array(int **puzzle, int size) const {
    int *elements = new int[size*size-1];
    int index = 0;

    for(int i = 0; i < size; ++i) {
        for(int j = 0; j < size; ++j) {
            if(puzzle[i][j] != 0)
                elements[index++] = puzzle[i][j];
        }
    }

    return elements;
}
int PuzzleSolver::count_inversions(int *elements, int size) const {
    int inversions = 0;

    for(int i = 0; i < size; ++i) {
        for(int j = i+1; j < size; ++j) {
            if(elements[i] > elements[j])
                ++inversions;
        }
    }

    return inversions;
}
int** PuzzleSolver::A_star(int** puzzle, int puzzle_size) {
    if (!is_solvable(puzzle, puzzle_size))
        throw "Puzzle doesn't have solution";

    Node initial_state(puzzle, puzzle_size, 1, INITIAL, nullptr);

    Node *result = _A_star(initial_state);
    result->display_path();
    int** result_state;
    MemoryHandler handler;
    result_state = handler.allocate_memory(puzzle_size, puzzle_size);
    copy(result_state, result->get_state(), puzzle_size);
```

```
        return result_state;
    }
    PuzzleSolver::Node *PuzzleSolver::_A_star(Node &node) {
        class NodeComparator {
        public:
            bool operator()(Node *&obj1, Node *&obj2) {
                int heur1 = obj1->get_depth() + obj1->manhattan_distance() -
1;
                int heur2 = obj2->get_depth() + obj2->manhattan_distance() -
1;
                return heur1 > heur2;
            }
        };

        priority_queue<Node*, vector<Node*>, NodeComparator> open;
        vector<Node*> close;

        open.push(&node);

        while (!open.empty()) {
            Node* current = open.top();
            open.pop();

            if (current->is_solution())
                return current;

            close.push_back(current);
            PuzzleSolver::Node* successors = current->expand();
            int amount_of_successors = current->get_amount_of_successors();

            for(int i = 0; i < amount_of_successors; ++i) {
                if(!contains(close, successors[i]))
                    open.push(&successors[i]);
            }
        }
    }

    bool PuzzleSolver::contains(vector<PuzzleSolver::Node*> &container, const
PuzzleSolver::Node &obj) const {
        for (Node*& curr : container) {
            if(*curr == obj)
                return true;
        }
```

```
        return false;
    }
```

3.2.2 Приклади роботи

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



…

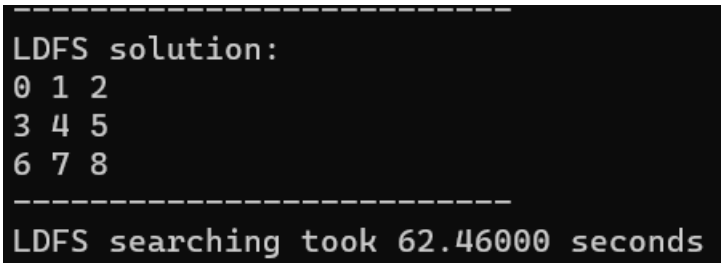
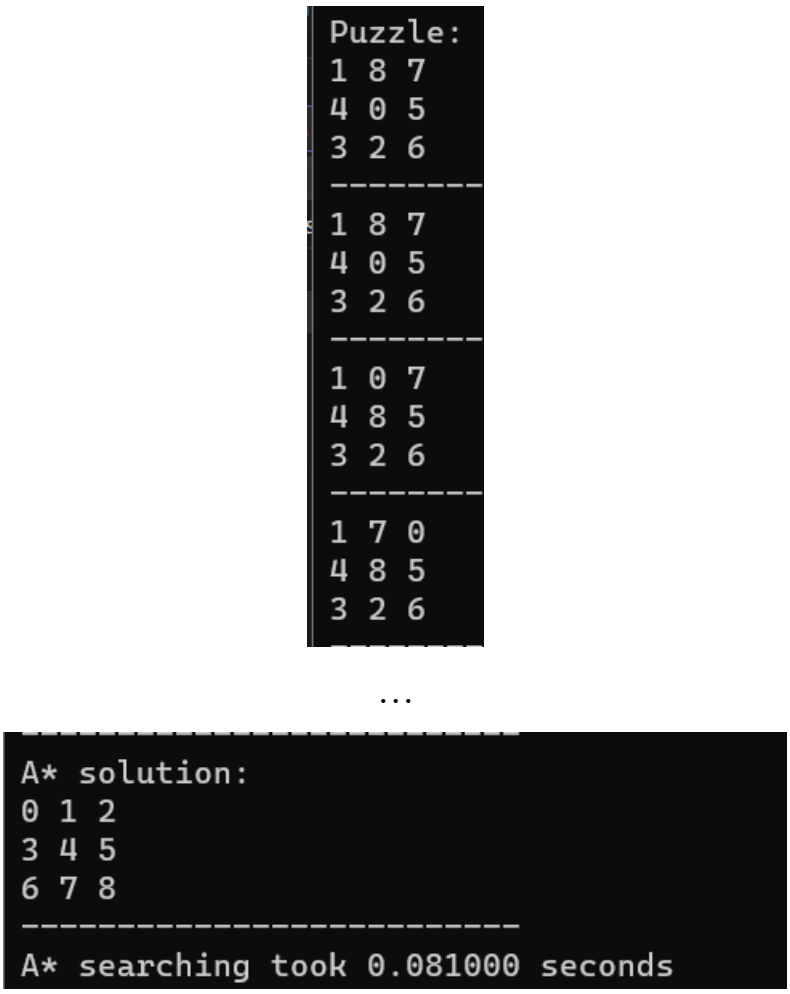


Рисунок 3.1 – Алгоритм LDFS



…

Рисунок 3.2 – Алгоритм A*

3.3 Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання LDFS, limit = 26

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом’яті
8 7 2 5 6 4 0 1 3	145236118	1	305136730	104
8 0 2 3 6 7 5 4 1	106298189	0	230790414	104
8 0 2 5 4 7 1 3 6	15061702	0	32701417	104
0 4 6 3 1 5 7 8 2	145236118	1	305136730	104
7 3 4 2 0 6 1 8 5	230733476	0	484764396	100
0 7 3 1 2 8 6 5 4	99816581	0	209711723	92
3 6 8 1 0 4 7 2 5	290471819	1	610273044	104
7 5 3 6 8 0 2 4 1	108504657	0	235582407	104
3 1 4 6 8 2	115824	0	243357	100

0 5 7				
4 1 2 3 6 8 0 5 7	1759900	0	3697477	100
3 6 7 4 8 1 5 0 2	52783168	0	114600664	104
0 4 1 3 8 7 6 5 2	99440753	0	208922113	100
0 2 8 4 6 3 1 7 5	145236118	1	305136730	104
8 3 4 1 2 5 7 0 6	238243669	1	517265700	104
1 0 8 4 5 6 2 3 7	238243669	1	517265700	104
4 6 2 1 0 5 8 7 3	8512939	0	17885318	92
7 6 2 0 8 1 5 3 4	171350193	1	372029999	104
5 8 0 2 1 3 4 7 6	145236118	1	305136730	104
3 2 4	145236118	1	305136730	104

1 8 5 7 6 0				
4 3 0 2 5 7 6 1 8	3153778	0	6625932	100

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

Таблиця 3.2 – Характеристики оцінювання A*

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом’яті
5 6 4 0 8 2 3 7 1	1203	0	2439	902
1 0 6 4 5 2 3 7 8	147	0	295	114
3 7 8 0 2 6 5 1 4	991	0	1976	734
8 5 3 2 1 7 4 0 6	3810	0	7660	2781
0 7 2 8 1 3 6 4 5	916	0	1846	676
8 3 7 5 2 0 1 4 6	4940	0	9911	3617
6 1 4 0 3 5 2 8 7	3770	0	7634	2767
7 2 3 4 1 5 8 6 0	2944	0	5917	2159
0 3 4 7 8 2	8746	0	17600	6338

6 5 1				
2 8 3 4 1 7 0 6 5	3070	0	6148	2255
8 0 1 3 6 4 7 2 5	233	0	477	179
8 4 3 5 1 7 2 0 6	11572	0	23234	8339
1 8 3 0 4 7 5 2 6	12597	0	25246	9050
7 5 6 0 8 4 1 3 2	505	0	1016	376
7 8 5 4 3 0 1 6 2	635	0	1272	473
1 5 8 2 4 7 6 0 3	206	0	422	158
5 0 1 3 7 4 6 2 8	3371	0	6813	2452
8 3 4 5 1 2 7 6 0	6308	0	12706	4596
2 0 3	3137	0	6335	2310

6 7 4 1 8 5				
3 1 4 0 7 2 5 6 8	1205	0	2449	896

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто задачу 8-puzzle та способи її вирішення за допомогою алгоритму неінформативного пошуку (обмежений пошук вглибину) та алгоритму інформативного пошуку з евристичною функцією Манхетенська відстань (A*). Я розробив програму на мові c++, яка реалізує ці два алгоритми. За результатами роботи програми я склав порівняльні характеристики алгоритмів (таблиці 3.1 та 3.2), а також навів скріншоти розв’язання головоломки. За даними в таблиці легко можна зрозуміти, що алгоритм A* працює набагато ефективніше за LDFS, про що свідчать і відповідні скріншоти. Так, A* справився з головоломкою за 0.08 секунд, а LDFS – за 62.46 секунд, що у 780 разів повільніше.

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%.