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 Медвідь Олександр* _____
(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: $T = 1000 - k \cdot t$, 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

BFS:

```
while Flag:
  temp = Plan.front()
  Plan.pop()
  ChessBoardArr = temp->GetData()->BoardArrGen()
  for i = 0 i < ChessBoardArr.size() && Flag i++:
    temp2 = temp->Insert(ChessBoardArr[i], i)
    Plan.push(temp2)
    if (!temp2->GetData()->ConflictCheck()):
      Flag = false
      Counter++
    if (MaxCount < Plan.size() + 1):
      MaxCount = Plan.size() + 1
    if (temp2 != temp):
      delete temp
```

A*:

```
while Flag:
 Iter++
 State Current = Nodes.top()
 Nodes.pop()
 std::vector<ChessBoard*> Descendants = Current.Data->BoardArrGen()
 for i = 0 i < Descendants.size() && Flag i++:
 if (Descendants[i] != nullptr):
  if (!Descendants[i]->ConflictCheck())
   Flag = false
   SetBoard(Descendants[i])
  else
   Nodes.push(State(Descendants[i], Current.Depth + 1))
   Counter++
 delete Current.Data;
 if (MaxCount < Nodes.size()):
  MaxCount = Nodes.size()
```

3.2 Програмна реалізація

3.2.1 Вихідний код

Файл Main

```
#include <iostream>
#include <ctime>
#include "ChessBoard.h"
using namespace std;

int main()
{
	ChessBoard chessboard;
	cout << "Starting board:\n";
	chessboard.Generation();
	chessboard.Output();
	int Choice;
	cout << "\nEnter 1 to complete the task with the BFS algorithm\nEnter 2 to complete
the task with the A* algorithm";
	for (int i = 0;; i++)
	{
		cout << "\nEnter the number ";
		cin >> Choice;

		if (Choice != 1 && Choice != 2)
		{
			cout << "\nThe input is incorrect, try again";
		}
		else
		{
			break;
		}
	}
	unsigned int StartTime;
	if (Choice == 1)
	{
		StartTime = clock();
		chessboard.BFS();
	}
	else
	{
		StartTime = clock();
		chessboard.As();
	}
	unsigned int EndTime = clock();
	int S, Ms;
	S = (EndTime - StartTime) / 1000;
	Ms = EndTime - StartTime - S * 1000;
	cout << endl << "Result:";
	cout << endl << "Time spent: " << S << " s " << Ms << " ms" << endl << endl;

	chessboard.Output();
	}
```

Файл ChessBoard.h

```
#pragma once
#include <iostream>
#include <ctime>
#include <queue>
#include <vector>
```

```
#include "windows.h"
#include "psapi.h"
using namespace std;

class ChessBoard
{
private:
	//Кількість ферзів на шаховій дошці
	int Quantity;
	//Представлення шахової дошки у виді двомірного масиву
	int Board[8][8];

public:
	//Конструктор класу
	ChessBoard();
	//Конструктор копіювання
	ChessBoard(ChessBoard* Previous);
	//Конструктор з перестановкою стовпчиків або колонок попередньої дошки
	ChessBoard(ChessBoard* Previous, int Position1, int Position2 = -1, int Orientation =
1);
	//Функція розставлення ферзів на дошці випадковим чином
	void Generation();
	//Сетер дошки
	void SetBoard(ChessBoard* Previous);
	//Функція виводу шахової дошки
	void Output();
	int F1();

	bool ConflictCheck();
	bool ConflictCheckStraight(int x, int y);
	bool ConflictCheckDiagonal(int x, int y);
	int QueenHitNumber();
	std::vector<ChessBoard*> BoardArrGen();

	//Пошук вшир
	void BFS();
	//A*
	void As ();
	};
```

Файл ChessBoard.cpp

```
#include "ChessBoard.h"
#include "Tree.h"
#include "State.h"
using namespace std;

//Конструктор класу
ChessBoard::ChessBoard()
{
	Quantity = 8;
	for (int i = 0; i < Quantity; i++)
	{
		for (int j = 0; j < Quantity; j++)
		{
			Board[i][j] = 0;
		}
	}
}


ChessBoard::ChessBoard(ChessBoard* Previous)
{
	Quantity = Previous->Quantity;
	SetBoard(Previous);
```

```
}

ChessBoard::ChessBoard(ChessBoard* Previous, int Position1, int Position2, int Orientation)
{
	SetBoard(Previous);
	Position1 %= Quantity;

	if (Position2 == -1)
	{
		Position2 = (Position1 + 1) % Quantity;
	}
	else
	{
		Position2 %= Quantity;
	}

	int temp;
	switch (Orientation)
	{
	case 1: //Стовбці
		for (int i = 0; i < Quantity; i++)
		{
			temp = Board[i][Position1];
			Board[i][Position1] = Board[i][Position2];
			Board[i][Position2] = temp;
		}
		break;
	case 2: //Рядки
		for (int i = 0; i < Quantity; i++)
		{
			temp = Board[Position1][i];
			Board[Position1][i] = Board[Position2][i];
			Board[Position2][i] = temp;
		}
	}
}

void ChessBoard::SetBoard(ChessBoard* Previous)
{
	Quantity = Previous->Quantity;
	for (int i = 0; i < Quantity; i++)
	{
		for (int j = 0; j < Quantity; j++)
		{
			Board[i][j] = Previous->Board[i][j];
		}
	}
}

//Функція розставлення ферзів на дошці випадковим чином
void ChessBoard::Generation()
{
	srand(time(NULL));
	int RandNum1, RandNum2, Temp;
	int Arr[8];

	for (int i = 0; i < Quantity; i++)
	{
		Arr[i] = i;
	}

	for (int i = 0; i < Quantity; i++)
	{
		RandNum1 = rand() % Quantity;
		RandNum2 = rand() % Quantity;
```

```
			Temp = Arr[RandNum1];
			Arr[RandNum1] = Arr[RandNum2];
			Arr[RandNum2] = Temp;
		}

		for (int i = 0; i < Quantity; i++)
		{
			Board[i][Arr[i]] = 1;
		}
}

//Функція виводу шахової дошки
void ChessBoard::Output()
{
		int arr[8];
		for (int i = 0; i < Quantity; i++)
		{
			for (int j = 0; j < Quantity; j++)
			{
				cout << Board[i][j] << " ";
				if (Board[i][j] == 1)
				{
					arr[i] = j;
				}
			}
			cout << endl;
		}
		cout << endl << "The State:" << endl;
		for (int i = 0; i < 8; i++)
		{
			cout << arr[i] << " ";
		}
}

int ChessBoard::F1()
{
		int* Location[8];
		int QuantDiagonal = 0;
		for (int i = 0; i < 8; i++)
		{
			for (int j = 0; j < 8; j++)
			{
				if (Board[i][j] == 1)
				{
					int tempx = i, tempy = j;
					while (tempx > 0 && tempy > 0)
					{
						tempx--;
						tempy--;
						if (Board[tempx][tempy] != 0)
						{
							QuantDiagonal++;
							break;
						}
					}

					tempx = i, tempy = j;
					while (tempx < Quantity - 1 && tempy < Quantity - 1)
					{
						tempx++;
						tempy++;
						if (Board[tempx][tempy] != 0)
						{
							QuantDiagonal++;
```

```
                                        break;
                                }
                        }

                        tempx = i, tempy = j;
                        while (tempx > 0 && tempy < Quantity - 1)
                        {
                                tempx--;
                                tempy++;
                                if (Board[tempx][tempy] != 0)
                                {
                                        QuantDiagonal++;
                                        break;
                                }
                        }

                        tempx = i, tempy = j;
                        while (tempx < Quantity - 1 && tempy > 0)
                        {
                                tempx++;
                                tempy--;
                                if (Board[tempx][tempy] != 0)
                                {
                                        QuantDiagonal++;
                                        break;
                                }
                        }
                }
        }
}

        return QuantDiagonal / 2;
}

bool ChessBoard::ConflictCheck()
{
        bool Flag = false;
        for (int i = 0; i < Quantity && !Flag; i++)
        {
                for (int j = 0; j < Quantity && !Flag; j++)
                {
                        if (Board[i][j] != 0 && !Flag)
                        {
                                Flag = ConflictCheckDiagonal(i, j);
                        }
                }
        }

        return Flag;
}

bool ChessBoard::ConflictCheckStraight(int x, int y)
{
        for (int i = 1; i < Quantity; i++)
        {
                if (Board[(x + i) % Quantity][y] != 0)
                {
                        return true;
                }
                if (Board[x][(y + i) % Quantity] != 0)
                {
                        return true;
                }
        }
```

```
	return false;
}

bool ChessBoard::ConflictCheckDiagonal(int x, int y)
{
	int tempx = x, tempy = y;
	while (tempx > 0 && tempy > 0)
	{
		tempx--;
		tempy--;
		if (Board[tempx][tempy] != 0)
		{
			return true;
		}
	}

	tempx = x, tempy = y;
	while (tempx < Quantity - 1 && tempy < Quantity - 1)
	{
		tempx++;
		tempy++;
		if (Board[tempx][tempy] != 0)
		{
			return true;
		}
	}

	tempx = x, tempy = y;
	while (tempx > 0 && tempy < Quantity - 1)
	{
		tempx--;
		tempy++;
		if (Board[tempx][tempy] != 0)
		{
			return true;
		}
	}

	tempx = x, tempy = y;
	while (tempx < Quantity - 1 && tempy > 0)
	{
		tempx++;
		tempy--;
		if (Board[tempx][tempy] != 0)
		{
			return true;
		}
	}

	return false;
}

int ChessBoard::QueenHitNumber()
{
	int count = 0;
	for (int i = 0; i < Quantity; i++)
	{
		for (int j = 0; j < Quantity; j++)
		{
			if (Board[i][j] != 0)
			{
				if (ConflictCheckDiagonal(i, j))
				{
					count++;
				}
```

```
                        }
                }
        }
        return count;
}

std::vector<ChessBoard*> ChessBoard::BoardArrGen()
{
        int QueenHit = Quantity;
        std::vector<ChessBoard*> result;

        for (int orientation = 1; orientation <= 2; orientation++)
        {
                for (int j = 0; j < Quantity - 1; j++)
                {
                        for (int k = j + 1; k < Quantity; k++)
                        {
                                result.push_back(new ChessBoard(this, j, k, orientation));
                        }
                }
        }
        return result;
}


void ChessBoard::BFS()
{
        if (!this->ConflictCheck())
        {
                return;
        }

        int Counter = 1; //Кількість станів
        int MaxCount = 1; //Кількість станів в пам'яті
        int Iter = 0; //Кількість ітерацій
        // змінні для BFS алгоритму
        queue <Tree*> Plan;
        Tree* temp = new Tree(new ChessBoard(this), pow(Quantity, 2) - Quantity);
        Plan.push(temp);
        bool Flag = temp->GetData()->ConflictCheck();
        Tree* temp2 = temp;
        std::vector<ChessBoard*> ChessBoardArr;

        while (Flag)
        {
                Iter++;
                //BFS алгоритм
                temp = Plan.front();
                Plan.pop();
                ChessBoardArr = temp->GetData()->BoardArrGen();

                for (int i = 0; i < ChessBoardArr.size() && Flag; i++)
                {
                        temp2 = temp->Insert(ChessBoardArr[i], i);
                        Plan.push(temp2);
                        if (!temp2->GetData()->ConflictCheck())
                        {
                                Flag = false;
                        }
                        Counter++;
                }

                // визначення найбільшої кількості вершин, що існували в один момент
                if (MaxCount < Plan.size() + 1)
                {
```

```
                        MaxCount = Plan.size() + 1;
                }

                // видалення непотрібних вершин
                if (temp2 != temp)
                {
                        delete temp;
                }
        }

        // видалення всіх вершин дерева, що залишилися
        while (Plan.size() > 1)
        {
                temp = Plan.front();
                Plan.pop();
                delete temp;
        }

        // результат
        for (int i = 0; i < Quantity; i++)
        {
                for (int j = 0; j < Quantity; j++)
                {
                        Board[i][j] = temp2->GetData()->Board[i][j];
                }
        }

        cout << endl << "Quantity of states " << Counter;
        cout << endl << "Quantity of states in the memory " << MaxCount;
        cout << endl << "Quantity of iterations " << Iter;
        delete temp2;
}

void ChessBoard::As()
{
        if (!this->ConflictCheck())
        {
                return;
        }

        int Counter = 1; //Кількість станів
        int MaxCount = 1; //Кількість станів в пам'яті
        int Iter = 0; //Кількість ітерацій

        auto compare = [](State a, State b)
        {
                return a.Value > b.Value;
        };

        priority_queue<State, vector<State>, decltype(compare)> Nodes(compare);
        Nodes.push(State(new ChessBoard(*this), 1));
        bool Flag = true;

        while (Flag)
        {
                Iter++;
                State Current = Nodes.top();
                Nodes.pop();
                std::vector<ChessBoard*> Descendants = Current.Data->BoardArrGen();
                for (int i = 0; i < Descendants.size() && Flag; i++)
                {
                        if (Descendants[i] != nullptr)
                        {
                                if (!Descendants[i]->ConflictCheck())
                                {
```

```
                                        Flag = false;
                                        SetBoard(Descendants[i]);
                                }
                                else
                                {
                                        Nodes.push(State(Descendants[i], Current.Depth + 1));
                                        Counter++;
                                }
                        }
                }
                delete Current.Data;
                if (MaxCount < Nodes.size())
                {
                        MaxCount = Nodes.size();
                }
        }
        while (Nodes.size())
        {
                delete Nodes.top().Data;
                Nodes.pop();
        }
        cout << endl << "Quantity of states " << Counter;
        cout << endl << "Quantity of states in the memory " << MaxCount;
        cout << endl << "Quantity of iterations " << Iter;
}
```

Файл Tree.h

```
#pragma once
#include "ChessBoard.h"
#include <iostream>

class Tree
{
private:
        //Масив покажчиків на вершини-нащадки
        Tree** Node;
        //Дані вершини - шахова дошка
        ChessBoard* Data;
        //Кількість нащадків
        int Descendants;
        //Видалення гілки дерева рекурсивною функцією
        void DeleteBranch(Tree* Node, int& Counter);

public:
        //Конструктор, покажчик на дошку та кількість нащадків
        Tree(ChessBoard*, int Num = 2);
        //Вставлення вершини-нащадка на задану позицію
        Tree* Insert(ChessBoard*, int Position);
        //Геттер покажчика на шахову дошку
        ChessBoard* GetData();
        //Геттер кількості нащадків
        int GetDescendants();
        //Геттер нащадка певної позиції
        Tree* GetDescendant(int Num);
        //Видалення усіх нащадків вершини
        int Clear();
        //Деструктор
        ~Tree();
        };
```

Файл Tree.cpp

```
#include "Tree.h"
```

```
Tree::Tree(ChessBoard* data, int Num)
{
	Data = data;
	Descendants = Num;
	Node = new Tree * [Descendants];
	for (int i = 0; i < Descendants; i++)
	{
		Node[i] = NULL;
	}
}

Tree* Tree::Insert(ChessBoard* data, int Position)
{
	Node[Position] = new Tree(data, Descendants);
	return Node[Position];
}

ChessBoard* Tree::GetData()
{
	return Data;
}

int Tree::GetDescendants()
{
	return Descendants;
}

Tree* Tree::GetDescendant(int Num)
{
	return Node[Num];
}

void Tree::DeleteBranch(Tree* Node, int& Counter)
{
	if (Node == NULL)
	{
		return;
	}

	Counter++;

	for (int i = 0; i < Descendants; i++)
	{
		DeleteBranch(Node->GetDescendant(i), Counter);
	}

	delete Node;
}

int Tree::Clear()
{
	int Counter = 0;
	for (int i = 0; i < Descendants; i++)
	{
		DeleteBranch(Node[i], Counter);
		Node[i] = NULL;
	}
	return Counter;
}

Tree::~Tree()
{
	delete Data;
	delete[Descendants] Node;
```

```
};
```

Файл State.cpp

```
#pragma once
#include "ChessBoard.h"

struct State
{
	ChessBoard* Data;
	int Depth;
	int Value;
	State(ChessBoard* Input, int Position)
	{
		Data = Input;
		Depth = Position;
		Value = Depth + Data->F1();
	}
};
```

3.2.2 Приклади роботи

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

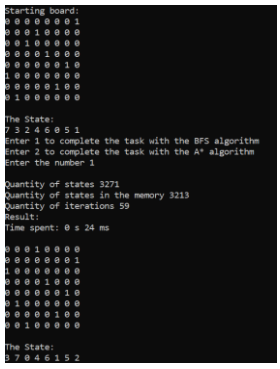

Рисунок 3.1 – Алгоритм BFS

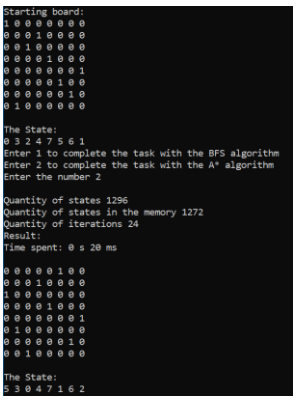

Рисунок 3.2 – Алгоритм A*

3.3 Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання BFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
5 0 2 1 7 6 3 4	802	0	44876	44075
1 6 0 3 2 4 5 7	61	0	3381	3321
4 2 5 3 7 1 6 0	3	0	133	131
0 5 2 3 6 1 4 7	3656	0	204706	201051
2 4 0 3 5 1 7 6	78	0	4336	4259
0 6 7 5 1 4 2 3	8	0	410	403
5 7 6 2 4 0 3 1	63	0	3491	3429
4 2 5 3 0 1 6 7	131	0	7301	7171
1 7 2 5 0 3 6 4	17	0	924	908
5 0 4 6 2 3 7 1	3	0	126	124
5 3 4 1 0 7 2 6	61	0	3381	3321
1 3 4 6 5 2 0 7	62	0	3431	3370
2 3 4 1 0 5 7 6	458	0	25605	25148
1 6 0 3 4 2 5 7	11	0	579	569
0 7 1 5 3 4 6 2	62	0	3431	3370
6 5 7 2 0 1 4 3	10	0	526	517
0 6 2 1 5 7 3 4	177	0	9869	9693
5 1 7 3 6 2 4 0	7	0	357	351
3 1 6 0 4 2 5 7	5	0	243	239
7 1 6 2 0 3 5 4	6	0	303	298

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

Таблиця 3.2 – Характеристики оцінювання A*

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
0 3 2 4 7 5 6 1	24	0	1296	1272
5 1 6 2 0 3 4 7	3	0	138	135
7 2 6 1 4 3 0 5	5	0	252	247
1 2 3 6 0 5 4 7	26	0	1425	1399
5 6 0 3 7 2 4 1	5	0	227	222
0 7 2 3 6 1 4 5	44	0	2410	2366
0 5 2 3 4 1 6 7	7	0	358	351
3 1 2 0 4 5 6 7	59	0	3262	3203
0 4 5 1 2 7 3 6	8	0	405	397
7 6 4 0 5 2 3 1	3	0	134	131
5 4 6 7 2 0 1 3	2	0	67	65
1 6 2 7 4 5 3 0	2	0	64	62
4 3 2 1 6 5 0 7	39	0	2146	2107
4 7 2 3 6 1 5 0	2	0	59	57
5 4 2 6 0 1 3 7	6	0	308	302
4 5 0 3 2 7 6 1	2	0	59	57
0 4 2 1 3 5 6 7	231	0	12882	12651
6 5 1 7 2 4 0 3	3	0	138	135
4 2 6 5 3 1 0 7	14	0	749	735
2 7 5 3 6 1 4 0	23	0	1245	1222

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуті алгоритми BFS та A* на прикладі задачі 8-ферзів. Були набуті навички їх використання у програмних специфікаціях. Для вирішення задачі була створена елементарна програма. Результати програми виявилися правильними, що стверджує на її дієвість. Завдання було виконано на мові програмування C++.

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%.