**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 № 1 з дисципліни

«Проектування алгоритмів»

**«Неінформативний, інформативний та локальний пошук»**

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-02 Василенко Павло Олександрович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Вєчєрковська А.С.*

Київ 2021

Зміст

[1 Мета лабораторної роботи 3](#_Toc81070688)

[2 ЗаВдання 4](#_Toc81070689)

[3 Виконання 8](#_Toc81070690)

[3.1 Псевдокод алгоритмів 8](#_Toc81070691)

[3.2 Програмна реалізація 8](#_Toc81070692)

[3.2.1 Вихідний код 8](#_Toc81070693)

[3.2.2 Приклади роботи 8](#_Toc81070694)

[3.3 Дослідження алгоритмів 8](#_Toc81070695)

[Висновок 11](#_Toc81070696)

[Критерії оцінювання 12](#_Toc81070697)

# Мета лабораторної роботи

Мета роботи – розглянути та дослідити алгоритми неінформативного, інформативного та локального пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

# ЗаВдання

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту (таблиця 2.1) за допомогою алгоритму неінформативного пошуку **АНП**, алгоритму інформативного пошуку **АІП,** що використовує задану евристичну функцію FIn, або алгоритму локального пошуку **АЛП та бектрекінгу,** що використовує задану евристичну функцію FIn.

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. 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 (512 Мб)

**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.
* **LDFS** – Пошук вглиб з обмеженням глибини.
* **BFS** – Пошук вшир.
* **IDS** – Пошук вглиб з ітеративним заглибленням.
* **A\*** – Пошук А\*.
* **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.
* **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 Т 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** | **FIn** |
| 1 | 8-ферзів | LDFS | A\* |  | F1 |
| 2 | 8-ферзів | LDFS | A\* |  | F2 |
| 3 | 8-ферзів | LDFS | RBFS |  | F1 |
| 4 | 8-ферзів | LDFS | RBFS |  | F2 |
| 5 | 8-ферзів | BFS | A\* |  | F1 |
| 6 | 8-ферзів | BFS | A\* |  | F2 |
| 7 | 8-ферзів | BFS | RBFS |  | F1 |
| 8 | 8-ферзів | BFS | RBFS |  | F2 |
| 9 | 8-ферзів | IDS | A\* |  | F1 |
| 10 | 8-ферзів | IDS | A\* |  | F2 |
| 11 | 8-ферзів | IDS | RBFS |  | F1 |
| 12 | 8-ферзів | IDS | RBFS |  | F2 |
| 13 | 8-puzzle | LDFS | A\* |  | H1 |
| 14 | 8-puzzle | LDFS | A\* |  | H2 |
| 15 | 8-puzzle | LDFS | RBFS |  | H1 |
| 16 | 8-puzzle | LDFS | RBFS |  | H2 |
| 17 | 8-puzzle | BFS | A\* |  | H1 |
| 18 | 8-puzzle | BFS | A\* |  | H2 |
| 19 | 8-puzzle | BFS | RBFS |  | H1 |
| 20 | 8-puzzle | BFS | RBFS |  | H2 |
| 21 | 8-puzzle | IDS | A\* |  | H1 |
| 22 | 8-puzzle | IDS | A\* |  | H2 |
| 23 | 8-puzzle | IDS | RBFS |  | H1 |
| 24 | 8-puzzle | IDS | RBFS |  | H2 |
| 25 | COLOR |  |  | HILL | MRV |
| 26 | COLOR |  |  | ANNEAL | MRV |
| 27 | COLOR |  |  | BEAM | MRV |
| 28 | COLOR |  |  | HILL | DGR |
| 29 | COLOR |  |  | ANNEAL | DGR |
| 30 | COLOR |  |  | BEAM | DGR |
| 31 | COLOR |  |  | HILL | MRV |
| 32 | COLOR |  |  | ANNEAL | MRV |
| 33 | COLOR |  |  | BEAM | MRV |
| 34 | COLOR |  |  | HILL | DGR |
| 35 | COLOR |  |  | ANNEAL | DGR |
| 36 | COLOR |  |  | BEAM | DGR |

# Виконання

## Псевдокод алгоритмів

1)LDFS

**Function** LDFS-Search (Tree, depthLimit)

**If**(checkStateForWin(tree.root)):

**Return** tree.root→state

**Else:**

**If**(tree.depth > depthLimit)

**Return** null

**Else:**

Children = GenerateChildrenForDepth(tree.depth);

Tree.Insert(Children);

Result = Children.forEach(LDFS(Childre[i], depthLimit));

**If**(result) : **return** result

2)A\*

**Function** AStar-Search(Tree):

Priority\_queue<Node\*, vector<Node\*>, COMPARATOR> q;

q.push(tree.root)

**while**(!q.empty()):

currentNode = q.pop();

**if**(checkStateForWin(currentNode)) **return** currentNode→state;

**else**:

children = GenerateChildrenForDepth(currentNode.depth+1);

children.forEach(q.push(children[i]));

**return** null;

**COMPARATOR function**(Node\* a, Node\* b):

Return a → countPairsOfBittenQueens() + a → depth \* 3 > b → countPairsOfBittenQueens() + b → depth \* 3

## Програмна реалізація

### Вихідний код

Node.h

#pragma once

#include <cstddef>

#include <iostream>

#include <vector>

using namespace std;

class Node

{

bool checkRow(int);

bool checkCol(int);

bool checkDiagonals(int, int);

int countConflictsInColumnAfterRow(int, int);

int countConflictsInDiagonals(int, int);

public:

int\*\* state;

Node\* parent;

vector<Node\*> children;

int depth = 0;

Node(Node\*);

Node(Node\* ,int\*\*);

bool checkStateForWin();

int countPairsOfBittenQueens();

Node\* createNewState(int, int);

vector<Node\*> generateChildrenNodes(int);

};

Node.cpp

#include "Node.h"

Node::Node(Node\* parentNode)

{

parent = parentNode;

for (int i = 0; i < 8; i++) {

for (int j = 0; j < 8; j++)

{

state[i][j] = 0;

}

}

}

Node::Node(Node\* parentNode, int\*\* newState)

{

state = new int\* [8];

for (int i = 0; i < 8; i++)

{

state[i] = new int[8];

}

parent = parentNode;

for (int i = 0; i < 8; i++) {

for (int j = 0; j < 8; j++)

{

state[i][j] = newState[i][j];

}

}

}

bool Node::checkStateForWin()

{

const int length = 8;

for (int i = 0; i < length; i++)

{

for (int j = 0; j < length; j++)

{

if (state[i][j] == 1) {

if (!this->checkRow(i)) return false;

if (!this->checkCol(j)) return false;

if (!this->checkDiagonals(i, j)) return false;

}

}

}

return true;

}

int Node::countPairsOfBittenQueens()

{

const int length = 8;

int counter = 0;

for (int i = 0; i < length; i++)

{

for (int j = 0; j < length; j++)

{

if (state[i][j] == 1) {

counter += this->countConflictsInColumnAfterRow(i, j);

counter += this->countConflictsInDiagonals(i, j);

}

}

}

return counter;

}

bool Node::checkRow(int row)

{

int counter = 0;

for (int j = 0; j < 8; j++)

{

if (state[row][j] == 1) counter++;

}

return counter == 1;

}

bool Node::checkCol(int col)

{

int counter = 0;

for (int i = 0; i < 8; i++)

{

if (state[i][col] == 1) counter++;

}

return counter == 1;

}

bool Node::checkDiagonals(int row, int col)

{

int counter = 0;

for (int i = row, j = col; i < 8 && j < 8; ++i, ++j)

{

if (state[i][j] == 1)counter++;

}

for (int i = row+1, j=col-1;j>=0 && i<8;i++, j--)

{

if (state[i][j] == 1)counter++;

}

return counter == 1;

}

int Node::countConflictsInDiagonals(int row, int col)

{

int counter = 0;

for (int i = row, j = col; i < 8 && j < 8; ++i, ++j)

{

if (state[i][j] == 1)counter++;

}

for (int i = row + 1, j = col - 1; j >= 0 && i < 8; i++, j--)

{

if (state[i][j] == 1)counter++;

}

return counter;

}

int Node::countConflictsInColumnAfterRow(int row, int col)

{

int counter = 0;

const int length = 8;

row++;

for (int i = row; i < length; i++)

{

if (state[i][col] == 1) {

counter++;

}

}

return counter;

}

Node\* Node::createNewState(int row, int col)

{

int\*\* stateCopy = new int\* [8];

//copy state

for (int i = 0; i < 8; i++)

{

stateCopy[i] = new int[8];

}

//copy values

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

stateCopy[i][j] = state[i][j];

}

}

//move queen

for (int j = 0; j < 8; j++)

{

if (stateCopy[row][j] == 1 && col != j) {

stateCopy[row][j] = 0;

}

else if (j == col) {

stateCopy[row][col] = 1;

}

}

//new node

Node\* nodeNewState = new Node(nullptr, stateCopy);

return nodeNewState;

}

vector<Node\*> Node::generateChildrenNodes(int row)

{

vector<Node\*> childrenNodes;

for (int j = 0; j < 8; j++)

{

Node\* newNode = this->createNewState(row, j);

if (newNode != NULL) {

childrenNodes.push\_back(newNode);

}

}

return childrenNodes;

}

SearchTree.h

#pragma once

#include "Node.h"

#include <queue>

class SearchTree

{

public:

Node\* root;

int depth;

SearchTree(Node\*, int);

void Insert(Node\* insertIntoNode);

static int\*\* LDFS(SearchTree tree, int depthLimit);

static int\*\* ASTAR(SearchTree tree);

static void displayBoard(int\*\* board);

};

SearchTree.cpp

#include "SearchTree.h"

SearchTree::SearchTree(Node\* node, int newDepth)

{

root = node;

depth = newDepth;

}

void SearchTree::Insert(Node\* nodeToInsert)

{

root->children.push\_back(nodeToInsert);

}

int\*\* SearchTree::LDFS(SearchTree tree, int depthLimit)

{

bool isWin = tree.root->checkStateForWin();

if (isWin) {

return tree.root->state;

}

else{

if (tree.depth > depthLimit) {

return nullptr;

}

vector<Node\*> childrens = tree.root->generateChildrenNodes(tree.depth);

for (int j = 0; j < childrens.size(); j++)

{

tree.Insert(childrens[j]);

}

for (int j = 0; j < childrens.size(); j++)

{

int\*\* result = new int\* [8];

//copy state

for (int i = 0; i < 8; i++)

{

result[i] = new int[8];

}

result = SearchTree::LDFS(SearchTree(childrens[j], tree.depth+1), depthLimit);

if (result != nullptr) {

return result;

}

}

return nullptr;

}

}

class Compare

{

public:

bool operator() (Node\* a, Node\* b)

{

return a->countPairsOfBittenQueens()+a->depth\*3 > b->countPairsOfBittenQueens()+b->depth\*3;

}

};

void printQueue(priority\_queue<Node\*, vector<Node\*>, Compare> q) {

while (!q.empty()) {

cout << '\n';

SearchTree::displayBoard(q.top()->state);

q.pop();

}

}

int\*\* SearchTree::ASTAR(SearchTree tree)

{

priority\_queue<Node\*, vector<Node\*>, Compare> open;

open.push(tree.root);

while (!open.empty()) {

Node\* current = open.top();

open.pop();

if (current->checkStateForWin()) {

return current->state;

}

vector<Node\*> childrens = current->generateChildrenNodes(current->depth+1);

current->children = childrens;

for (int i = 0; i < childrens.size(); i++)

{

childrens[i]->depth = current->depth+1;

open.push(childrens[i]);

}

}

return nullptr;

}

void SearchTree::displayBoard(int\*\* board)

{

const int size = 8;

if (board == nullptr) {

cout << "not found";

return;

}

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

cout << board[i][j] << ' ';

}

cout << endl;

}

}

Main.cpp

#include <iostream>

#include <string>

#include <ctime>

#include "SearchTree.h"

using namespace std;

int\*\* generateState(int\*\*);

int main()

{

srand(time(NULL));

string method;

cout << "What do you want to use? 1-LDFS/2-A\* : ";

getline(cin,method);

int\*\* initialState;

int size = 8;

initialState = new int\* [size];

for (int i = 0; i < size; i++)

{

initialState[i] = new int[size];

}

initialState = generateState(initialState);

Node\* root = new Node(nullptr, initialState);

cout << endl << "===============" << endl << "INITIAL STATE"<<endl << "===============" << endl;

SearchTree::displayBoard(root->state);

SearchTree tree(root, 0);

if (method == "1") {

const int depthLimit = 5;

int\*\* result = new int\* [8];

for (int i = 0; i < 8; i++) {

result[i] = new int[8];

}

result = SearchTree::LDFS(tree, depthLimit);

cout << endl << "===============" << endl << "RESULT" << endl << "===============" << endl;

SearchTree::displayBoard(result);

}

else if (method == "2") {

int\*\* result = new int\* [8];

for (int i = 0; i < 8; i++) {

result[i] = new int[8];

}

result = SearchTree::ASTAR(tree);

cout << endl << "===============" << endl << "RESULT" << endl << "===============" << endl;

SearchTree::displayBoard(result);

}

else {

cout << "ERROR" << endl;

}

return 0;

}

int\*\* generateState(int\*\* result) {

const int size = 8;

for (int i = 0; i < size; i++)

{

int queen = rand() % 8;

for (int j = 0; j < size; j++) {

if (j == queen) {

result[i][j] = 1;

}

else {

result[i][j] = 0;

}

}

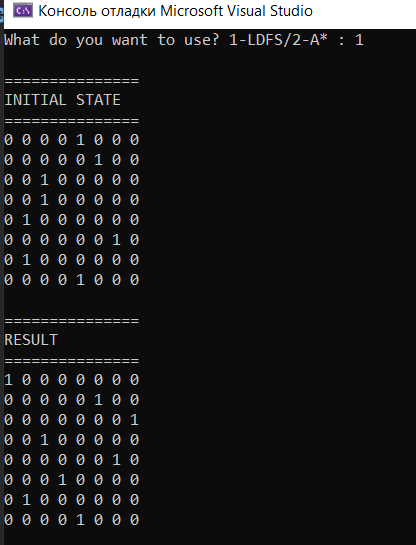
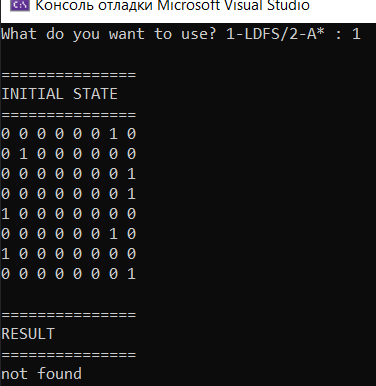
}

return result;

}

### Приклади роботи

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

Рисунок 3.1 – Алгоритм LDFS з depthLimit = 5

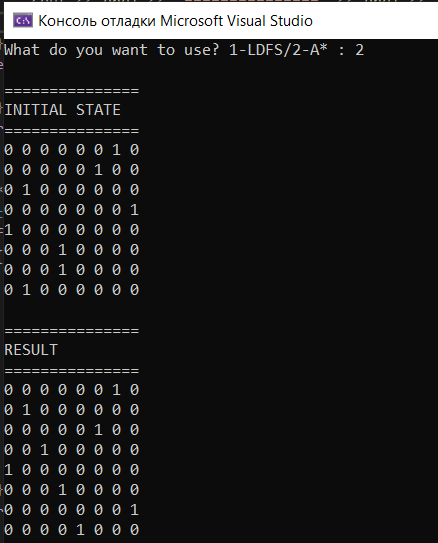


Рисунок 3.2 – Алгоритм A\*

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму LDFS, задачі про 8 ферзів для 20 початкових станів за умови обмеження глибини 5.

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пом’яті |
| Стан 1 | 120743 | 105647 | 120743 | 41 |
| Стан 2 | 97949 | 85702 | 97949 | 41 |
| Стан 3 | 99336 | 86915 | 99336 | 41 |
| Стан 4 | 299593 | 262144 | 299593 | 0 |
| Стан 5 | 67144 | 58748 | 67144 | 41 |
| Стан 6 | 299593 | 262144 | 299593 | 0 |
| Стан 7 | 100132 | 87612 | 100132 | 41 |
| Стан 8 | 299593 | 262144 | 299593 | 0 |
| Стан 9 | 299593 | 262144 | 299593 | 0 |
| Стан 10 | 178857 | 156496 | 178857 | 41 |
| Стан 11 | 299593 | 262144 | 299593 | 0 |
| Стан 12 | 157466 | 137779 | 157466 | 41 |
| Стан 13 | 299593 | 262144 | 299593 | 0 |
| Стан 14 | 156694 | 137104 | 156694 | 41 |
| Стан 15 | 299593 | 262144 | 299593 | 0 |
| Стан 16 | 67144 | 58748 | 67144 | 41 |
| Стан 17 | 54953 | 48080 | 54953 | 41 |
| Стан 18 | 299593 | 262144 | 299593 | 0 |
| Стан 19 | 99369 | 86944 | 99369 | 41 |
| Стан 20 | 59715 | 52247 | 59715 | 41 |

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

Таблиця 3.2 – Характеристики оцінювання алгоритму A\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пом’яті |
| Стан 1 | 12802 | 2507 | 12802 | 89607 |
| Стан 2 | 875 | 204 | 875 | 6118 |
| Стан 3 | 11444 | 2154 | 11444 | 80101 |
| Стан 4 | 88 | 15 | 88 | 609 |
| Стан 5 | 8323 | 1937 | 8323 | 58254 |
| Стан 6 | 1670 | 345 | 1670 | 11683 |
| Стан 7 | 89 | 10 | 89 | 616 |
| Стан 8 | 9661 | 1570 | 9661 | 67620 |
| Стан 9 | 8030 | 1798 | 8030 | 56203 |
| Стан 10 | 2012 | 398 | 2012 | 14077 |
| Стан 11 | 2707 | 471 | 2707 | 18942 |
| Стан 12 | 1872 | 336 | 1872 | 13097 |
| Стан 13 | 10117 | 1774 | 10117 | 70812 |
| Стан 14 | 8619 | 1815 | 8619 | 60326 |
| Стан 15 | 1160 | 253 | 1160 | 8113 |
| Стан 16 | 1084 | 254 | 1084 | 7581 |
| Стан 17 | 7189 | 1436 | 7189 | 50316 |
| Стан 18 | 93 | 15 | 93 | 644 |
| Стан 19 | 1483 | 259 | 1483 | 10374 |
| Стан 20 | 145 | 27 | 145 | 1008 |

Висновок

При виконанні даної лабораторної роботи було розглянуто задачу про вісім ферзів, та її розв’язання використовуючи інформативний(A\*) та неінформативний(LDFS) пошуки. Було написано програмну реалізацію та псевдокод. З проведених досліджень, можна переконатися в тому, що LDFS(який є модифікацією DFS) потребує дуже малих затрат в пам’яті( а саме mb+1, де m-коефіцієнт розгалудження, а b-максимальна глибина пошуку), але витрачає більше часу на ітерації тощо. А\* в свою чергу є більш ефективним, адже 1)завжди знаходить відповідь до задачі 2) робить це швидко(кількість ітерацій), але попри це потребує більших витрат у пам’яті, для зберігання черги наступних станів.

Критерії оцінювання

За умови здачі лабораторної роботи до 25.09.2021 включно максимальний бал дорівнює – 5. Після 25.09.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 10%;
* програмна реалізація алгоритму – 60%;
* дослідження алгоритмів – 25%;
* висновок – 5%.