**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)**

(шифр, прізвище, ім'я, по батькові)

ІП-14 Прокопенко Олексій

Київ 2022

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 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 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.
* **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 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** | **Func** |
| 13 | 8-ферзів | BFS | RBFS |  | F1 |

# Виконання

## Псевдокод алгоритмів

## Програмна реалізація

### Вихідний код

1.2.1.1

**BFS**

**Cell.cs**

public class Cell

{

public int X { get; set; }

public int Y { get; set; }

public bool IsEmpty { get; set; }

public Cell(int x, int y)

{

X = x;

Y = y;

IsEmpty = true;

}

}

**Grid.cs**

public class Grid

{

public Cell[,] Map { get; set; }

public Cell this[int x, int y]

{

get => Map[x, y];

set => Map[x, y] = value;

}

public Grid()

{

Map = new Cell[8, 8];

for (int i = 0; i < Map.GetLength(0); i++)

{

for (int j = 0; j < Map.GetLength(1); j++)

{

Map[i, j] = new Cell(i,j);

}

}

}

public void Print()

{

for (int i = 0; i < Map.GetLength(0); i++)

{

for (int j = 0; j < Map.GetLength(1); j++)

{

var str = Map[i, j].IsEmpty ? "." : "\_";

Console.Write(str + " ");

}

Console.WriteLine();

}

}

}

**BFS.cs**

public class BFS

{

private Grid Map { get; set; }

public BFS(Grid map)

{

Map = map;

}

public bool Start(int strNum)

{

if (strNum == 8)

{

return true;

}

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

{

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

{

Map[j,strNum].IsEmpty = true;

}

if (Check(Map[i, strNum]))

{

Map[i, strNum].IsEmpty = false;

if (Start(strNum+1))

{

return true;

}

}

}

return false;

}

private bool Check(Cell cell)

{

int x = cell.X;

int y = cell.Y;

if (!cell.IsEmpty)

{

return false;

}

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

{

if (!Map[x,i].IsEmpty)

{

return false;

}

if (x - 1 - i >= 0 && !Map[x-1-i,y-1-i].IsEmpty)

{

return false;

}

if (x + 1 + i < 8 && !Map[x+1+i, y-1-i].IsEmpty)

{

return false;

}

}

return true;

}

public bool IsValid()

{

int count = 0;

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

{

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

{

if (!Map[i,j].IsEmpty)

{

count++;

var cell = Map[i, j];

int x = cell.X;

int y = cell.Y;

for (int k = 0; k < y; k++)

{

if (!Map[x,i].IsEmpty)

{

return false;

}

if (x - 1 - k >= 0 && !Map[x-1-k,y-1-k].IsEmpty)

{

return false;

}

if (x + 1 + k < 8 && !Map[x+1+k, y-1-k].IsEmpty)

{

return false;

}

}

}

}

}

return count == 8;

}

}

1.2.1.2

**RBFS**

**main.py**

from RBFS.RBFS import RBFS

import random

import time

def generate\_chessboards(number):

result = []

for \_ in range(number):

board = []

for j in range(8):

is\_added = False

while not is\_added:

queen = [random.randint(0, 7), j]

if queen not in board:

board.append(queen)

is\_added = True

result.append(board)

return result

def print\_chessboard (chessboard):

l = ""

for i in range(8):

for j in range(8):

queen = [i,j]

if queen in chessboard:

l += '\_ '

else:

l += ". "

l += "\n"

return l

chessboards = [

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [3, 7]],

[[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6], [0, 7]],

[[1, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [3, 7]],

[[3, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [5, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [9, 7]],

[[4, 0], [0, 1], [7, 2], [3, 3], [2, 4], [6, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [2, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [0, 7]],

[[4, 0], [0, 1], [0, 2], [5, 3], [2, 4], [6, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [6, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [0, 4], [6, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [5, 2], [5, 3], [2, 4], [6, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [6, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [8, 3], [2, 4], [6, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [5, 7]],

[[4, 0], [6, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [9, 5], [1, 6], [3, 7]],

[[4, 0], [0, 1], [7, 2], [5, 3], [2, 4], [6, 5], [1, 6], [9, 7]],

[[4, 0], [0, 1], [3, 2], [5, 3], [2, 4], [6, 5], [1, 6], [3, 7]]

]

print('------RBFS------')

for chessboard in generate\_chessboards(20):

start\_time = time.time()

rbfs = RBFS(chessboard)

print('Input chessboard: \n' + print\_chessboard(chessboard))

rbfs.recursive\_best\_first\_search()

print('Result chessboard: \n' + print\_chessboard(rbfs.answer.result\_chessboard))

print('states number = ' + str(rbfs.states\_number))

print('iterations = ' + str(rbfs.iterations))

print('dead ends = ' + str(rbfs.dead\_ends))

print("%s seconds" % (time.time() - start\_time))

print()

**RBFS.py**

from RBFS.answer import Answer

from queue import PriorityQueue

import copy

class RBFS:

def \_\_init\_\_(self, initial\_chessboard):

self.initial\_chessboard = initial\_chessboard

self.answer = None

self.\_\_checked\_states = []

self.iterations = 0

self.states\_number = 0

self.dead\_ends = 0

self.\_\_queue = PriorityQueue()

def recursive\_best\_first\_search(self):

self.answer = self.\_\_rbfs(self.initial\_chessboard, float("inf"))

def \_\_rbfs(self, current\_state, f\_limit):

self.iterations += 1

if RBFS.\_\_chessboard\_validation(current\_state):

return Answer(True, result\_chessboard=current\_state)

successors = self.\_\_get\_successors(current\_state)

if len(successors) == 0:

self.dead\_ends += 1

return Answer(False, f\_limit=float("inf"))

for successor in successors:

self.\_\_queue.put((RBFS.\_\_heuristic\_func(successor), successor))

best = self.\_\_queue.get()

if best[0] > f\_limit:

return Answer(False, f\_limit=best[0])

alternative = self.\_\_queue.get()

self.\_\_queue.empty()

result = self.\_\_rbfs(best[1], min(best, alternative)[0])

if result is not None and result.is\_success:

return result

else:

self.dead\_ends += 1

return self.\_\_rbfs(self.\_\_queue.get()[1], float('inf'))

def \_\_get\_successors(self, chessboard):

result = []

for i in range(8):

for j in range(8):

temp = copy.deepcopy(chessboard)

temp[i][0] = j

if not chessboard[i][0] == j and temp not in self.\_\_checked\_states:

result.append(copy.deepcopy(temp))

self.\_\_checked\_states.append(temp)

self.states\_number += len(result)

return result

def \_\_heuristic\_func(chessboard):

result = 0

for i in range(8):

for j in range(i + 1, 8):

if chessboard[i][0] == chessboard[j][0] or chessboard[i][1] == chessboard[j][1] \

or abs(chessboard[i][0] - chessboard[j][0]) == abs(chessboard[i][1] - chessboard[j][1]):

result += 1

return result

def \_\_chessboard\_validation(chessboard):

for i in range(8):

for j in range(i + 1, 8):

if chessboard[i][0] == chessboard[j][0] or chessboard[i][1] == chessboard[j][1] \

or abs(chessboard[i][0] - chessboard[j][0]) == abs(chessboard[i][1] - chessboard[j][1]):

return False

return True

### Приклади роботи

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

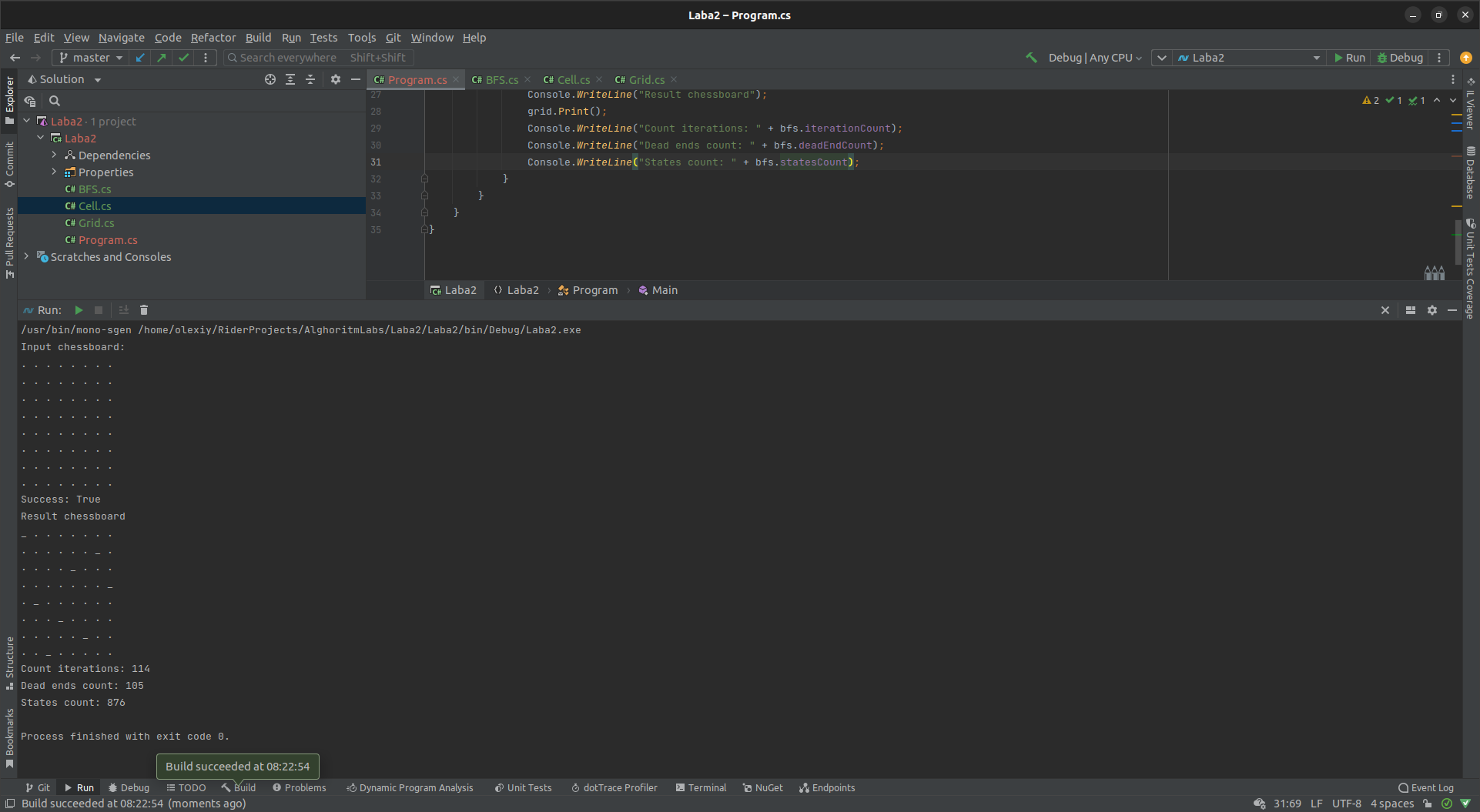


Рисунок 3.1 – Алгоритм **BFS**

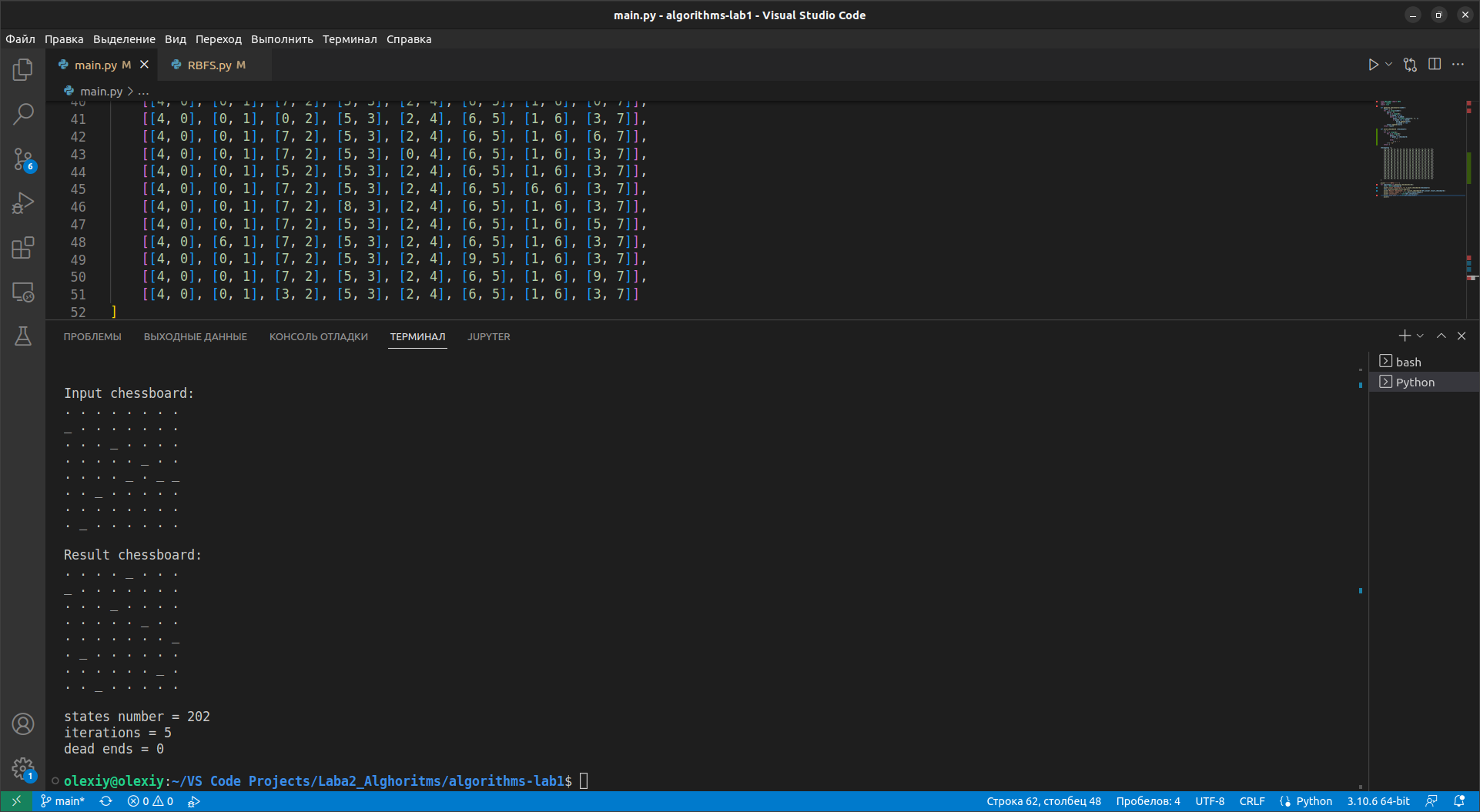


Рисунок 3.2 – Алгоритм **RBFS**

## Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пом’яті |
| Стан 1 | 114 | 105 | 867 |  |
| Стан 2 | 44 | 36 | 316 |  |
| Стан 3 | 33 | 26 | 238 |  |
| Стан 4 | 28 | 20 | 194 |  |
| Стан 5 | 20 | 12 | 126 |  |
| Стан 6 | 11 | 5 | 64 |  |
| Стан 7 | 19 | 11 | 117 |  |
| Стан 8 | 15 | 8 | 94 |  |
| Стан 9 | 113 | 105 | 875 |  |
| Стан 10 | 18 | 11 | 115 |  |
| Стан 12 | 49 | 41 | 361 |  |
| Стан 13 | 41 | 34 | 295 |  |
| Стан 14 | 31 | 24 | 213 |  |
| Стан 15 | 27 | 19 | 184 |  |
| Стан 16 | 21 | 13 | 137 |  |
| Стан 17 | 9 | 1 | 39 |  |
| Стан 18 | 9 | 2 | 49 |  |
| Стан 19 | 56 | 40 | 369 |  |
| Стан 20 | 45 | 31 | 287 |  |

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

Таблиця 3.3 – Характеристики оцінювання алгоритму RBFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пом’яті |
| Стан 1 | 10 | 1 | 442 |  |
| Стан 2 | 31 | 1 | 1444 |  |
| Стан 3 | 10 | 0 | 441 |  |
| Стан 4 | 5 | 0 | 202 |  |
| Стан 5 | 9 | 0 | 394 |  |
| Стан 6 | 8 | 0 | 345 |  |
| Стан 7 | 18 | 1 | 824 |  |
| Стан 8 | 12 | 0 | 536 |  |
| Стан 9 | 7 | 0 | 298 |  |
| Стан 10 | 6 | 0 | 250 |  |
| Стан 12 | 14 | 1 | 631 |  |
| Стан 13 | 4 | 0 | 154 |  |
| Стан 14 | 13 | 0 | 585 |  |
| Стан 15 | 3 | 0 | 106 |  |
| Стан 16 | 15 | 0 | 682 |  |
| Стан 17 | 12 | 0 | 537 |  |
| Стан 18 | 10 | 0 | 442 |  |
| Стан 19 | 52 | 3 | 2444 |  |
| Стан 20 | 20 | 1 | 922 |  |

Висновок

При виконанні даної лабораторної роботи було розглянуто два алгоритми: BFS та RBFS. У алгоритмі неінформативно пошуку при пошуку рішення на пустій шаховій дошці проходить 114 ітерацій, 105 випадків коли програма неправильно поставила фігуру на дошку, і потрібно було її видаляти та знаходити рішення заново, та 876 станів, при виставленні однієї фігури на дошку, до початку пошуку рішення, його вдалося знаходити вавжди за меншу кількість ітерацій (наприклад: 49 ітерацій, 41 глухий кут, 361 стан), при виставленні двох фігур рішення знаходилось завжди (окрім випадків коли виставлені фігури б’ють одна одну), за наприклад 41 ітерацію, 34 глухих кути, 295 станів. При виставленні трьох фігур рішення знаходилось в 1 з 3 випадків, за наприклад 11 ітерацій, 5 глухих кутів, 64 стани. У алгоритмі інформативного пошуку було використано інший підхід, коли на шаховій дошці уже були рандомно розставлені фігури, і їх потрібно розставити правильно. Тому рішення знаходилось завжди. Також було використано евристичну функцію для оцінки «ваги» ходу, та чергу для додавання ходів в порядку їх зростання, та ходу в клітинку з найбільшою вагою. Якщо після ходу в клітинку з найбільшою вагою програма потрапляє в тупик, то вона відміняє цей хід, і робить хого в наступні клітинку в черзі. Зазвичай програма знаходить рішення за >14 ітерацій та ~630 станів, якщо програма потрапляє в шлухий кут, то їх кількість збільшується.

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%.