**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-25 Карпов Л.В.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2023

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 12](#_Toc81070694)

[3.3 Дослідження алгоритмів 14](#_Toc81070695)

[Висновок 17](#_Toc81070696)

[Критерії оцінювання 18](#_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** |
| 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 |

# Виконання

## Псевдокод алгоритмів

NSA

Func solve(depth)

if depth < 0

return

Check if solved

Return state

Expand node

solve(depth – 1, new\_state)

ISA

*states* = [TUPLE(0, initial\_state)]

while true

sort *states* by (price + queens\_under\_attack)

*price*, *state* = first of *states*

expand node and add new states to *states*

if any of states is solved

return this state

## Програмна реалізація

### Вихідний код

import functools  
import random  
import time  
  
# Empty = object()  
# UnderAttack = object()  
# Queen = object()  
BS = 8 # Board Size  
QC = 8 # Queen Count  
  
  
class Cell:  
 x: int  
 y: int  
  
 def \_\_init\_\_(self, x: int = None, y: int = None):  
 self.x = x if x is not None else random.randint(0, BS - 1)  
 self.y = y if y is not None else random.randint(0, BS - 1)  
  
 def \_\_eq\_\_(self, other: 'Cell'):  
 return self.x == other.x and self.y == other.y  
  
 def \_\_str\_\_(self):  
 return f"Cell({self.x}, {self.y})"  
  
  
class State:  
 solve\_counter = 1  
 memory = 0  
 max\_depth = 0  
 queens: list[Cell]  
 states: list['State']  
  
 # exceptions: list[Cell]  
  
 def \_\_init\_\_(self, queens: list[Cell] = None): # , exceptions: list[Cell] = None  
 self.states = []  
 if queens is None:  
 self.queens = []  
 for i in range(QC):  
 while (cell := Cell()) in self.queens:  
 pass  
 self.queens.append(cell)  
 else:  
 self.queens = queens  
  
 @property  
 def under\_attack(self): # F2 function  
 board = [[0 for \_ in range(BS)] for \_ in range(BS)]  
  
 for queen in self.queens:  
 for i in range(BS):  
 if i != queen.x:  
 board[i][queen.y] += 1  
 if i != queen.y:  
 board[queen.x][i] += 1  
  
 if 0 <= (j := queen.y - queen.x + i) < BS:  
 if not (i == queen.x and j == queen.y):  
 board[i][j] += 1  
  
 if 0 <= (j := (queen.x + queen.y) - i) < BS:  
 if not (i == queen.x and j == queen.y):  
 board[i][j] += 1  
  
 n = 0  
  
 for queen in self.queens:  
 n += board[queen.x][queen.y]  
  
 return n // 2  
  
 @property  
 def is\_solved(self):  
 return self.under\_attack == 0  
  
 def get\_board(self, queens=None):  
 if queens is None:  
 queens = self.queens  
  
 board = [[0 for \_ in range(BS)] for \_ in range(BS)]  
  
 for queen in queens:  
 for i in range(BS):  
 if board[i][queen.y] == 0:  
 board[i][queen.y] = 1  
 if board[queen.x][i] == 0:  
 board[queen.x][i] = 1  
  
 if 0 <= (j := queen.y - queen.x + i) < BS:  
 if board[i][j] == 0:  
 board[i][j] = 1  
  
 if 0 <= (j := (queen.x + queen.y) - i) < BS:  
 if board[i][j] == 0:  
 board[i][j] = 1  
  
 board[queen.x][queen.y] = 2  
  
 return board  
  
 def draw\_board(self):  
 board = [['#' for \_ in range(BS)] for \_ in range(BS)]  
  
 for queen in self.queens:  
 board[queen.x][queen.y] = '&'  
 return '\n'.join([' '.join(i) for i in board])  
  
 @staticmethod  
 def create\_new\_states(state: 'State'):  
 states = []  
  
 for queen\_to\_move in state.queens:  
 queens = [queen for queen in state.queens if queen is not queen\_to\_move]  
  
 assert QC - 1 == len(queens)  
 board = state.get\_board(queens)  
  
 for i in range(BS):  
 for j in range(BS):  
 if board[i][j] == 0:  
 new\_state = State(queens.copy())  
 new\_state.queens.append(Cell(i, j))  
 states.append(new\_state)  
 assert state.under\_attack >= new\_state.under\_attack  
 return states  
  
 def solve\_NSA(self, depth: int = QC - 1):  
 State.solve\_counter += 1  
  
 if depth < 0:  
 return  
  
 if self.is\_solved:  
 State.max\_depth = depth  
 return self  
  
 new\_states = State.create\_new\_states(self)  
 State.solve\_counter += len(new\_states)  
 for state in new\_states:  
 if (res := state.solve\_NSA(depth - 1)) is not None:  
 return res  
  
 State.memory += len(new\_states)  
  
 def solve\_ISA(self):  
  
 if self.is\_solved:  
 return self  
  
 states = [(0, self)]  
  
 iterations = 0  
 states\_total = 1  
  
 while True:  
  
 states.sort(key=lambda x: x[0] + x[1].under\_attack, reverse=True) # A\*  
 iterations += 1  
  
 price, state = states.pop()  
 # print(len(states), price, state.under\_attack)  
  
 new\_states = [(price + 1, state) for state in State.create\_new\_states(state)]  
 states\_total += len(new\_states)  
  
 states += new\_states  
  
 for pr, st in states:  
 if st.is\_solved:  
 print("Iter: ", iterations)  
 print("States: ", states\_total)  
 print("Memory: ", len(states))  
 return st  
  
  
DEPTH = 7  
  
  
def NSA():  
 state = State()  
 print(state.draw\_board(), end='\n\n')  
 print("Solved: ", state.is\_solved)  
 print("Func:", state.under\_attack)  
 start\_time = time.time()  
 solved\_state = state.solve\_NSA(DEPTH)  
 end\_time = time.time()  
  
 if solved\_state is not None:  
 print(solved\_state.draw\_board(), end='\n\n')  
 print("Solved: ", solved\_state.is\_solved)  
  
 print("Elapsed time: ", end\_time - start\_time)  
 print("Iter:", State.solve\_counter)  
 print("Memory:", State.solve\_counter - State.memory)  
 print("Depth : ", DEPTH - State.max\_depth)  
  
  
def ISA():  
 state = State()  
 print(state.draw\_board(), end='\n\n')  
 print("Solved: ", state.is\_solved)  
 print("Func:", state.under\_attack)  
 start\_time = time.time()  
 solved\_state = state.solve\_ISA()  
 end\_time = time.time()  
  
 if solved\_state is not None:  
 print(solved\_state.draw\_board(), end='\n\n')  
 print("Solved: ", solved\_state.is\_solved)  
  
 print("Elapsed time: ", end\_time - start\_time)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # NSA()  
 ISA()

### Приклади роботи

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

Рисунок 3.1 – Алгоритм АНП LDFS

A screenshot of a computer screen

Description automatically generated

Рисунок 3.2 – Алгоритм АІП A\*

A screenshot of a computer

Description automatically generated

## Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання LDFS

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Грибина | Всього станів | Всього станів у пом’яті |
| Стан 1 | 7 | 144601 | 72335 |
| Стан 2 | 7 | 13938 | 7011 |
| Стан 3 | 6 | 150114 | 75087 |
| Стан 4 | 7 | 92 | 92 |
| Стан 5 | 6 | 86 | 86 |
| Стан 6 | 7 | 24540 | 12302 |
| Стан 7 | 5 | 735 | 395 |
| Стан 8 | 5 | 65 | 65 |
| Стан 9 | 7 | 67015 | 33542 |
| Стан 10 | 7 | 40985 | 20530 |
| Стан 11 | 7 | 119628 | 59866 |
| Стан 12 | 7 | 1226 | 652 |
| Стан 13 | 7 | 303 | 189 |
| Стан 14 | 5 | 38555 | 19308 |
| Стан 15 | 7 | 1043 | 578 |
| Стан 16 | 7 | 2651 | 1364 |
| Стан 17 | 7 | 5084359 | 2542223 |
| Стан 18 | 7 | 62213 | 31155 |
| Стан 19 | 7 | 2305 | 1209 |
| Стан 20 | 7 | 2794 | 1431 |

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

Таблиця 3.2 – Характеристики оцінювання АІП A\*

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пом’яті |
| Стан 1 | 6 | 39 | 33 |
| Стан 2 | 8 | 97 | 89 |
| Стан 3 | 63 | 443 | 380 |
| Стан 4 | 3 | 28 | 25 |
| Стан 5 | 43 | 268 | 225 |
| Стан 6 | 7 | 126 | 119 |
| Стан 7 | 20 | 126 | 106 |
| Стан 8 | 25 | 129 | 104 |
| Стан 9 | 150 | 901 | 751 |
| Стан 10 | 7 | 119 | 112 |
| Стан 11 | 23 | 164 | 141 |
| Стан 12 | 37 | 270 | 233 |
| Стан 13 | 36 | 259 | 223 |
| Стан 14 | 167 | 1069 | 902 |
| Стан 15 | 139 | 719 | 580 |
| Стан 16 | 176 | 1030 | 854 |
| Стан 17 | 4 | 46 | 42 |
| Стан 18 | 15 | 125 | 110 |
| Стан 19 | 5 | 54 | 49 |
| Стан 20 | 173 | 1056 | 883 |

Висновок

При виконанні даної лабораторної роботи було розглянуто два алгоритми пошуку рішень задачі про 8 ферзів алгоритм неінформативного пошуку LDFS та алгоритм інформованого пошуку A\*.

Перший алгоритм можна назвати сліпим перебором станів та їх перевіркою. Щоб уникнути знаходження занадто довгих рішень було встановленне обмеження глибини розгортання вузла, оскільки ферзів на шахівниці вісім то в найгіршому початковому стані прийдеться змінити позицію семи з них, таке обмеження в глибині я і задав.

Другий алгоритм розгортає вузол який вважає найкращим для поточного набору, це забезпечує знаходження коротшого вирішення проблеми але також збільшує обєм опрацьованих данних оскільки алгоритм одночасно може працювати над великою кількістю віток.

Критерії оцінювання

За умови здачі лабораторної роботи до 5.11.2023 включно максимальний бал дорівнює – 5. Після 5.11.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 10%;
* програмна реалізація алгоритму – 40%;
* робота з гіт – 20%;
* дослідження алгоритмів – 25%;
* висновок – 5%.