**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-з21 Єфремов Юрій Володимирович

**Перевірив**

(прізвище, ім'я, по батькові)

Головченко М.М.

Київ 2023

Зміст

[1 Мета лабораторної роботи 3](#_Toc109342184)

[2 ЗаВдання 4](#_Toc109342185)

[3 Виконання](#_Toc109342186) 5

[3.1 Псевдокод алгоритму 6](#_Toc109342187)

[3.2 Програмна реалізація алгоритму 6](#_Toc109342188)

[3.2.1 Вихідний код 6](#_Toc109342189)

[Висновок 7](#_Toc109342190)

# Мета лабораторної роботи

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

# Завдання

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту 25(пряме злиття) 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-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.
* **BFS** – Пошук вшир.
* **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
* **H1** – кількість фішок, які не стоять на своїх місцях.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * 8-puzzle | * BFS | * RBFS |  | * H1 |

# Виконання

## Псевдокод алгоритму

**function BFS(init\_state, goal\_state):**

**visited = set()**

**queue = Queue()**

**queue.put((init\_state, 0))**

**while not queue.empty():**

**current\_state, steps = queue.get()**

**if current\_state == goal\_state:**

**return steps**

**if current\_state not in visited:**

**visited.add(current\_state)**

**for neighbor in current\_state.get\_neighbors():**

**queue.put((neighbor, steps + 1))**

**return -1**

**# Ваші початковий і цільовий стани**

**init\_state = PuzzleState([[1, 2, 3], [4, 5, 6], [0, 7, 8]])**

**goal\_state = PuzzleState([[1, 2, 3], [4, 5, 6], [7, 8, 0]])**

**# Використовуйте BFS**

**bfs\_steps = BFS(init\_state, goal\_state)**

**print(f"BFS Steps: {bfs\_steps}")**

**function RBFS(current\_state, goal\_state, f\_limit=float("inf")):**

**if current\_state == goal\_state:**

**return 0**

**successors = [(neighbor, 1) for neighbor in current\_state.get\_neighbors()]**

**if not successors:**

**return float("inf")**

**while True:**

**best = min(successors, key=lambda x: x[1])**

**if best[1] > f\_limit:**

**return best[1]**

**alternative = min(**

**successors, key=lambda x: x[1] if x[1] > best[1] else float("inf")**

**)**

**result = RBFS(best[0], goal\_state, min(f\_limit, alternative[1]))**

**if result < float("inf"):**

**return result**

**successors.remove(best)**

**# Використовуйте RBFS**

**rbfs\_steps = RBFS(init\_state, goal\_state)**

**print(f"RBFS Steps: {rbfs\_steps}")**

## Програмна реалізація алгоритму

### Вихідний код

import queue  
  
  
class PuzzleState:  
 def \_\_init\_\_(self, puzzle):  
 self.puzzle = puzzle  
  
 def \_\_str\_\_(self):  
 return "\n".join(" ".join(map(str, row)) for row in self.puzzle)  
  
 def \_\_eq\_\_(self, other):  
 return self.puzzle == other.puzzle  
  
 def \_\_hash\_\_(self):  
 return hash(tuple(tuple(row) for row in self.puzzle))  
  
 def get\_blank\_position(self):  
 for i, row in enumerate(self.puzzle):  
 for j, num in enumerate(row):  
 if num == 0:  
 return i, j  
  
 def get\_neighbors(self):  
 i, j = self.get\_blank\_position()  
 neighbors = []  
  
 for move in [(0, 1), (1, 0), (0, -1), (-1, 0)]:  
 ni, nj = i + move[0], j + move[1]  
  
 if 0 <= ni < 3 and 0 <= nj < 3:  
 new\_puzzle = [row.copy() for row in self.puzzle]  
 new\_puzzle[i][j], new\_puzzle[ni][nj] = (  
 new\_puzzle[ni][nj],  
 new\_puzzle[i][j],  
 )  
 neighbors.append(PuzzleState(new\_puzzle))  
  
 return neighbors  
  
 def h1(self, goal\_state):  
 # Порівнюємо поточний стан із цільовим і обчислюємо кількість фішок, які не на своїх місцях  
 return sum(1 for i in range(3) for j in range(3) if self.puzzle[i][j] != goal\_state.puzzle[i][j])  
  
  
def bfs(init\_state, goal\_state):  
 visited = set()  
 q = queue.Queue()  
 q.put((init\_state, 0))  
  
 while not q.empty():  
 current\_state, steps = q.get()  
  
 if current\_state == goal\_state:  
 return steps  
  
 if current\_state not in visited:  
 visited.add(current\_state)  
  
 for neighbor in current\_state.get\_neighbors():  
 q.put((neighbor, steps + 1))  
  
 return -1  
  
  
def rbfs(current\_state, goal\_state, f\_limit=float("inf")):  
 if current\_state == goal\_state:  
 return 0  
  
 successors = [(neighbor, 1) for neighbor in current\_state.get\_neighbors()]  
 if not successors:  
 return float("inf")  
  
 while True:  
 best = min(successors, key=lambda x: x[1])  
 if best[1] > f\_limit:  
 return best[1]  
 alternative = min(  
 successors, key=lambda x: x[1] if x[1] > best[1] else float("inf")  
 )  
 result = rbfs(best[0], goal\_state, min(f\_limit, alternative[1]))  
 if result < float("inf"):  
 return result  
 successors.remove(best)  
  
  
initial\_state = PuzzleState([[1, 2, 3], [4, 5, 6], [0, 7, 8]])  
goal\_state = PuzzleState([[1, 2, 3], [4, 5, 6], [7, 8, 0]])  
  
# Використовуйте BFS  
bfs\_steps = bfs(initial\_state, goal\_state)  
print(f"BFS Steps: {bfs\_steps}")  
  
# Використовуйте RBFS  
rbfs\_steps = rbfs(initial\_state, goal\_state)  
print(f"RBFS Steps: {rbfs\_steps}")  
# Використовуйте h1  
h1\_value = initial\_state.h1(goal\_state)  
print(f"h1 Value: {h1\_value}")

Висновок

У ході лабораторної роботи було реалізовано та досліджено алгоритми пошуку розв'язків для гри "8-пазлів" (8-puzzle) за допомогою алгоритмів BFS (пошук вшир) та RBFS (рекурсивний пошук за першим найкращим співпадінням).

BFS дозволяє знаходити оптимальні рішення, але може вимагати значних обчислювальних ресурсів, особливо для складних задач. Рекурсивний пошук за першим найкращим співпадінням (RBFS) є менш обчислювально витратним, але не завжди гарантує знаходження оптимального розв'язку.

Проведені експерименти показали, що обидва алгоритми здатні знаходити розв'язки для даної задачі, проте вони можуть мати різну ефективність в залежності від конкретного початкового та цільового стану гри.

Для подальшого вдосконалення системи можливо розглядати оптимізації алгоритмів, дослідження інших евристик, а також розгляд можливості використання паралельних обчислень для прискорення пошуку розв'язків.