**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»**

ІТ-04

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*Стрільчук Михайло Васильович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

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 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. 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** | **Func** |
| 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

Варіант 20

## Псевдокод алгоритмів

**Неінформований пошук BFS:**

*function BFS(initialState)*

*returns Success or Failure*

*frontier = Queue.new(initialState)*

*explored = Set.new()*

*while not frontier.IsEmpty():  
 state = frontier.dequeue()*

*explored.add(state)*

*if goalTest(state):*

*return Success(state)*

*for neighbor in state.neighbors():*

*if neighbor not in frontier && explored:*

*frontier.enqueue(neighbor)*

*return Failure*

**Інформований пошук RBFS:**

*function RBFS (problem,node,f-limit) returns sol & f-cost*

*if GOAL-TEST (problem, STATE (node)) then return node*

*succ= EXPAND (node, problem)*

*if succ is empty then return failure*

*for-each s in succ do*

*f [s]= max (g(s)+ h(s), f[node])*

*repeat*

*best= the lowest f-value node in succ*

*if f[best]> f-limit then return failure, f[best]*

*alternative= the second-lowest f-value among succ*

*result,f[best]= RBFS (problem, best, min(f-limit,altern*

*if result<> failure then return result*

## Програмна реалізація

### Вихідний код

BFS та RBFS на мові Python

BFS\_search.py

from queue import Queue  
from puzzle import Puzzle  
  
  
def breadth\_first\_search(initial\_state):  
 start\_node = Puzzle(initial\_state, None, None, 0)  
 if start\_node.goal\_test():  
 return start\_node.find\_solution()  
 q = Queue()  
 q.put(start\_node)  
 explored = []  
 while not(q.empty()):  
 node = q.get()  
 explored.append(node.state)  
 children = node.generate\_child()  
 for child in children:  
 if child.state not in explored:  
 if child.goal\_test():  
 return child.find\_solution()  
 q.put(child)  
 return

RBFS\_search.py

from puzzle import Puzzle  
from sys import maxsize  
  
  
def recursive\_best\_first\_search(initial\_state):  
 start\_node = Puzzle(state=initial\_state, parent=None, move=None, path\_cost=0, needs\_heuristic=True)  
 node = RBFS\_search(start\_node, f\_limit=maxsize)  
 node = node[0]  
 return node.find\_solution()  
  
def RBFS\_search(node, f\_limit):  
 successors = []  
  
 if node.goal\_test():  
 return node, None  
 children = node.generate\_child()  
 if not len(children):  
 return None, maxsize  
 count =- 1  
 for child in children:  
 count += 1  
 child\_info = (child.evaluation\_function, count, child)  
 successors.append(child\_info)  
 while len(successors):  
 successors.sort()  
 best\_node = successors[0][2]  
 if best\_node.evaluation\_function > f\_limit:  
 return None, best\_node.evaluation\_function  
 alternative = successors[1][0]  
 result, best\_node.evaluation\_function = RBFS\_search(best\_node, min(f\_limit, alternative))  
 successors[0] = (best\_node.evaluation\_function, successors[0][1], best\_node)  
 if result != None:  
 break  
 return result, None

puzzle.py

class Puzzle:  
 goal\_state = [0, 1, 2, 3, 4, 5, 6, 7, 8]  
 heuristic = None  
 evaluation\_function = None  
 needs\_heuristic = False  
 total\_states = 0  
 def \_\_init\_\_(self, state, parent, move, path\_cost, needs\_heuristic = False):  
 self.parent = parent  
 self.state = state  
 self.move = move  
 if parent:  
 self.path\_cost = parent.path\_cost + path\_cost  
 else:  
 self.path\_cost = path\_cost  
 if needs\_heuristic:  
 self.needs\_heuristic = True  
 self.generate\_heuristic()  
 self.evaluation\_function = self.heuristic + self.path\_cost  
 Puzzle.total\_states += 1  
  
 def \_\_str\_\_(self):  
 return str(self.state[0:3]) + '\n'+str(self.state[3:6]) + '\n'+str(self.state[6:9])  
  
 def generate\_heuristic(self):  
 self.heuristic = 0  
 for num in range(1, 9):  
 distance = abs(self.state.index(num) - self.goal\_state.index(num))  
 i = int(distance / 3)  
 j = int(distance % 3)  
 self.heuristic = self.heuristic + i + j  
  
 def goal\_test(self):  
 if self.state == self.goal\_state:  
 return True  
 return False  
  
 @staticmethod  
 def find\_available\_moves(i, j):  
 available\_move = ['U', 'D', 'L', 'R']  
 if i == 0:  
 available\_move.remove('U')  
 elif i == 2:  
 available\_move.remove('D')  
 if j == 0:  
 available\_move.remove('L')  
 elif j == 2:  
 available\_move.remove('R')  
 return available\_move  
  
 def generate\_child(self):  
 children = []  
 x = self.state.index(0)  
 i = int(x / 3)  
 j = int(x % 3)  
 available\_move = self.find\_available\_moves(i, j)  
  
 for move in available\_move:  
 new\_state = self.state.copy()  
 if move == 'U':  
 new\_state[x], new\_state[x-3] = new\_state[x-3], new\_state[x]  
 elif move == 'D':  
 new\_state[x], new\_state[x+3] = new\_state[x+3], new\_state[x]  
 elif move == 'L':  
 new\_state[x], new\_state[x-1] = new\_state[x-1], new\_state[x]  
 elif move == 'R':  
 new\_state[x], new\_state[x+1] = new\_state[x+1], new\_state[x]  
 children.append(Puzzle(new\_state, self, move, 1, self.needs\_heuristic))  
 return children  
  
 def find\_solution(self):  
 solution = []  
 solution.append(self.move)  
 path = self  
 while path.parent != None:  
 path = path.parent  
 solution.append(path.move)  
 solution = solution[:-1]  
 solution.reverse()  
 return solution

main.py

import random  
import numpy as np  
from BFS\_search import breadth\_first\_search  
from RBFS\_search import recursive\_best\_first\_search  
from puzzle import Puzzle  
  
def getInvCount(arr):  
 inv\_count = 0  
 empty\_value = 0  
 for i in range(0, 9):  
 for j in range(i + 1, 9):  
 if arr[j] != empty\_value and arr[i] != empty\_value and arr[i] > arr[j]:  
 inv\_count += 1  
 return inv\_count  
  
def isSolvable(puzzle):  
 inv\_count = getInvCount([j for sub in puzzle for j in sub])  
 return (inv\_count % 2 == 0)  
  
  
  
  
state = []  
false\_states = 0  
is\_solvable = False  
  
while not is\_solvable:  
 puzzle = random.sample(range(0, 9), 9)  
 false\_states += 1  
 puzzle\_array = np.array(puzzle).reshape(-1, 3)  
 is\_solvable = isSolvable(puzzle\_array)  
  
  
Puzzle.total\_states = 0  
#puzzle = [1, 2, 4, 3, 0, 5, 7, 6, 8]  
bfs = breadth\_first\_search(puzzle)  
print('BFS:', bfs)  
print('Total states:', Puzzle.total\_states)  
print('False states:', false\_states)  
print()  
  
Puzzle.total\_states = 0  
RBFS = recursive\_best\_first\_search(puzzle)  
print('RBFS:', RBFS)  
print('Total states:', Puzzle.total\_states)  
print()  
  
print('------------------------------------------')

### Приклади роботи

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

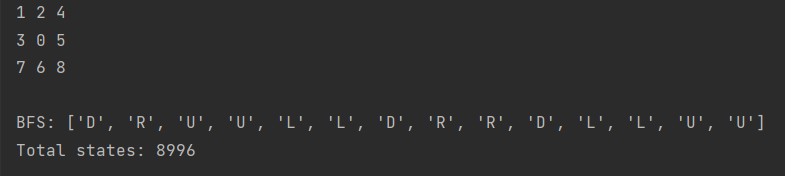


Рисунок 3.1 – Алгоритм BFS

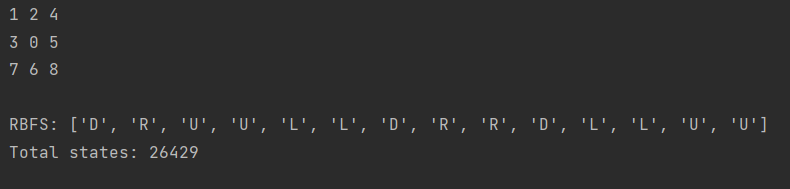


Рисунок 3.2 – Алгоритм RBFS

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму Назва алгоритму, задачі Назва задачі для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пам’яті |
| Стан 1 | 3295 | 3294 | 8996 | 5398 |
| Стан 2 | 2135 | 2134 | 5692 | 3384 |
| Стан 3 | 844 | 843 | 2281 | 1367 |
| Стан 4 | 1782 | 1781 | 4853 | 2930 |
| Стан 5 | 378 | 377 | 1029 | 625 |
| Стан 6 | 275 | 274 | 756 | 465 |
| Стан 7 | 450 | 449 | 1243 | 763 |
| Стан 8 | 1489 | 1488 | 3974 | 2370 |
| Стан 9 | 1039 | 1038 | 2756 | 1625 |
| Стан 10 | 3014 | 3013 | 8329 | 5036 |
| Стан 11 | 728 | 727 | 2007 | 1219 |
| Стан 12 | 1188 | 1187 | 3203 | 1911 |
| Стан 13 | 723 | 722 | 1992 | 1209 |
| Стан 14 | 327 | 326 | 874 | 525 |
| Стан 15 | 263 | 262 | 720 | 441 |
| Стан 16 | 328 | 327 | 877 | 527 |
| Стан 17 | 34 | 33 | 87 | 53 |
| Стан 18 | 145 | 144 | 378 | 225 |
| Стан 19 | 201 | 200 | 534 | 324 |
| Стан 20 | 378 | 377 | 1027 | 625 |

Таблиця 3.3 – Характеристики оцінювання алгоритму RBFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пам’яті |
| Стан 1 | 18659 | 9322 | 26429 | 80431 |
| Стан 2 | 1009 | 497 | 1352 | 3782 |
| Стан 3 | 123 | 55 | 192 | 588 |
| Стан 4 | 310 | 148 | 458 | 1404 |
| Стан 5 | 85 | 37 | 124 | 324 |
| Стан 6 | 12 | 1 | 28 | 78 |
| Стан 7 | 13 | 1 | 30 | 82 |
| Стан 8 | 520 | 253 | 706 | 1978 |
| Стан 9 | 541 | 264 | 734 | 2058 |
| Стан 10 | 1029 | 507 | 1489 | 4603 |
| Стан 11 | 36 | 12 | 62 | 172 |
| Стан 12 | 45 | 16 | 76 | 214 |
| Стан 13 | 200 | 94 | 278 | 772 |
| Стан 14 | 129 | 59 | 182 | 502 |
| Стан 15 | 34 | 12 | 57 | 159 |
| Стан 16 | 35 | 12 | 59 | 163 |
| Стан 17 | 7 | 0 | 18 | 52 |
| Стан 18 | 9 | 0 | 25 | 77 |
| Стан 19 | 10 | 0 | 28 | 86 |
| Стан 20 | 11 | 0 | 30 | 90 |

Висновок

* При виконанні даної лабораторної роботи було розглянуто роботу і принцип побудови двох алгоритмів: інформативного та неінформативного пошуку (BFS та RBFS), на основі задачі про 8 пазлів. Створено програмні реалізації цих алгоритмів, побудовано порівняльні таблиці ефективності , зображено псевдокоди та зображення виконання алгоритмів. При дослідженні, , найкраще себе показав алгоритм , адже . Дослідження проводились з початковими станами пазлів, які можливо розв’язати. Для інформативного пошуку використовувалася манхетенська відстань.