**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-15 Химич А.М.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Соколовський В.В.*

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 14](#_Toc81070692)

[3.2.1 Вихідний код 14](#_Toc81070693)

[3.2.2 Приклади роботи 20](#_Toc81070694)

[3.3 Дослідження алгоритмів 22](#_Toc81070695)

[Висновок 29](#_Toc81070696)

[Критерії оцінювання 30](#_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

Метод **BFS\_search()**:

Ініціалізувати чергу **q**

Додати в чергу **q** початковий стан **startState**

Поки **q** **НЕ** порожня

**node** = **q**.get()

ініціалізувати список **children[]**

**children** = **node**.generateChild()

Переглянути всі **child** в **children**

Якщо **child** – цільовий стан

Повернути **child**.findSolution()

**q**.put(**child**)

Метод **generateChild(state)**

ініціалізувати список **children[]**

ініціалізувати **zeroPos** значення індекса 0 в **state[]**

ініціалізувати **zeroX**  = цілою частиною (**zeroPos**/3)

ініціалізувати **zeroY** = залишком (**zeroPos**/3)

ініціалізувати **posibleMoves[]** = findposibleMoves(**zeroX**, **zeroY**)

Переглянути всі **move** в **posibleMoves[]**

**ChildState = state**

Якщо **move** == “UP”

Поміняти значеннями **childState[zeroPos] childState[zeroPos-3]**

Якщо **move** == “DOWN”

Поміняти значеннями **childState[zeroPos] childState[zeroPos+3]**

Якщо **move** == “LEFT”

Поміняти значеннями **childState[zeroPos] childState[zeroPos-1]**

Якщо **move** == “RIGHT”

Поміняти значеннями **childState[zeroPos] childState[zeroPos+1]**

**Children**.append(new-node(**childstate, move**))

Повернути children

Метод **findSoluito**n(node)

Ініціалізувати список **solutionLog[]**

**solutionLog**.append(**node.move**)

**path = node**

Поки **path.parent** НЕ NULL

**path = path.parent**

solutionLog.append(**path.move**)

Інвертувати **solutionLog**

Повернути **solutionlog**

Метод Recursive-Best-First-Search(**startState**)

Ініціалізувати **fLimit** великим константним значенням **maxConst**

**node** = RBFS\_search(**node(startState),fLimit**)

**node = node[0]**

повернути **node**.findSolution()

метод f**indPosibleMoves(zeroX,zeroY**)

ініціалізувати **posibleMoves** = [“UP”,”DOWN”,”LEFT”,”RIGHT”]

якщо **zeroX**==0

видалити “UP” з **possibleMoves**

якщо **zeroX**==2

видалити “DOWN” з **possibleMoves**

якщо **zeroY**==0

видалити “LEFT” з **possibleMoves**

якщо **zeroY**==0

видалити “RIGHT” з **possibleMoves**

повернути **posibleMoves**

метод **RBFS\_search**(**node,fLimit**)

ініціалізувати список **successors[]**

якщо **node** - цільовий стан

повернути **node,** NULL

ініціалізувати список **children[]**

**children** = generateChild()

Якщо **children** – порожній

Повернути NULL, **maxConst**

Ініціалізувати кількість нащадків **Count** = -1

Переглянути всі **child** в **children**

**Count = count +1**

**Successors**.append(**child.evaluationFunction,count,child**)

Поки **successors** НЕ порожній

Відсортувати список **Successors**

Ініціалізувати **bestNode** першим вузлом із **successors**

Якщо **bestNode**.evaluationFunction > **fLimit**

Повернути NULL, **bestNode.evaluationFunction**

Ініціалізувати **alternative** наступним після **bestNode** вузлом із **successors**

Ініціалізувати **min** мінімальним між **fLimit** та **alternative**

**result** та **bestNode.**evaluationFunction = значення, що поверне RBFS\_search(**bestNode,min**)

**successors[**0] = (**bestNode**.evaluationFunction, **successors**[0][1], **bestNode**)

якщо **result** не NULL

перервати

повернути **result**, NULL

метод **generateHeuristic(node**)

повторити поки **num**<9

**distance** = abs(state.index(**num**)-puzzleSolution.index(**num**))

горизонтальна відстань до потрібного місця **x** = ціла частина (**distance**/3)

вертикальна відстань до потрібного місця **у** = дробова частина (**distance**/3)

**node.heuristic = node.heuristic + x+ y**

повернути **node**

## Програмна реалізація

### Вихідний код

Main.py

from BFS import \*  
from RBFS import \*  
  
  
  
startState=[1,3,7,  
 5,0,2,  
 4,8,6]  
  
  
bfs=BFS\_search(startState)  
print('moves:', bfs)  
Puzzle.printSolutionLog(Puzzle,bfs,startState)  
  
time = datetime.now()  
mem = memory\_usage()  
  
rbfs=RBFS\_search(startState)  
print('moves:', rbfs)  
PuzzleWithHeuristic.printSolutionLog(PuzzleWithHeuristic,rbfs,startState)

Puzzle.py

from datetime import \*  
from memory\_profiler import memory\_usage  
  
class Puzzle:  
 puzzleSolution=[1,2,3,4,5,6,7,8,0]  
 time = datetime.now()  
 mem = memory\_usage()  
 movesCount=0  
 def \_\_init\_\_(self,state,parent,move,pathCost):  
 self.parent=parent  
 self.state=state  
 self.move=move  
 if parent:  
 self.pathCost = parent.pathCost + pathCost  
 else:  
 self.pathCost = pathCost  
 Puzzle.movesCount+=1  
  
 def isSolved(self):  
 if self.state == self.puzzleSolution:  
 return True  
 return False  
  
 def findPosibleMoves(self,x, y):  
 posibleMoves = ['Up', 'Down', 'Left', 'Right']  
 if x == 0:  
 posibleMoves.remove('Up')  
 elif x == 2:  
 posibleMoves.remove('Down')  
 if y == 0:  
 posibleMoves.remove('Left')  
 elif y == 2:  
 posibleMoves.remove('Right')  
 return posibleMoves  
  
 def generateChild(self):  
 children = []  
 zeroPos = self.state.index(0)  
 zeroX = int(zeroPos / 3)  
 zeroY = int(zeroPos % 3)  
  
 posibleMoves = self.findPosibleMoves(zeroX, zeroY)  
  
 for move in posibleMoves:  
 childState = self.state.copy()  
 if move == 'Up':  
 childState[zeroPos], childState[zeroPos - 3] = childState[zeroPos - 3],childState[zeroPos]  
 elif move == 'Down':  
 childState[zeroPos], childState[zeroPos + 3] = childState[zeroPos + 3],childState[zeroPos]  
 elif move == 'Left':  
 childState[zeroPos], childState[zeroPos - 1] = childState[zeroPos - 1],childState[zeroPos]  
 elif move == 'Right':  
 childState[zeroPos], childState[zeroPos + 1] = childState[zeroPos + 1],childState[zeroPos]  
 children.append(Puzzle(childState, self,move, 1))  
 return children  
  
 def findSolution(self):  
 solutionLog = []  
 solutionLog.append(self.move)  
 path = self  
 while path.parent != None:  
 path = path.parent  
 solutionLog.append(path.move)  
 solutionLog = solutionLog[:-1]  
 solutionLog.reverse()  
 return solutionLog  
 def printSolutionLog(self,solutionLog, startState):  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")  
  
 for move in solutionLog:  
 zeroPos = startState.index(0)  
 if move == 'Up':  
 startState[zeroPos], startState[zeroPos - 3] = startState[zeroPos - 3], startState[zeroPos]  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")  
 elif move == 'Down':  
 startState[zeroPos], startState[zeroPos + 3] = startState[zeroPos + 3], startState[zeroPos]  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")  
 elif move == 'Left':  
 startState[zeroPos], startState[zeroPos - 1] = startState[zeroPos - 1], startState[zeroPos]  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")  
 elif move == 'Right':  
 startState[zeroPos], startState[zeroPos + 1] = startState[zeroPos + 1], startState[zeroPos]  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")

BFS.py

import sys  
from queue import Queue  
from Puzzle import Puzzle  
from datetime import \*  
  
  
def BFS\_search(initialState):  
 startNode = Puzzle(initialState, None, None, 0)  
 if startNode.isSolved():  
 return startNode.findSolution()  
 q = Queue()  
 q.put(startNode)  
 visited = []  
  
 while not(q.empty()):  
 if (Puzzle.mem[0] > 1024 \* 1024 \* 1024 or (datetime.now() - Puzzle.time).seconds > 1800):  
 print("no solution")  
 sys.exit()  
  
 node=q.get()  
 visited.append(node.state)  
 children=node.generateChild()  
 for child in children:  
 if child.isSolved():  
 return child.findSolution()  
 q.put(child)  
 return

PuzzleWithHeuristics.py

from sys import maxsize  
from datetime import \*  
from memory\_profiler import memory\_usage  
  
class PuzzleWithHeuristic():  
 time = datetime.now()  
 mem = memory\_usage()  
 puzzleSolution = [1,2,3,4,5,6,7,8,0]  
 visited=[]  
 movesCount = 0  
 heuristic = None  
 evaluationFunction = None  
 def \_\_init\_\_(self,state,parent,move,pathCost):  
 self.parent = parent  
 self.state = state  
 self.move = move  
 if parent:  
 self.pathCost = parent.pathCost + pathCost  
 else:  
 self.pathCost = pathCost  
  
 self.generateHeuristic()  
 self.evaluationFunction = self.heuristic + self.pathCost  
  
 def generateHeuristic(self):  
 self.heuristic = 0  
 for num in range(1, 9):  
 distance = abs(self.state.index(num) - self.puzzleSolution.index(num))  
 x = int(distance / 3)  
 y = int(distance % 3)  
 self.heuristic = self.heuristic + x + y  
  
  
 def isSolved(self):  
 if self.state == self.puzzleSolution:  
 return True  
 return False  
  
 def findPosibleMoves(self,x, y):  
 posibleMoves = ['Up', 'Down', 'Left', 'Right']  
 if x == 0:  
 posibleMoves.remove('Up')  
 elif x == 2:  
 posibleMoves.remove('Down')  
 if y == 0:  
 posibleMoves.remove('Left')  
 elif y == 2:  
 posibleMoves.remove('Right')  
 return posibleMoves  
  
 def generateChild(self):  
 children = []  
 zeroPos = self.state.index(0)  
 zeroX = int(zeroPos / 3)  
 zeroY = int(zeroPos % 3)  
  
 posibleMoves = self.findPosibleMoves(zeroX, zeroY)  
  
 for move in posibleMoves:  
 childState = self.state.copy()  
 if move == 'Up':  
 childState[zeroPos], childState[zeroPos - 3] = childState[zeroPos - 3],childState[zeroPos]  
 elif move == 'Down':  
 childState[zeroPos], childState[zeroPos + 3] = childState[zeroPos + 3],childState[zeroPos]  
 elif move == 'Left':  
 childState[zeroPos], childState[zeroPos - 1] = childState[zeroPos - 1],childState[zeroPos]  
 elif move == 'Right':  
 childState[zeroPos], childState[zeroPos + 1] = childState[zeroPos + 1],childState[zeroPos]  
 children.append(PuzzleWithHeuristic(childState, self,move, 1))  
 return children  
  
 def findSolution(self):  
 solutionLog = []  
 solutionLog.append(self.move)  
 path = self  
 while path.parent != None:  
 path = path.parent  
 solutionLog.append(path.move)  
 solutionLog = solutionLog[:-1]  
 solutionLog.reverse()  
 return solutionLog  
  
  
 def printSolutionLog(self,solutionLog, startState):  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")  
  
 for move in solutionLog:  
 zeroPos = startState.index(0)  
 if move == 'Up':  
 startState[zeroPos], startState[zeroPos - 3] = startState[zeroPos - 3], startState[zeroPos]  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")  
 elif move == 'Down':  
 startState[zeroPos], startState[zeroPos + 3] = startState[zeroPos + 3], startState[zeroPos]  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")  
 elif move == 'Left':  
 startState[zeroPos], startState[zeroPos - 1] = startState[zeroPos - 1], startState[zeroPos]  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")  
 elif move == 'Right':  
 startState[zeroPos], startState[zeroPos + 1] = startState[zeroPos + 1], startState[zeroPos]  
 print(str(startState[0:3]) + '\n' + str(startState[3:6]) + '\n' + str(startState[6:9]))  
 print("\t|\n\tV")

RBFS.py

from PuzzleWithHeuristic import \*  
import sys  
from datetime import \*  
def RBFS\_search(initial\_state):  
  
 node=search(PuzzleWithHeuristic(state=initial\_state, parent=None, move=None, pathCost=0),fLimit=maxsize)  
 node=node[0]  
 return node.findSolution()  
  
def search(node, fLimit):  
 successors=[]  
 if (PuzzleWithHeuristic.mem[0] > 1024\*1024\*1024 or (datetime.now()-PuzzleWithHeuristic.time).seconds > 1800):  
 print("no solution")  
 sys.exit()  
  
 if node.isSolved():  
 return node,None  
 children=node.generateChild()  
 if not len(children):  
 return None, maxsize  
 count=-1  
 for child in children:  
 if child not in PuzzleWithHeuristic.visited:  
 PuzzleWithHeuristic.visited.append(child)  
 count+=1  
 successors.append((child.evaluationFunction,count,child))  
  
  
 while len(successors):  
 PuzzleWithHeuristic.movesCount += 1  
 successors.sort()  
 bestNode=successors[0][2]  
 if bestNode.evaluationFunction > fLimit:  
 return None, bestNode.evaluationFunction  
 alternative=successors[1][0]  
 result,bestNode.evaluationFunction=search(bestNode,min(fLimit,alternative))  
 successors[0]=(bestNode.evaluationFunction,successors[0][1],bestNode)  
  
 if result is not None:  
 break  
  
 return result,None

### Приклади роботи

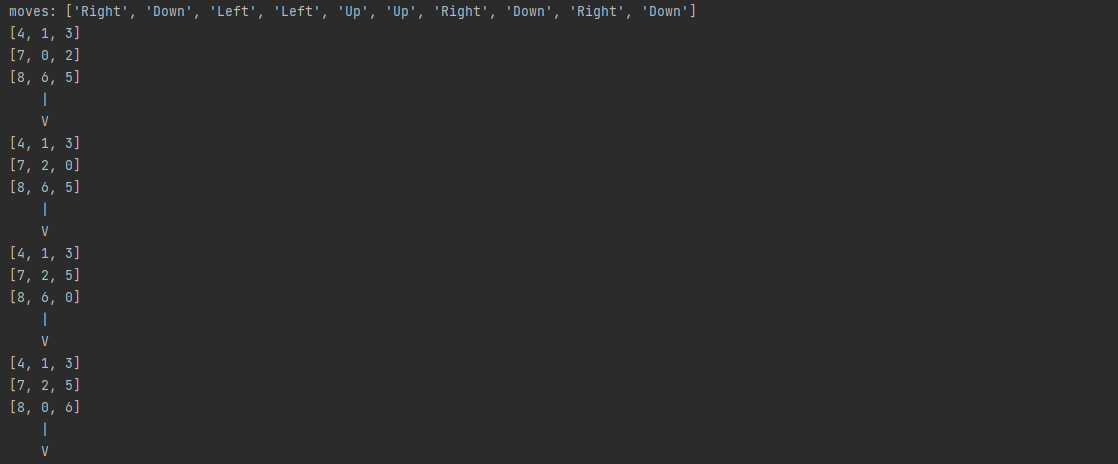
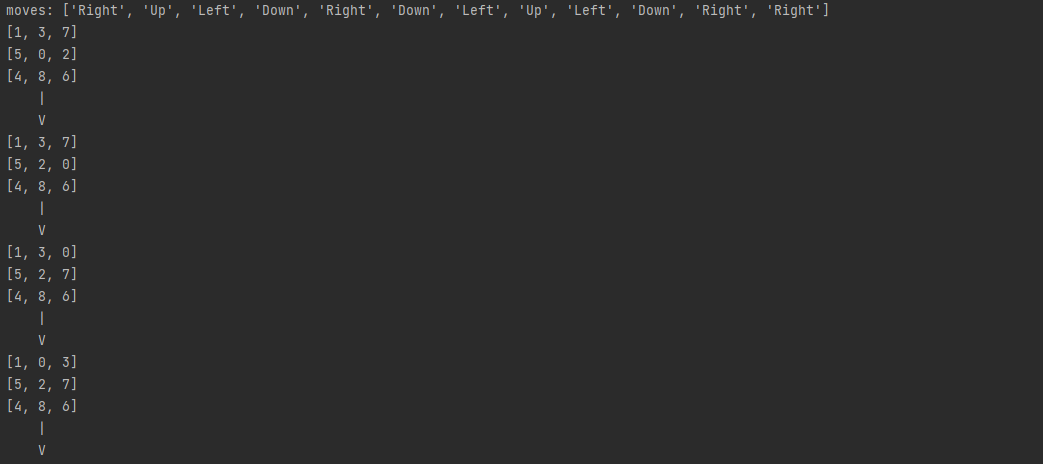
На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.



Рисунок 3.1 – Алгоритм BFS для стану (4 1 3 7 0 2 8 6 5)

Рисунок 3.2 –Алгоритм RBFS стану(1 3 7 5 0 2 4 8 6)

## Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання BFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього унікальних станів | Всього станів у пам’яті |
| 1 2 7  3 4 5  6 0 8 | 127407 | 44029 | 1485 | 127407 |
| 1 3 7  5 0 2  4 8 6 | 518413 | 179045 | 3540 | 518413 |
| 1 8 2  0 4 3  7 6 5 | 13816 | 9014 | 444 | 13816 |
| 1 3 5  7 0 2  8 4 6 | 45557 | 29590 | 724 | 45557 |
| 1 6 0  7 3 2  5 4 8 | 162427 | 105163 | 1440 | 162427 |
| 3 5 0  1 4 8  7 6 2 | 2094765 | 723221 | 20142 | 2094765 |
| 1 5 2  8 7 3  0 4 6 | 21438 | 13900 | 551 | 21438 |
| 1 3 0  4 6 5  7 2 8 | 217416 | 14102 | 561 | 217416 |
| 1 6 2  4 5 3  7 8 0 | 330778 | 115112 | 655 | 330778 |
| 2 4 0  5 3 1  7 8 6 | 1436397 | 503765 | 9354 | 1436397 |
| 1 3 0  8 2 5  4 7 6 | 3613 | 2358 | 249 | 3613 |
| 2 3 6  4 1 5  0 7 8 | 23925 | 15559 | 583 | 23925 |
| 4 1 3  7 0 8  5 6 2 | 386479 | 251411 | 2056 | 386479 |
| 2 3 0  1 5 7  4 8 6 | 242869 | 158791 | 1887 | 242869 |
| 0 5 3  4 1 6  7 2 8 | 219973 | 143527 | 1658 | 219973 |
| 4 1 3  2 8 5  0 7 6 | 3565 | 2306 | 242 | 3565 |
| 0 5 3  2 1 7  4 8 6 | 193717 | 126023 | 1607 | 193717 |
| 1 3 0  8 2 7  4 6 5 | 243229 | 159030 | 1905 | 243229 |
| 2 3 6  1 5 8  0 4 7 | 34677 | 22727 | 719 | 34677 |
| 4 1 3  7 0 2  8 6 5 | 68021 | 44566 | 991 | 68021 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього унікальних станів | Всього станів у пам’яті |
| 1 2 7  3 4 5  6 0 8 | 118 | 79 | 81 | 80 |
| 1 3 7  5 0 2  4 8 6 | 2776 | 35 | 1998 | 36 |
| 1 8 2  0 4 3  7 6 5 | 27 | 8 | 9 | 9 |
| 1 3 5  7 0 2  8 4 6 | 31 | 19 | 10 | 20 |
| 1 6 0  7 3 2  5 4 8 | 36 | 35 | 24 | 36 |
| 3 5 0  1 4 8  7 6 2 | 78 | 54 | 42 | 56 |
| 1 5 2  8 7 3  0 4 6 | 39 | 18 | 21 | 18 |
| 1 3 0  4 6 5  7 2 8 | 62 | 33 | 28 | 34 |
| 1 6 2  4 5 3  7 8 0 | 41 | 21 | 30 | 22 |
| 2 4 0  5 3 1  7 8 6 | 3167 | 2381 | 1654 | 2382 |
| 1 3 0  8 2 5  4 7 6 | 32 | 11 | 24 | 12 |
| 2 3 6  4 1 5  0 7 8 | 86 | 57 | 56 | 58 |
| 4 1 3  7 0 8  5 6 2 | 46 | 15 | 20 | 16 |
| 2 3 0  1 5 7  4 8 6 | 1700 | 1285 | 1286 | 1286 |
| 0 5 3  4 1 6  7 2 8 | 2330 | 1675 | 1104 | 1676 |
| 4 1 3  2 8 5  0 7 6 | 45 | 21 | 30 | 22 |
| 0 5 3  2 1 7  4 8 6 | 1751 | 1283 | 1294 | 1284 |
| 1 3 0  8 2 7  4 6 5 | 257 | 167 | 93 | 168 |
| 2 3 6  1 5 8  0 4 7 | 25 | 9 | 10 | 10 |
| 4 1 3  7 0 2  8 6 5 | 44 | 21 | 22 | 22 |

Таблиця 3.2 – Характеристики оцінювання RBFS

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритми інформативного та неінформативного пошуку, а саме BFS та RBFS для вирішення задачі 8puzzle. Було розроблено програмні специфікації для імплементації даних алгоритмів. За результатами тестування вищезгаданих алгоритмів на 20 різних вхідних станах можна зробити висновок:

Алгоритми інформативного пошуку мають переваги у всіх критеріях оцінювання, так як в загальному випадку генерують менше станів, а отже потребують менше пам’яті, та не можуть ввійти в цикл

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%.