**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-12 Сімчук Андрій Володимирович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов Олексій Олександрович*

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** |
| 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

3.1.1 LDFS

start\_node = the initial state of the problem

iterations = 1

**if** (start\_node == goal) **do**

**return** True, actions taken to achieve the goal, 1

**end if**

q = queue

**add** start\_node **to the beginning of** q

explored = empty list

iterations = 0

**while not** (q is empty) **do**

iterations += 1

node = **get element from end of q**

**add** state **of** node **to the end of** explored

**if** (depth **of** node == limit) **do**

**return** False, None, **length of** explored

**end if**

children = **generate children of** node

**for** (child **in** children[::-1]) **do**

**if** (state **of** child **not in** explored) **do**

 **if** (child == goal) **do**

**return** True, actions taken to achieve the goal, **length of** e explored

**end if**

**add** child **to the beginning of** q

**end if**

**end for**

**end while**

**return** False, None, **length of** explored

3.1.2 RBFS

**function** **\_rbfs**(initial\_state) **do**

node = **RBFS\_search**(Puzzle(state=initial\_state, parent=None, action=None, path\_cost=0, needs\_hueristic=True), f\_limit=maxsize)

node = **zero element of** node

**return** actions taken to achieve the goal

**end function**

**function RBFS\_search**(node, f\_limit) **do**

iterations += 1

successors = empty list

result = None

**if** (node is goal) **do**

**return** node, None

**end if**

children = **generate children of** node

**if not** (**length** **of** children):

**return** None, maxsize

**end if**

count = -1

**for** (child **in** children) **do**

count += 1

 **add list**(evaluation\_function **of** child, count, child) **to end of** \_ successors

**end for**

states\_in\_memory **= length of** successors

**while** (**length** **of** successors):

**sort** successors

best\_node = successors[0][2]

**if** (evaluation\_function **of** best\_node > f\_limit) **do**

**return** None, evaluation\_function **of** best\_node

**end if**

 alternative = successors[1][0]

result, best\_node.evaluation\_function = **RBFS\_search**(best\_node, min **min**(f\_limit, alternative))

 successors[0] = (best\_node.evaluation\_function, successors[0][1], b best\_node)

**if** (result != None) **do**

**break**

**end if**

**end while**

**return** result, None

## Програмна реалізація

### Вихідний код

3.2.1.1 main.py

from time import time

from LDFS import ldfs

from RBFS import rbfs

from puzzle import Puzzle

def main():

state = [[0, 1, 2,

3, 4, 5,

6, 7, 8],

[3, 1, 2,

4, 7, 5,

0, 6, 8],

[6, 3, 2,

0, 1, 5,

7, 4, 8],

[1, 0, 5,

3, 2, 4,

6, 7, 8],

[1, 4, 2,

3, 7, 5,

6, 0, 8],

[3, 1, 0,

4, 5, 2,

6, 7, 8],

[3, 1, 2,

6, 4, 0,

7, 8, 5],

[1, 4, 2,

3, 0, 5,

6, 7, 8],

[1, 4, 2,

6, 3, 5,

0, 7, 8],

[1, 2, 5,

6, 3, 4,

7, 8, 0],

[3, 2, 5,

4, 7, 1,

0, 6, 8],

[1, 2, 5,

3, 0, 8,

6, 4, 7],

[3, 1, 0,

6, 4, 2,

7, 8, 5],

[3, 1, 2,

0, 4, 5,

6, 7, 8],

[3, 0, 2,

6, 1, 5,

7, 4, 8],

[0, 4, 2,

1, 7, 5,

3, 6, 8],

[3, 1, 0,

4, 5, 2,

6, 7, 8],

[3, 1, 2,

6, 4, 5,

7, 0, 8],

[1, 2, 5,

3, 4, 8,

6, 7, 0],

[3, 0, 1,

4, 7, 2,

6, 8, 5]]

limit = 100000

for i in range(20):

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

print('Test number:', i + 1)

print()

try:

Puzzle.num\_of\_states = 0

Puzzle.iterations = 0

t0 = time()

flag, LDFS, states\_in\_memory = ldfs(state[i], limit)

t1 = time()

t = t1 - t0

if not flag:

print("Couldn't find solution")

print('Iterations:', Puzzle.iterations)

print('Number of states:', Puzzle.num\_of\_states)

print('States stored in memory:', states\_in\_memory)

print('Time:', t)

print()

else:

print('LDFS:', LDFS)

print('Iterations:', Puzzle.iterations)

print('Number of states:', Puzzle.num\_of\_states)

print('States stored in memory:', states\_in\_memory)

print('Time:', t)

print()

except:

print("Timeout")

print('Iterations:', Puzzle.iterations)

print('Number of states:', Puzzle.num\_of\_states)

print('States stored in memory:', states\_in\_memory)

print('Time:', t)

print()

try:

Puzzle.num\_of\_states = 0

Puzzle.iterations = 0

t0 = time()

RBFS = rbfs(state[i])

t1 = time()

t = t1 - t0

print('RBFS:', RBFS)

print('Iterations:', Puzzle.iterations)

print('Number of states:', Puzzle.num\_of\_states)

print('States stored in memory:', Puzzle.states\_in\_memory)

print('Time:', t)

print()

except:

print("Timeout")

print('Iterations:', Puzzle.iterations)

print('Number of states:', Puzzle.num\_of\_states)

print('States stored in memory:', Puzzle.states\_in\_memory)

print('Time:', t)

print()

if \_\_name\_\_ == "\_\_main\_\_":

main()

3.2.1.2 LDFS.py

from func\_timeout import func\_timeout

from queue import Queue

from puzzle import Puzzle

def ldfs(initial\_state, limit):

return func\_timeout(30 \* 60, \_ldfs, args=[initial\_state, limit])

def \_ldfs(initial\_state, limit):

start\_node = Puzzle(initial\_state, None, None, 0)

Puzzle.iterations = 1

if start\_node.goal\_test():

return True, start\_node.find\_solution(), 1

q = Queue()

q.put(start\_node)

explored = []

Puzzle.iterations = 0

while not (q.empty()):

Puzzle.iterations += 1

node = q.get()

explored.append(node.state)

if node.depth == limit:

return False, None, len(explored)

children = node.generate\_child()

for child in children[::-1]:

if child.state not in explored:

if child.goal\_test():

return True, child.find\_solution(), len(explored)

q.put(child)

return False, None, len(explored)

3.2.1.3 RBFS.py

from func\_timeout import func\_timeout

from sys import maxsize

from puzzle import Puzzle

def rbfs(initial\_state):

return func\_timeout(30 \* 60, \_rbfs, args=[initial\_state])

def \_rbfs(initial\_state):

node = RBFS\_search(Puzzle(state=initial\_state, parent=None, action=None, path\_cost=0, needs\_hueristic=True),

f\_limit=maxsize)

node = node[0]

return node.find\_solution()

def RBFS\_search(node, f\_limit):

Puzzle.iterations += 1

successors = []

result = None

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

successors.append((child.evaluation\_function, count, child))

Puzzle.states\_in\_memory = len(successors)

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

3.2.1.4 puzzle.py

class Puzzle:

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

heuristic = None

evaluation\_function = None

needs\_hueristic = False

num\_of\_states = 0

states\_in\_memory = 1

iterations = 0

def \_\_init\_\_(self, state, parent, action, path\_cost, needs\_hueristic=False):

self.parent = parent

self.state = state

self.action = action

if not parent:

self.depth = 0

if parent:

self.path\_cost = parent.path\_cost + path\_cost

self.depth = parent.depth + 1

else:

self.path\_cost = path\_cost

if needs\_hueristic:

self.needs\_hueristic = True

self.generate\_heuristic()

self.evaluation\_function = self.heuristic + self.path\_cost

Puzzle.num\_of\_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):

if self.state.index(num) != self.goal\_state.index(num):

self.heuristic = self.heuristic + 1

return

def goal\_test(self):

if self.state == self.goal\_state:

return True

return False

@staticmethod

def find\_legal\_actions(i, j):

legal\_action = ['U', 'D', 'L', 'R']

if i == 0:

legal\_action.remove('U')

elif i == 2:

legal\_action.remove('D')

if j == 0:

legal\_action.remove('L')

elif j == 2:

legal\_action.remove('R')

return legal\_action

def generate\_child(self):

children = []

x = self.state.index(0)

i = int(x / 3)

j = int(x % 3)

legal\_actions = self.find\_legal\_actions(i, j)

for action in legal\_actions:

new\_state = self.state.copy()

if action is 'U':

new\_state[x], new\_state[x - 3] = new\_state[x - 3], new\_state[x]

elif action is 'D':

new\_state[x], new\_state[x + 3] = new\_state[x + 3], new\_state[x]

elif action is 'L':

new\_state[x], new\_state[x - 1] = new\_state[x - 1], new\_state[x]

elif action is 'R':

new\_state[x], new\_state[x + 1] = new\_state[x + 1], new\_state[x]

children.append(Puzzle(new\_state, self, action, 1, self.needs\_hueristic))

self.depth += 1

return children

def find\_solution(self):

solution = []

solution.append(self.action)

path = self

while path.parent != None:

path = path.parent

solution.append(path.action)

solution = solution[:-1]

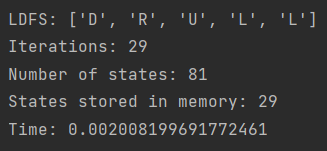
solution.reverse()

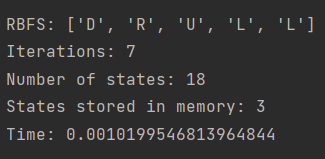
return solution

### Приклади роботи

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

Рисунок 3.1 – Алгоритм «Пошук вглиб з обмеженням глибини»



Рисунок 3.2 – Алгоритм «Рекурсивний пошук за першим найкращим співпадінням»

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму «Пошук вглиб з обмеженням глибини», задачі «8-puzzle» для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму «Пошук вглиб з обмеженням глибини»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
| Стан 1 | 1 | - | 1 | 1 |
| Стан 2 | 10 | - | 30 | 10 |
| Стан 3 | 105 | - | 297 | 105 |
| Стан 4 | 29 | - | 81 | 29 |
| Стан 5 | 9 | - | 27 | 9 |
| Стан 6 | 12 | - | 36 | 12 |
| Стан 7 | 26 | - | 72 | 26 |
| Стан 8 | 5 | - | 17 | 5 |
| Стан 9 | 14 | - | 42 | 14 |
| Стан 10 | 93 | - | 255 | 93 |
| Стан 11 | 108 | - | 300 | 108 |
| Стан 12 | 57 | - | 165 | 57 |
| Стан 13 | 48 | - | 132 | 48 |
| Стан 14 | 1 | - | 4 | 1 |
| Стан 15 | 33 | - | 93 | 33 |
| Стан 16 | 51 | - | 141 | 51 |
| Стан 17 | 12 | - | 36 | 12 |
| Стан 18 | 6 | - | 18 | 6 |
| Стан 19 | 15 | - | 45 | 15 |
| Стан 20 | 70 | - | 192 | 70 |

1) Середня кількість етапів , які знадобилось для досягнення розв’язку (ітерації): 35;

2) Середня кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе: 0;

3) Середня кількість згенерованих станів під час пошуку: 99;

4) Середня кількість станів, що зберігаються в пам’яті під час роботи програми: 35;

В таблиці 3.2 наведені характеристики оцінювання алгоритму «Рекурсивний пошук за першим найкращим співпадінням», задачі «8-puzzle» для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання алгоритму «Рекурсивний пошук за першим найкращим співпадінням»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
| Стан 1 | 1 | - | 1 | 1 |
| Стан 2 | 5 | - | 13 | 3 |
| Стан 3 | 17 | - | 46 | 3 |
| Стан 4 | 7 | - | 18 | 3 |
| Стан 5 | 4 | - | 11 | 3 |
| Стан 6 | 5 | - | 13 | 3 |
| Стан 7 | 6 | - | 14 | 3 |
| Стан 8 | 3 | - | 8 | 3 |
| Стан 9 | 5 | - | 13 | 3 |
| Стан 10 | 9 | - | 23 | 3 |
| Стан 11 | 13 | - | 36 | 3 |
| Стан 12 | 7 | - | 18 | 3 |
| Стан 13 | 7 | - | 16 | 3 |
| Стан 14 | 2 | - | 4 | 3 |
| Стан 15 | 6 | - | 16 | 3 |
| Стан 16 | 7 | - | 18 | 3 |
| Стан 17 | 5 | - | 13 | 3 |
| Стан 18 | 4 | - | 9 | 3 |
| Стан 19 | 5 | - | 11 | 3 |
| Стан 20 | 8 | - | 21 | 3 |

1) Середня кількість етапів , які знадобилось для досягнення розв’язку (ітерації): 6;

2) Середня кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе: 0;

3) Середня кількість згенерованих станів під час пошуку: 16;

4) Середня кількість станів, що зберігаються в пам’яті під час роботи програми: 3;

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритми неінформативного (Пошук вглиб з обмеженням глибини), інформативного та локального пошуку (Рекурсивний пошук за першим найкращим співпадінням), було проведено порівняльний аналіз ефективності використання алгоритмів. «Рекурсивний пошук за першим найкращим співпадінням» виявився ефективнішим за «Пошук вглиб з обмеженням глибини» для заданої задачі «8-puzzle» у всіх заданих початкових станах.

Критерії оцінювання

За умови здачі лабораторної роботи до 30.10.2022 включно максимальний бал дорівнює – 5. Після 30.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 10%;
* програмна реалізація алгоритму – 60%;
* дослідження алгоритмів – 25%;
* висновок – 5%.