МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ»

Лабораторна робота № 2

з дисципліни «Системи штучного інтелекту»

на тему:

«Класичні методи пошуку рішень у просторах станів»

Перевірив

Коломоєць С.О.

Виконали

студентки ФІОТ

групи ІС-12

Павлова Софія

Гоголь Софія

Київ 2023

Лабораторна робота №2

Тема: Класичні методи пошуку рішень у просторах станів

Мета роботи:

Розглянути та дослідити алгоритми неінформативного та інформативного пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

Завдання

1) Реалізувати програму, яка розв’язує поставлену задачу Task за допомогою алгоритму неінформативного пошуку AlgNoInf та алгоритму інформативного пошуку AlgInf, що використовує задану евристичну функцію Func.
2) 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:

   - середній час пошуку рішення у секундах
   - середні витрати по пам’яті
   - середню кількість згенерованих станів під час пошуку

Для реалізації завдання маємо 4 варіант:

4	8-puzzle	LDFS	RBFS	H1

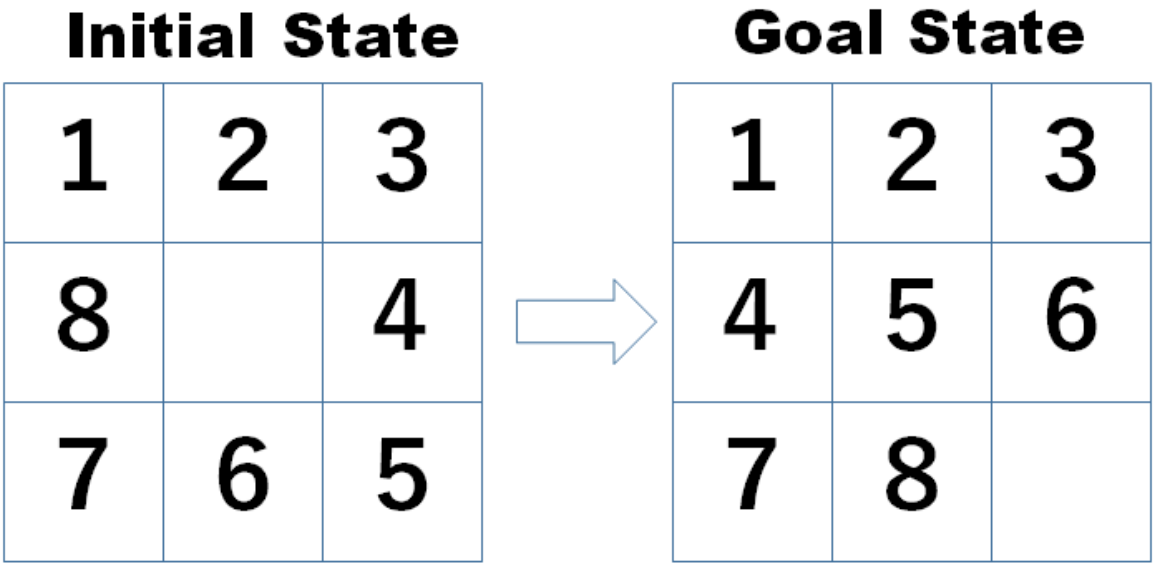
Хід роботи

1. **Огляд варіанту**

Перед початком роботи ознайомимося з поданими варіантом проблеми та алгоритмів для її рішення.

8-puzzle - це головоломка, яка полягає в тому, щоб переміщати вісімки з числами в межах квадратної дошки 3x3, залишаючи одну клітинку порожньою. Мета полягає в тому, щоб досягти певної кінцевої конфігурації пазлу, яка зазвичай має вигляд відсортованих чисел від 1 до 8 в порядку, а порожню клітинку можна розміщувати в різних частинах дошки.

Суть проблеми полягає в тому, що, починаючи з початкового стану пазлу і виконуючи допустимі ходи, потрібно знайти найкоротший шлях для досягнення цільового стану.



LDFS (Limited Depth First Search) -- це алгоритм пошуку, який відноситься до неінформативних або сліпих алгоритмів пошуку, оскільки не використовує додаткової інформації про структуру задачі, окрім самого початкового стану і правил переходу між станами.

Основна ідея LDFS полягає в тому, що він використовує глибинний пошук, але з обмеженням глибини. Тобто, він розглядає різні глибини пошуку і перевіряє всі можливі ходи на кожному рівні глибини, поки не досягне максимальної глибини або не знайде розв'язку.

RBFS (Recursive Best-First Search) – це алгоритм інформативного пошуку, який використовує інформацію про структуру задачі для більш ефективного пошуку.

Основна ідея RBFS полягає в тому, щоб використовувати інформацію про структуру задачі для вибору найперспективніших гілок пошуку, що зазвичай призводить до більш швидкого знаходження розв'язку.

2. Реалізації алгоритмів для розв’язання задачі

Реалізуємо алгоритм LDFS найпростішим чином без додаткових модифікацій.

Лістинг коду:

```
def limited_deep_first_search(initial_state, max_depth):
    start_node = Puzzle(initial_state, None, None, 0)
    stack = LifoQueue()
    stack.put((start_node, 0))  # Починаємо з кореневого вузла
і глибини 0
    while not stack.empty():
        node, depth = stack.get()
        if depth > max_depth:
            continue  # Пропускаємо вузли глибше за max_depth
        if node.goal_test():
            return node.find_solution()
        if depth < max_depth:
            children = node.generate_child()
            for child in children:
                stack.put((child, depth + 1))
    return None  # Розв'язок не знайдено
```

Реалізуємо алгоритм RDFS:

Лістинг коду для функції **recursive_best_first_search**

```
def recursive_best_first_search(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()
```

Лістинг коду для функції **RBFS_search**

```
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
       successors.append((child.evaluation_function, count,
child))
   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
```

Далі реалізуємо саму головоломку.

Для цього опишемо клас із поточним станом та оцінками різних аспектів.

```
class Puzzle:
    goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5] #цільовий стан, де 0 -
порожня клітинка
    heuristic = None
    evaluation_function = None
    needs_hueristic = False
    num_of_instances = 0

    # --------- Ініціалізація об'єкту головоломки ---------
    def __init__(self, state, parent, action, path_cost,
needs_hueristic = False):
        self.parent = parent
        self.state = state
        self.action = action
        if parent:
            self.path_cost = parent.path_cost + path_cost
        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_instances += 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 = sum(1 for i in range(9) if self.state[i]
!= self.goal_state[i])

    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:  # up is disable
            legal_action.remove('U')
        elif i == 2:  # down is disable
            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 == 'U':
                new_state[x], new_state[x-3] = new_state[x-3],
new_state[x]
            elif action == 'D':
                new_state[x], new_state[x+3] = new_state[x+3],
new_state[x]
            elif action == 'L':
                new_state[x], new_state[x-1] = new_state[x-1],
new_state[x]
            elif action == '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))
        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
```

3. **Проведення експериментів**

Для вивчення ефективності роботи алгоритмів проведемо серію з 20 експериментів, що відрізняються початковим станом.

За результатами визначимо:

- середній час пошуку рішення у секундах
- середні витрати по пам’яті
- середню кількість згенерованих станів під час пошуку

```
import pandas as pd
from time import time
from ldfs import limited_deep_first_search
from rbfs import recursive_best_first_search
from puzzle import Puzzle

# --------- Функція табличного представлення поточного стану
def print_state(state, state_number=None):
    if state_number is not None:
        print(f'Стан {state_number}:')
    for i in range(0, len(state), 3):
```

```
        row = state[i:i + 3]
        print(row)

state = [[1, 3, 4,
          8, 6, 2,
          7, 0, 5],

         [1, 3, 4,
          0, 6, 2,
          8, 7, 5],

         [2, 8, 1,
          0, 4, 3,
          7, 6, 5],

         [2, 1, 0,
          4, 8, 3,
          7, 6, 5],

         [2, 8, 1,
          4, 6, 3,
          0, 7, 5],

         [2, 8, 1,
          4, 0, 6,
          7, 5, 3],

         [1, 8, 4,
          6, 7, 2,
          5, 3, 0],

         [0, 1, 4,
          6, 8, 2,
          5, 7, 3],

         [1, 0, 6,
          8, 3, 5,
          7, 4, 2],

         [1, 3, 6,
          7, 8, 5,
          4, 2, 0],

         [1, 6, 2,
          7, 4, 0,
          5, 3, 8],

         [1, 6, 2,
          7, 0, 8,
          5, 4, 3],

         [8, 7, 1,
          6, 0, 3,
          2, 5, 4],

         [8, 7, 1,
          6, 3, 4,
          2, 5, 0],

         [5, 1, 3,
```

```
         2, 7, 6,
         0, 8, 4],

        [5, 1, 3,
         2, 0, 6,
         8, 7, 4],

        [4, 2, 6,
         1, 5, 0,
         8, 7, 3],

        [4, 2, 6,
         1, 0, 5,
         8, 7, 3],

        [1, 2, 3,
         5, 6, 4,
         0, 8, 7],

        [1, 2, 3,
         6, 4, 0,
         5, 8, 7]]

data = pd.DataFrame(columns=['Алгоритм', 'Час', 'Операцій',
'Пам\'ять'])

sum_memo_ldfs = 0
sum_memo_rbfs = 0
sum_lens_ldfs = 0
sum_lens_rbfs = 0
sum_time_ldfs = 0
sum_time_rbfs = 0


for i in range(0, 20):
   print('-----------------------------------------')
   print('Стартовий стан', i + 1)
   print_state(state[i])

   Puzzle.num_of_instances = 0
   t0 = time()
   max_depth = 20
   ldfs = limited_deep_first_search(state[i], max_depth)
   t1 = time() - t0
   print('\nLDFS:', ldfs)
   print('Операцій:', len(ldfs))
   print('Пам\'ять:', Puzzle.num_of_instances)
   print('Час:', t1)

sum_time_ldfs = sum_time_ldfs + t1
sum_lens_ldfs = sum_lens_ldfs + len(ldfs)
sum_memo_ldfs = sum_memo_ldfs + Puzzle.num_of_instances


      new_row = pd.Series({'Алгоритм': 'LDFS', 'Час': t1,
'Операцій': len(ldfs), 'Пам\'ять': Puzzle.num_of_instances})
   data = pd.concat([data, new_row.to_frame().T],
ignore_index=True)
```

```
    Puzzle.num_of_instances = 0
    t0 = time()
    rbfs = recursive_best_first_search(state[i])
    t1 = time() - t0
    print('\nRBFS:', rbfs)
    print('Операцій:', len(rbfs))
    print('Пам\'ять:', Puzzle.num_of_instances)
    print('Час:', t1)

sum_time_rbfs = sum_time_rbfs + t1
sum_lens_rbfs = sum_lens_rbfs + len(rbfs)
sum_memo_rbfs = sum_memo_rbfs + Puzzle.num_of_instances


      new_row = pd.Series({'Алгоритм': 'RBFS', 'Час': t1,
'Операцій': len(rbfs), 'Пам\'ять': Puzzle.num_of_instances})
    data = pd.concat([data, new_row.to_frame().T],
ignore_index=True)

print('\nПорівняння алгоритмів:\n', data)
print('\n----------------------------------------\n')
print('\n----------------------------------------\n')
print('--------- Середні результати алгоритмів ---------')

print('\nСереднє по пам\'яті LDFS', (sum_memo_ldfs / 20))
print('Середнє по операціям LDFS', (sum_lens_ldfs / 20))
print('Середнє по часу LDFS', (sum_time_ldfs / 20))

print('\nСереднє по пам\'яті RBFS', (sum_memo_rbfs / 20))
print('Середнє по операціям RBFS', (sum_lens_rbfs / 20))
print('Середнє по часу RBFS', (sum_time_rbfs / 20))
```

Після кожного експерименту в консоль виводиться:

- шлях, за яким йшов алгоритм
- кількість операцій до досягнення цільового стану
- використана алгоритмом пам’ять
- час роботи алгоритму

```
----------------------------------------
Стартовий стан 1
[1, 3, 4]
[8, 6, 2]
[7, 0, 5]

LDFS: ['R', 'L', 'R', 'L', 'R', 'L', 'R', 'L', 'R', 'L', 'R', 'L', 'R', 'L', 'U', 'R', 'U', 'L', 'D']
Операцій: 19
Пам'ять: 527
Час: 0.00283050537109375

RBFS: ['U', 'R', 'U', 'L', 'D']
Операцій: 5
Пам'ять: 23
Час: 0.0
```

Рис. 1 Приклад виводу результатів експерименту

Для аналізу роботи алгоритмів виведемо середні результати по обом алгоритмам.

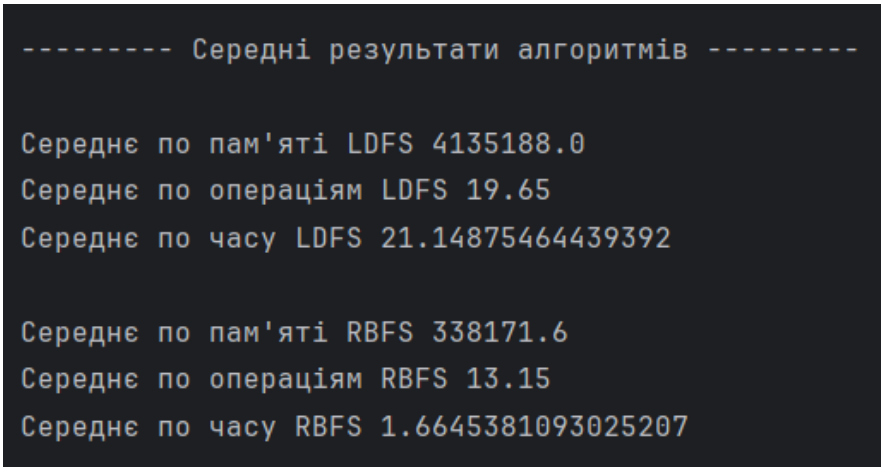


Рис. 2 Вивід середніх характеристик після серії експериментів

Загалом бачимо, що в результаті роботи обох алгоритмів LDFS за середніми характеристиками показує гірші параметри, аніж RBFS.

4. Аналіз алгоритмів

Проведемо порівняльний аналіз двох алгоритмів

LDFS (Limited Depth-First Search):

Переваги:

- Простий у реалізації та легкий для розуміння.
- Добре підходить для пошуку розв'язку на малих глибинах, коли глибина пошуку відома або невелика.
- Може бути менш вимогливим до пам'яті, оскільки використовує стек для збереження вузлів.

Недоліки:

- Не гарантує знаходження оптимального розв'язку, оскільки він не використовує інформацію про евристику.
- Може втратити багато часу, досліджуючи глибокі гілки пошуку, якщо оптимальний розв'язок знаходиться на значній відстані від початкового стану.

RBFS (Recursive Best-First Search):

Переваги:

- Гарантує знаходження оптимального розв'язку, оскільки використовує перший кращий пошук.
- Адаптивний, тобто алгоритм може адаптуватися до важких випадків, змінюючи ліміт функції оцінки.
- Використовує евристику для пришвидшення пошуку та керування процесом.

Недоліки:

- Може вимагати багато пам'яті для збереження стеку рекурсії та списку потенційних наступників.
- Має більш складну реалізацію порівняно з LDFS.
- При великих глибинах пошуку може стати менш ефективним через необхідність зберігати багато інформації у пам'яті.

Висновок

У ході виконання цієї лабораторної роботи було виявлено та досліджено особливості застосування алгоритмів неінформативного та інформативного пошуку. На основі наданих даних про середні результати роботи алгоритмів LDFS і RBFS можна зробимо наступні висновки:

Швидкодія: RBFS виявився швидшим алгоритмом у цьому дослідженні, оскільки середній показник часу - 1.66, на противагу LDFS - 21.15 середнє число операцій (13.15) у нього нижче, ніж у LDFS (19.65). Це свідчить про більшу продуктивність RBFS в розв'язанні задачі 8-puzzle.

Ефективність за ресурсами: LDFS використовує значно менше пам'яті, а саме в десять разів менше, ніж RBFS.

Завершеність: RBFS гарантує знаходження оптимального розв'язку, що важливо в багатьох практичних застосуваннях. LDFS, не маючи інформації про евристику, не може гарантувати знаходження оптимального розв'язку.

Практичний вибір: Вибір між LDFS і RBFS залежить від конкретного контексту та вимог задачі. Якщо важливо знайти найкращий розв'язок і пам'ять не обмежена, то RBFS може бути кращим вибором завдяки своїй швидкодії та гарантії оптимальності. Однак, якщо ресурси пам'яті обмежені, а оптимальність не є критичною, LDFS може бути практичнішим варіантом.