Національний технічний університет України «КПІ ім. Ігоря Сікорського»

Факультет Інформатики та Обчислювальної Техніки

Кафедра Автоматизованих Систем Обробки Інформації та Управління

**Лабораторна робота №2**

з дисципліни «Основи штучного інтелекту»

на тему:

*«***Інформативний пошук***»*

Варіант -13

Виконав:

студент групи ІС-72

Кривохижа Роман Андрійович

Київ - 2020

**Мета роботи**: розв’язати логічну задачу з використанням алгоритму інформативного пошуку.

**Завдання**: RBFS – Рекурсивний пошук за першим найкращим збігом

**Умова**: На ігровому полі 3 × 3 розставлені фішки з цифрами від 1 до 8. Є одна вільна клітина, на яку можуть бути перенесені фішки з сусідніх позицій. При цьому не допускається переміщення фішок через одну, а також по діагоналі. Завдання полягає в тому, щоб перетворити початкову конфігурацію в цільову конфігурацію

**GitHub:** <https://github.com/kryvokhyzha/Courses/tree/master/AI>

**Лістинг програми**:

* main.py

import numpy as np

from sys import maxsize

from node import Node

def recursive\_best\_first\_search(initial\_state):

node = Node(initial\_state)

result, best\_h = rbfs(node, maxsize, 0)

return result.find\_solution()

def rbfs(node, h\_limit, iter):

if node.is\_goal:

return node, None

iter += 1

successors = node.generate\_child()

if not len(successors):

return None, maxsize

for s in successors:

s.h = max(s.h, node.h)

while len(successors):

successors.sort(key=lambda x: x.h)

best = successors[0]

if best.h > h\_limit:

return None, best.h

if len(successors) > 1:

alternative = successors[1].h

else:

alternative = maxsize

# print(best)

result, best.h = rbfs(best, min(h\_limit, alternative), iter)

# print(best.h)

if result is not None:

return result, best.h

return None, None

if \_\_name\_\_ == '\_\_main\_\_':

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

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

a = np.array(a)

RBFS, depth = recursive\_best\_first\_search(a)

for i in RBFS:

print(i)

print('Depth:', depth)

* node.py

import numpy as np

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, goal=None, h\_mode='misplaced'):

self.state = state

self.parent = parent

self.action = action

self.goal = goal

if parent is not None:

self.depth = parent.depth + 1

else:

self.depth = 0

if self.goal is None:

self.goal = np.array([[1, 2, 3], [4, 0, 5], [6, 7, 8]])

else:

assert isinstance(self.goal, np.array)

self.h = self.\_\_heuristic(mode=h\_mode)

# print(self, self.h)

self.is\_goal = self.\_\_is\_goal()

def \_\_find\_zero\_pos(self):

i, j = np.where(self.state == 0)

return i[0], j[0]

@staticmethod

def \_\_find\_actions(i, j):

legal\_action = ['Up', 'Down', 'Left', 'Right']

if i == 0: # up is disable

legal\_action.remove('Up')

elif i == 2: # down is disable

legal\_action.remove('Down')

if j == 0:

legal\_action.remove('Left')

elif j == 2:

legal\_action.remove('Right')

return legal\_action

@staticmethod

def do\_action(new\_state, action):

shape = new\_state.shape

new\_state = np.reshape(new\_state, (1, 9))[0]

x = np.where(new\_state == 0)[0]

if action == 'Up':

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

elif action == 'Down':

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

elif action == 'Left':

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

elif action == 'Right':

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

new\_state = np.reshape(new\_state, shape)

return new\_state

def generate\_child(self):

children = []

i, j = self.\_\_find\_zero\_pos()

legal\_actions = self.\_\_find\_actions(i, j)

for action in legal\_actions:

new\_state = self.state.copy()

new\_state = self.do\_action(new\_state, action)

if self.parent is not None:

if (new\_state == self.parent.state).all():

continue

children.append(Node(new\_state, self, action))

return children

def \_\_heuristic(self, mode):

"""

Compute the heuristic value for current state.

"""

assert mode in {'manhattan', 'misplaced', 'linear\_m'}

if mode == 'manhattan':

return np.sum(np.abs(self.state - self.goal)) + self.depth

elif mode == 'misplaced':

return np.sum(self.state != self.goal) + self.depth

elif mode == 'linear\_m':

heuristic = 0

goal = np.reshape(self.goal, (1, 9))[0].tolist()

state = np.reshape(self.state, (1, 9))[0].tolist()

for num in range(1, 9):

distance = abs(state.index(num) - goal.index(num))

i = int(distance / 3)

j = int(distance % 3)

heuristic += i + j

return heuristic + self.depth

def \_\_is\_goal(self):

return (self.state == self.goal).all()

def find\_solution(self):

solution = [(self.state, self.action)]

path = self

while path.parent is not None:

path = path.parent

solution.append((path.state, path.action))

solution[-1] = (solution[-1][0], 'Start')

solution.reverse()

return solution, self.depth

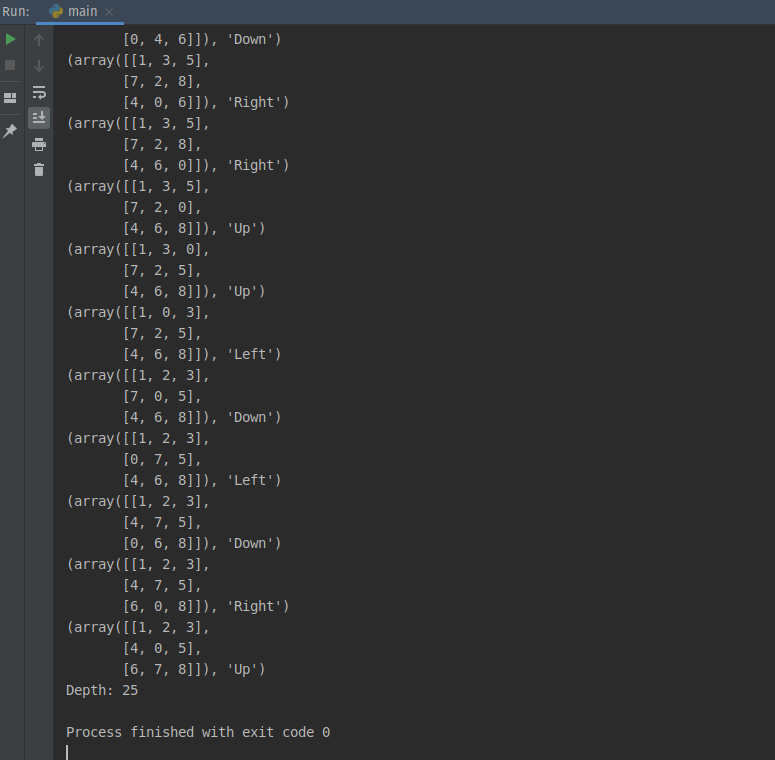
def \_\_repr\_\_(self):

return 'Node: h = {}; depth = {}; state = {}'.format(self.h, self.depth, self.state)

# def \_\_lt\_\_(self, node):

# return (self.state < node.state).all()

**Виконання програми**:

****

**Контрольні питання:**



unction A\*(start, goal, f)

% множество уже пройденных вершин

var closed := the empty set

% множество частных решений

var open := make\_queue(f)

enqueue(open, path(start))

while open is not empty

var p := remove\_first(open)

var x := the last node of p

if x in closed

continue

if x = goal

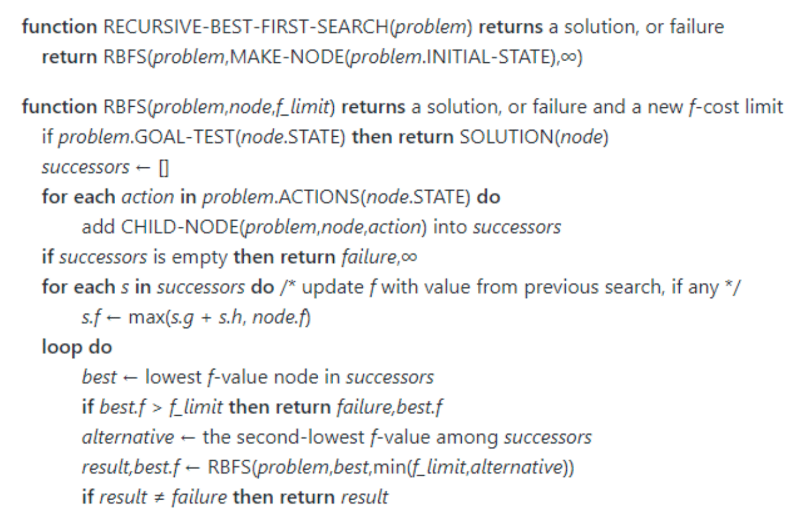
return p

add(closed, x)

% добавляем смежные вершины

foreach y in successors(x)

enqueue(open, add\_to\_path(p, y))



1. Інформативний пошук зазвичай забезпечує більш ефективний пошук у порівнянні з неінформативним пошуком, оскільки використовує додаткові знання про конкретну задачу, що формулюються у вигляді евристичної функції, яка на кожному кроці оцінює альтернативи на основі додаткової інформації, щоб прийняти рішення про подальший напрямок алгоритму.
2. Еристична функція потрібна, щоб спробувати уникнути повного перебору всіх можливих станів.
3. Функція завжди повинна повертати точне значення довжини найкоротшого рішення
4. Алгоритми A\* та RBFS є повними та оптимальними, якщо їх евристичні функції є допустимими, тобто такими, що ніколи не перевищують фактичну мінімальну вартість досягнення мети. Якщо алгоритм пошуку має ідеальну евристичну функцію, тобто таку, що завжди повертає точне значення довжини найкоротшого рішення, то він завжди буде оптимальним.
5. RBFS має структуру, аналогічну структурі рекурсивного пошуку в глибину, але замість нескінченного проходження вниз по поточному шляху даний алгоритм контролює f-значення найкращого альтернативного шляху, доступного з будь-якого предка поточного вузла. Якщо поточний вузол перевищує задану межу, то поточний етап рекурсії скасовується і рекурсія продовжується з альтернативного шляху. Після скасування даного етапу рекурсії в алгоритмі RBFS відбувається заміна f-значення кожного вузла вздовж даного шляху найкращим f-значенням його дочірнього вузла. Завдяки цьому в алгоритмі RBFS запам'ятовується f-значення найкращого листового вузла з забутого піддерева і тому в деякий наступний момент часу може бути прийнято рішення про те, чи варто знову розгортати це піддерево.

**Висновок:** для вирішення даної задачі мною було реалізовано алгоритм RBFS та декілька евристичних функцій: на основі поелементного порівняння станів та на основі манхеттенської відстані. Також, до кожної з евристичниї функцій додається штраф за глубину рекурсії. Для даної умови, найкращий шлях має глубину 25, відповідний скріншот наведено вище.