Laporan Tugas Kecil
Algoritma Branch and Bound untuk Menyelesaikan 15-puzzle

Maria Khelli
K01 / 13520115

I. Spesifikasi program

Bahasa yang digunakan : Python3
Tautan GitHub : https://github.com/khelli07/15-puzzle-solver
Keberjalanan program

No.	Poin	Ya	Tidak
1.	Program berhasil kompilasi	✓	
2.	Program berhasil dijalankan	✓	
3.	Program dapat menerima masukan dan menuliskan keluaran.	✓	
4.	Luaran sudah benar untuk seluruh data uji.	✓	
5.	Bonus dibuat.	Membuat animasi berbasis CLI (menggunakan timer dan cmd "cls")	

II. Algoritma program

Pada permasalahan ini, saya menggunakan algoritma *branch and bound* dengan solusi yang belum tentu optimal, yaitu mengambil jawaban kemunculan pertama—tidak disebutkan bahwa harus optimal. Algoritmanya berjalan sebagai berikut.

1. **Pembacaan masukan**: program menerima teks berisi 16 buah angka.
   Prekondisi: blok puzzle yang kosong diisi angka 16. Format masukan bebas karena program akan mengonversi masukan menjadi sebuah larik yang beranggotakan 16 elemen. Setiap angka harus dipisahkan oleh *whitespace*.
2. **Pengecekan awal**: larik diubah menggunakan numpy.reshape menjadi matriks 4×4. Kemudian, dicek sesuai fungsi berikut.

$$\sum_{i=1}^{16} \text{kurang(i)} + X$$

   Dengan kurang(i) adalah jumlah angka yang memiliki nilai kurang dari matriks[i][j] dan berada pada indeks linear (4i + j) yang lebih besar daripada indeks linear sekarang. Kemudian, X adalah nilai integer 0/1 yang bernilai 0 jika indeks (i, j) dari blok kosong keduanya genap atau keduanya ganjil. X akan bernilai 1 jika indeks (i, j) hanya salah satu yang ganjil/genap.
3. Jika nilai pada butir 2 adalah ganjil (tidak memenuhi): akan muncul pesan, "Board can not be solved."
4. Jika nilai pada butir 2 adalah genap (memenuhi): akan dijalankan algoritma penyelesaian.

5. **Algoritma penyelesaian**: program membangkitkan sebuah node akar yang menyimpan informasi *board puzzle.* Kemudian, program membuat sebuah struktur data *priority queue* (PQ) yang diimplementasikan menggunakan *heap*. Program juga akan menyimpan sebuah *hashmap* yang memiliki *key* representasi data larik (*board*) yang disimpan dalam bentuk byte mentah (konversi menggunakan numpy.tobytes).
6. Simpul akar yang dibangkitkan dimasukkan ke dalam PQ.
7. Kemudian, head dari PQ di-*pop* dan diproses. Pada setiap pemrosesan, simpul akan ditandai sudah dikunjungi. Kemudian, pohon ruang status akan diperluas dengan membangkitkan anak dari simpul dengan urutan langkah *up, right, down, left*.
8. Untuk setiap simpul anak yang valid, akan dilakukan pemrosesan. Pertama, akan dicek apakah larik sudah terselesaikan atau belum. Jika sudah, lanjut ke langkah berikutnya. Jika belum, tambahkan simpul anak ke dalam antrian PQ.
9. Lakukan langkah 8 sampai seluruh simpul anak telah diproses.
10. Selama jawaban belum ditemukan, ulangi dari langkah 7.
11. Jika jawaban sudah ditemukan, program akan melakukan *tracing* dari simpul daun sampai ke simpul akar untuk membangkitkan *path* atau jalan yang dilakukan. Program akan mengeluarkan seluruh langkah yang dilakukan, total simpul yang dibangkitkan, total simpul yang diproses, serta waktu eksekusi dari langkah 5 sampai 10.

Perhatikan bahwa setiap simpul memiliki informasi *cost*, yaitu level (akar memiliki level 0) dari simpul ditambah dengan taksiran *cost* yang merupakan jumlah (*count*) angka yang tidak pada tempatnya. Informasi ini yang digunakan untuk menentukan simpul selanjutnya yang akan diproses (yang terkecil dan terbaru dimasukkan).

III. Tangkapan layar masukan dan keluaran

Masukan: nama file “test1.txt” yang berada dalam folder test.

Luaran:

```
Input filename: test1.txt
Path: RIGHT -> RIGHT -> UP -> UP -> LEFT -> DOWN -> RIGHT -> DOWN
MOVE = DOWN, COST = 8
==================
  1   2   3   4
  5   6   7   8
  9   10  11  12
  13  14  15
==================
Total node generated: 33
Total node processed: 13
Total solve time: 0.00 seconds
```

Jika dijalankan secara langsung, path yang dilakukan akan disimulasikan secara dinamis (antarmuka berbasis CLI).

IV. Contoh persoalan yang diselesaikan

Persoalan 1 (bisa diselesaikan)

1 2 3 4
5 6 8 11
9 10 7 12
13 16 14 15

Persoalan 2 (bisa diselesaikan)

2 3 4 16
1 5 8 11
9 6 10 12
13 14 7 15

Persoalan 3 (bisa diselesaikan)

1 3 15 4
10 2 16 11
5 14 8 6
9 13 7 12

Persoalan 4 (tidak bisa diselesaikan)

1 3 15 4
10 2 7 11
5 14 8 6
9 13 16 12

Persoalan 5 (tidak bisa diselesaikan)

2 3 4 11
1 5 7 8
9 6 12 15
13 14 16 10

V. Kode program

board.py

```
from src.utils import EMPTY
import numpy as np


class Board:
    """
        This class is the representation of the puzzle.
        The class stores the array (board) and the block index
        which is empty. Hence, the program need not to iterate
        everytime it need to know where is the empty block.
    """

    def __init__(self, array, empty, cost):
        self.array = array
        self.empty = empty
        self.cost = cost
        if not (cost):
            _ = self.calc_cost()

        self.str = ""

    def is_index_valid(self, i, j):
        """
            Returns true if index is not out of bound.
        """
        return (0 <= i < 4) and (0 <= j < 4)

    def to_xy(self, i):
        """
            Convert array index to matrix index.
        """
        return (i // 4, i % 4)
```

```
def wrong_pos(self, array, i):
    return array[i] != i + 1

def calc_cost(self):
    """
        Count the blocks that are not in position.
    """

    if not (self.cost):
        total = 0
        for i in range(len(self.array)):
            if self.array[i] != EMPTY and (self.wrong_pos(self.array, i)):
                total += 1

            self.cost = total

    return self.cost

def move(self, to):
    """
        Generating a child class if the direction is valid.
        Returns None if direction is not valid.
    """

    i = self.empty
    x, y = self.to_xy(to)

    if self.is_index_valid(x, y):
        acopy = np.copy(self.array)
        acopy[i], acopy[to] = acopy[to], EMPTY
        cost = self.cost

        if self.wrong_pos(self.array, to) and not(self.wrong_pos(acopy, i)):
            cost -= 1
```

```
        elif not(self.wrong_pos(self.array, to)) and self.wrong_pos(acopy, i):
            cost += 1

        return Board(acopy, to, cost)
    else:
        return None

def from_right(self):
    return self.move(self.empty + 1)

def from_left(self):
    return self.move(self.empty - 1)

def from_up(self):
    return self.move(self.empty - 4)

def from_down(self):
    return self.move(self.empty + 4)

def to_string(self):
    """
        Converting the array to string (matrix-like shape).
    """

    if self.str == "":
        self.str = "  "
        for i in range(len(self.array)):
            num = self.array[i]
            if num == EMPTY:
                self.str += "   "
            else:
                self.str += str(num).ljust(3, " ")
            self.str += " "
```

```
                if (i + 1) % 4 == 0 and i != (len(self.array) - 1):
                    self.str += "\n  "

        return self.str

    def print_board(self):
        print(self.to_string())

    def is_finish(self):
        return self.cost == 0
```

node.py

```
from src.utils import UP, RIGHT, DOWN, LEFT

class Node:
    """
        Node class is used to generate state space tree.
        It stores its parent's node and how many steps or level
        it has gone through (important to find the total cost).
    """

    def __init__(self, board, before, steps, move):
        self.board = board
        self.before = before

        self.steps = steps
        self.move = move

    def cost(self):
        """
            Cost function g(i) + f(i)
        """
        return self.board.calc_cost() + self.steps
```

```
def __lt__(self, other):
    """
        Overriding comparison for the heap to work.
    """
    return self.cost() <= other.cost()


def key(self):
    """
        Convert node to string (raw) to be used in dictionary key.
    """
    return self.board.array.tobytes()


def expand(self):
    """
        Generating valid child nodes.
    """
    all_possible_children = [
        (self.board.from_up(), UP),
        (self.board.from_right(), RIGHT),
        (self.board.from_down(), DOWN),
        (self.board.from_left(), LEFT),
    ]

    return [Node(child, self, self.steps + 1, move)
            for child, move in all_possible_children if child]


def print_node(self):
    print("MOVE = " + self.move + ", COST = " + str(self.steps))
    print("==================")
    self.board.print_board()
    print("==================")
```

utils.py

```
import time
import os

EMPTY = 16
INF = 999
UP = "UP"
RIGHT = "RIGHT"
DOWN = "DOWN"
LEFT = "LEFT"

def find_empty(array):
    for i in array:
        if array[i - 1] == EMPTY:
            return i - 1

def is_solvable(matrix):
    total = 0
    X = -1

    passed = []
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            passed.append(matrix[i][j])
            if matrix[i][j] == EMPTY:
                X = int((i % 2) + (j % 2) == 1)

            total += matrix[i][j] - 1 - \
                sum([1 for el in passed if el < matrix[i][j]])
    return (total + X) % 2 == 0
```

```
def print_steps(root, nodes):
    root.print_node()
    time.sleep(0.5)
    os.system('cls')
    for node in nodes:
        node.print_node()
        time.sleep(0.5)
        os.system('cls')

    return
```

solver.py

```
import numpy as np
from heapq import heappop, heappush
from src.board import Board
from src.node import Node
from src.utils import *


def solve(filename):
    f = open("test/" + filename, "r")
    array = [int(i) for i in f.read().strip().split()]
    matrix = np.reshape(array, (4, 4))
    f.close()

    if not (is_solvable(matrix)):
        print("Board can not be solved.")
    else:
        root = Node(
            Board(np.array(array), find_empty(array), None),
            None, 0, "INITIAL"
        )
        heap = []
```

```
heappush(heap, root)

visited = {}

answer = None
node_counter = 1  # ROOT
processed_counter = 1

start_time = time.time()
while not(answer):
    current_node = heappop(heap)
    processed_counter += 1
    visited[current_node.key()] = True

    if current_node.board.is_finish():
        answer = root
        break

    for child in current_node.expand():
        if child.board.is_finish():
            node_counter += 1
            answer = child
            break

        elif child.key() not in visited:
            node_counter += 1
            heappush(heap, child)

total_exec = time.time() - start_time

# Compute paths and nodes
paths = []
nodes = []
cn = answer
```

```
while cn.steps != 0:
    paths.insert(0, cn.move)
    nodes.insert(0, cn)
    cn = cn.before

print_steps(root, nodes)

print("Path:", " -> ".join(paths))
answer.print_node()
print("Total node generated:", node_counter)
print("Total node processed:", processed_counter)
print("Total solve time: {:.2f} seconds".format(total_exec))
```