PENYELESAIAN PERSOALAN 15-PUZZLE DENGAN ALGORITMA *BRANCH AND BOUND*

LAPORAN TUGAS KECIL

diajukan untuk memenuhi persyaratan
IF2211 Strategi Algoritma

oleh
Jeremy S.O.N. Simbolon
13520042

TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2022

A. Deskripsi Algoritma

Untuk menyelesaikan *mystic square/15-puzzle*, program akan menggunakan pendekatan *branch-and-bound*. Pertama, program akan memeriksa apakah *mystic square* dapat dicari solusinya menggunakan kosnep *inversion*. Apabila *mystic square* tidak dapat diselesaikan, program akan mencetak hal tersebut ke layar dan berhenti berjalan. Apabila *mystic square* dapat diselesaikan, instans *tile* dari *puzzle* akan dimasukkan ke dalam *loop* yang bertujuan untuk mencari solusi.

Di dalam *loop*, anak-anak dari instans *tile* akan dibangkitkan dan dimasukkan ke dalam *priority queue* berdasarkan *cost*-nya, yakni hasil penjumlahan kedalaman anak pada pohon ruang pencarian dengan taksiran jarak anak ke instans solusi berupa jumlah *tile* yang tempatnya tidak sesuai. Anak dengan *cost* terendah akan dikeluarkan dari *priority queue* dan diperiksa apakah memenuhi solusi. Bila tidak, *loop* akan diulang kembali dengan instans anak tersebut sebagai instans induk yang anaknya akan dibangkitkan. Apabila anak tersebut memenuhi solusi, instans pada *priority queue* dengan *cost* yang lebih tinggi dari *cost* anak akan dikeluarkan. Jika *priority queue* menjadi kosong setelah proses pengeluaran, solusi telah ditemukan. Apabila masih ada instans yang terletak di dalam *priority queue*, instans dengan *cost* terendah akan dikeluarkan dan dinyatakan sebagai instans induk baru; *loop* pun akan dimulai kembali.

B. Kode Sumber

1. mystic_square_solver.py

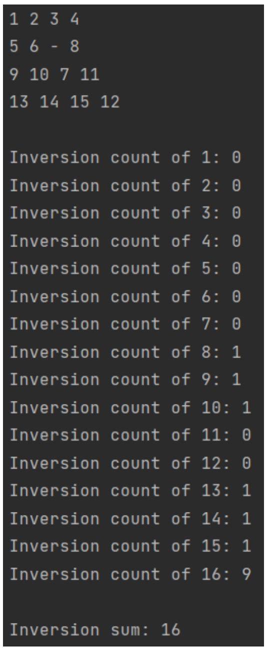
```
import argparse

import mystic

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-s", "--source",
                        help="text file containing your mystic square",
                        type=str)

    source_file = parser.parse_args()
    mystic_square = mystic.MysticSquare(source_file.source)
    mystic_square.build_mystic_square()
    mystic_square.display_mystic_square()
    mystic_square.solvable_check()
    mystic_square.solve_mystic()
    mystic_square.print_solution()
```

2. mystic.py

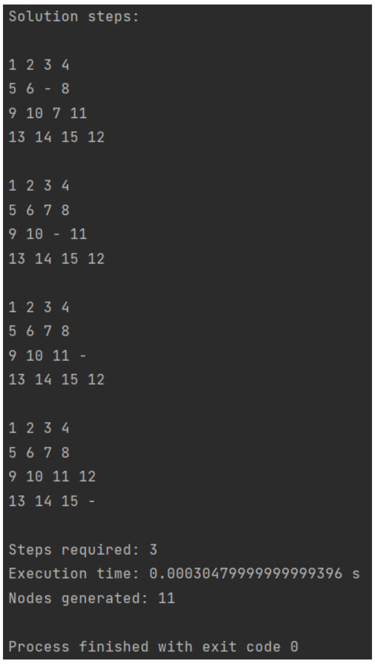
```
import copy
import heapq
import os
import random
import timeit


class MysticSquare:
    def __init__(self, source_file):
        self.__SOLUTION = ("1", "2", "3", "4", "5", "6", "7", "8", "9",
                           "10", "11", "12", "13", "14", "15", "-")
        self.__VALID_MOVES = ("up", "right", "down", "left")

        self.__inversion_count = 0
        self.__moves = []
        self.__node_count = 0
        self.__solution_stack = []
        self.__source_file = source_file
        self.__tiles = []
        self.__time = 0

    def build_mystic_square(self):
        if not self.__source_file:
            self.__tiles = random.sample(self.__SOLUTION, len(self.__SOLUTION))
        else:
            self.__tiles = self.__parse_mystic_txt()

    def __parse_mystic_txt(self):
        os.chdir("../test/")
        file = open(self.__source_file, "r")
        raw_lines = file.readlines()
        file.close()

        mystic_tiles = []
        lines = [raw_line.replace("\n", "") for raw_line in raw_lines]
        tile_rows = [line.split(" ") for line in lines]
        for tile_row in tile_rows:
            mystic_tiles.extend(tile_row)

        return mystic_tiles

    def display_mystic_square(self, tiles=None):
        if not tiles:
            tiles = self.__tiles
        for i in range(len(tiles)):
            print(tiles[i], end="")
            if i % 4 != 3:
                print(end=" ")
            else:
                print(end="\n")
        print()

    def solvable_check(self):
        self.__time = timeit.default_timer()
        for i in range(1, 17):
            inversion_count = 0
            if i == 16:
                inversion_count += 15 - self.__tiles.index("-")
            else:
                for j in range(1, i):
                    if self.__tiles.index(str(j)) > self.__tiles.index(str(i)):
                        inversion_count += 1
            print(f"Inversion count of {i}: {inversion_count}")
            self.__inversion_count += inversion_count

        if self.__tiles.index("-") % 8 in (1, 3, 4, 6):
            self.__inversion_count += 1

        print()
        print(f"Inversion sum: {self.__inversion_count}")
        print()

    def solve_mystic(self):
        if self.__inversion_count % 2 == 1:
            print("Mystic square is not solvable :(")
        else:
            start_instance = copy.deepcopy(self.__tiles)
            self.__build_tree(start_instance)
            self.__time -= timeit.default_timer()
            self.__time *= -1

    def __build_tree(self, start_instance, depth=-1):
        while True:
            blank_index = start_instance.index("-")
            for move in self.__VALID_MOVES:
                tile_instance = copy.deepcopy(start_instance)

                if move == "up" and blank_index not in (0, 1, 2, 3):
                    (tile_instance[blank_index],
                     tile_instance[blank_index - 4]) = (tile_instance[blank_index - 4],
                                                        tile_instance[blank_index])
                    self.__node_count += 1
                elif move == "right" and blank_index not in (3, 7, 11, 15):
                    (tile_instance[blank_index],
                     tile_instance[blank_index + 1]) = (tile_instance[blank_index + 1],
                                                        tile_instance[blank_index])
                    self.__node_count += 1
                elif move == "down" and blank_index not in (12, 13, 14, 15):
                    (tile_instance[blank_index],
                     tile_instance[blank_index + 4]) = (tile_instance[blank_index + 4],
                                                        tile_instance[blank_index])
                    self.__node_count += 1
                elif move == "left" and blank_index not in (0, 4, 8, 12):
                    (tile_instance[blank_index],
                     tile_instance[blank_index - 1]) = (tile_instance[blank_index - 1],
                                                        tile_instance[blank_index])
                    self.__node_count += 1
                else:
                    continue

                cost = self.__calculate_cost(tile_instance, depth)
                heapq.heappush(self.__moves, (cost, depth, tile_instance))

            next_instance = heapq.heappop(self.__moves)
            self.__solution_stack.append(next_instance)
            if tuple(next_instance[2]) == self.__SOLUTION:
                for i in range(len(self.__moves)):
                    if self.__moves[i][0] > next_instance[0]:
                        self.__moves = copy.deepcopy(self.__moves[:i])
                        break
                if len(self.__moves) > 1:
                    next_instance = heapq.heappop(self.__moves)
                    for _ in range(depth - next_instance[1]):
                        self.__solution_stack.pop()
                    start_instance = copy.deepcopy(next_instance[2])
                    depth = next_instance[1] - 1
                else:
                    break
            else:
                start_instance = copy.deepcopy(next_instance[2])
                depth = next_instance[1] - 1

    @staticmethod
    def __calculate_cost(instance, depth):
        incorrect_position = 0

        for i in range(len(instance)):
            if instance[i] == "-":
                continue
            elif instance[i] != str(i + 1):
                incorrect_position += 1

        return incorrect_position - depth

    def print_solution(self):
        if self.__solution_stack:
            print("Solution steps:\n")
            self.display_mystic_square()
            for i in range(len(self.__solution_stack)):
                self.display_mystic_square(self.__solution_stack[i][2])

            print(f"Steps required: {len(self.__solution_stack)}")
            print(f"Execution time: {self.__time} s")
            print(f"Nodes generated: {self.__node_count}")
```

C. Uji Coba Program

1. *Testcase* 1

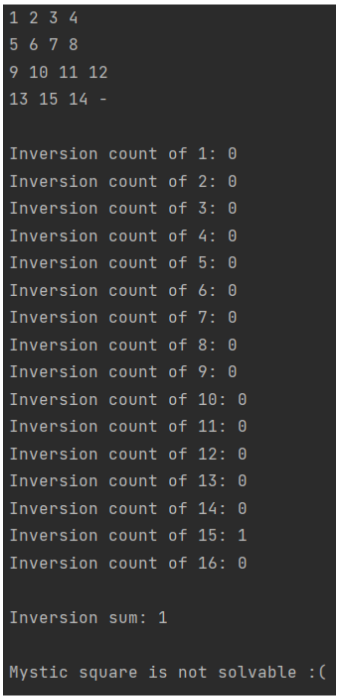


```
1 2 3 4
5 6 - 8
9 10 7 11
13 14 15 12

Inversion count of 1: 0
Inversion count of 2: 0
Inversion count of 3: 0
Inversion count of 4: 0
Inversion count of 5: 0
Inversion count of 6: 0
Inversion count of 7: 0
Inversion count of 8: 1
Inversion count of 9: 1
Inversion count of 10: 1
Inversion count of 11: 0
Inversion count of 12: 0
Inversion count of 13: 1
Inversion count of 14: 1
Inversion count of 15: 1
Inversion count of 16: 9

Inversion sum: 16
```



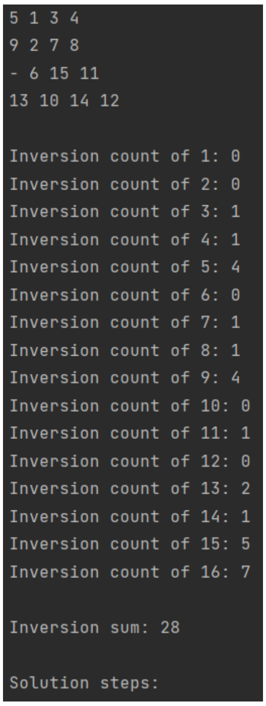
```
Solution steps:

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

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

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

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

Steps required: 3
Execution time: 0.00030479999999999396 s
Nodes generated: 11

Process finished with exit code 0
```

2. *Testcase* 2

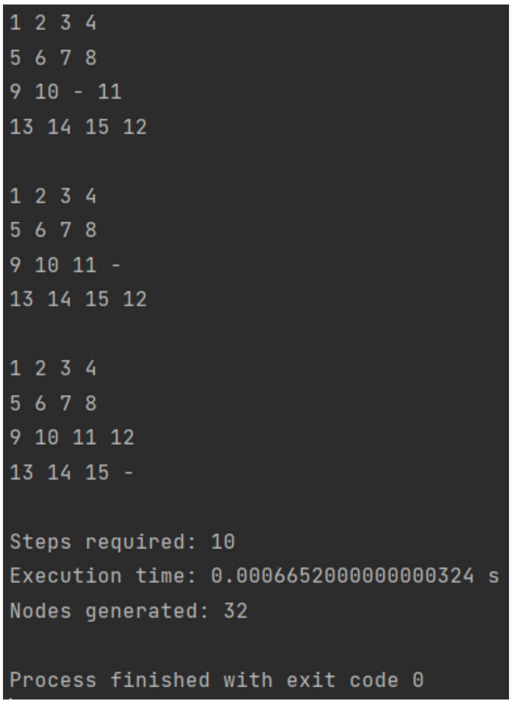


```
1 2 3 4
5 6 7 8
9 10 11 12
13 15 14 -

Inversion count of 1: 0
Inversion count of 2: 0
Inversion count of 3: 0
Inversion count of 4: 0
Inversion count of 5: 0
Inversion count of 6: 0
Inversion count of 7: 0
Inversion count of 8: 0
Inversion count of 9: 0
Inversion count of 10: 0
Inversion count of 11: 0
Inversion count of 12: 0
Inversion count of 13: 0
Inversion count of 14: 0
Inversion count of 15: 1
Inversion count of 16: 0

Inversion sum: 1

Mystic square is not solvable :(
```

3. *Testcase* 3

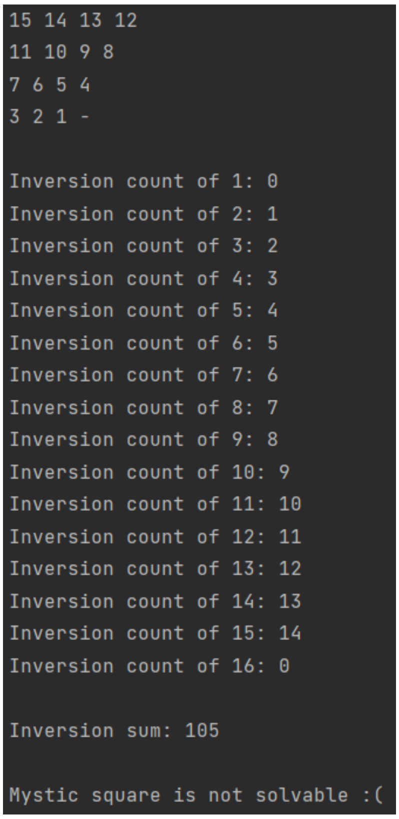


```
5 1 3 4
9 2 7 8
- 6 15 11
13 10 14 12

Inversion count of 1: 0
Inversion count of 2: 0
Inversion count of 3: 1
Inversion count of 4: 1
Inversion count of 5: 4
Inversion count of 6: 0
Inversion count of 7: 1
Inversion count of 8: 1
Inversion count of 9: 4
Inversion count of 10: 0
Inversion count of 11: 1
Inversion count of 12: 0
Inversion count of 13: 2
Inversion count of 14: 1
Inversion count of 15: 5
Inversion count of 16: 7

Inversion sum: 28

Solution steps:
```



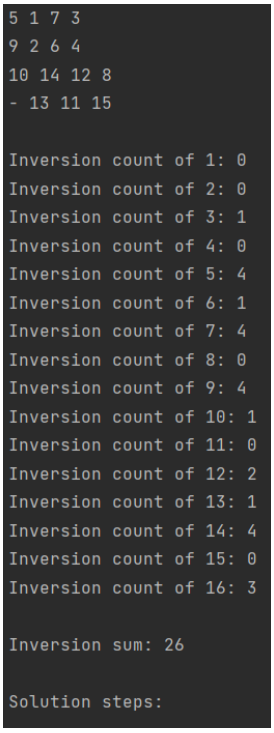
```
1 2 3 4
5 6 7 8
9 10 - 11
13 14 15 12

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

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

Steps required: 10
Execution time: 0.0006652000000000324 s
Nodes generated: 32

Process finished with exit code 0
```

4. *Testcase* 4

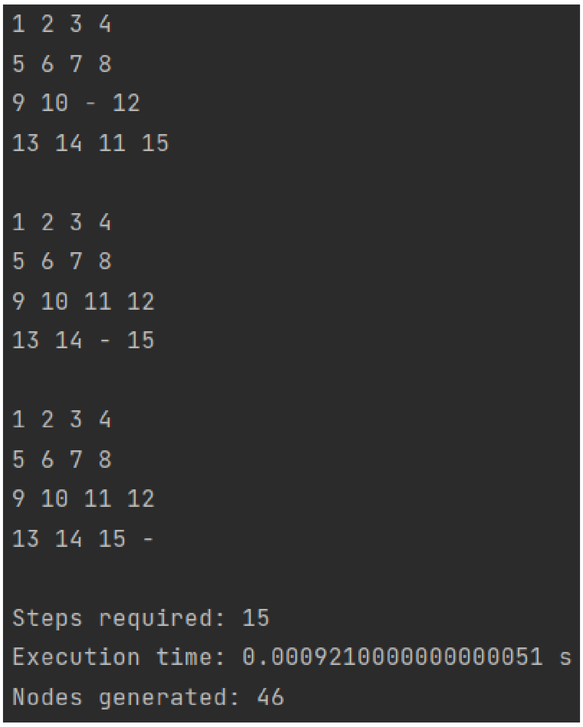


```
15 14 13 12
11 10 9 8
7 6 5 4
3 2 1 -

Inversion count of 1: 0
Inversion count of 2: 1
Inversion count of 3: 2
Inversion count of 4: 3
Inversion count of 5: 4
Inversion count of 6: 5
Inversion count of 7: 6
Inversion count of 8: 7
Inversion count of 9: 8
Inversion count of 10: 9
Inversion count of 11: 10
Inversion count of 12: 11
Inversion count of 13: 12
Inversion count of 14: 13
Inversion count of 15: 14
Inversion count of 16: 0

Inversion sum: 105

Mystic square is not solvable :(
```

5. *Testcase* 5



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

Inversion count of 1: 0
Inversion count of 2: 0
Inversion count of 3: 1
Inversion count of 4: 0
Inversion count of 5: 4
Inversion count of 6: 1
Inversion count of 7: 4
Inversion count of 8: 0
Inversion count of 9: 4
Inversion count of 10: 1
Inversion count of 11: 0
Inversion count of 12: 2
Inversion count of 13: 1
Inversion count of 14: 4
Inversion count of 15: 0
Inversion count of 16: 3

Inversion sum: 26

Solution steps:
```



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

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

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

Steps required: 15
Execution time: 0.00092100000000000051 s
Nodes generated: 46
```

D. Repositori Program

Kode sumber dari program ini dapat diakses pada repositori GitHub yang terdapat pada laman https://github.com/tastytypist/mystic-square-solver.

E. Lampiran

Poin	**Ya**	**Tidak**
Program berhasil dikompilasi	✓	
Program berhasil *running*	✓	
Program dapat menerima *input* dan menuliskan *output*	✓	
Luaran sudah benar untuk semua data uji	✓	
Bonus dibuat		✓