LAPORAN TUGAS KECIL 1

IF2211 STRATEGI ALGORITMA

Penyelesaian IQ Puzzler Pro dengan Algoritma Brute Force

Disusun oleh:

Muhammad Aufa Farabi 13523023

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2025

DAFTAR ISI

BAGIAN I
ALGORITMA BRUTE FORCE

Algoritma Brute Force adalah algoritma yang menggunakan secara lurus dalam menyelesaikan suatu masalah. Algoritma ini merupakan suatu algoritma yang dapat secara pasti menemukan solusi dari suatu persoalan meskipun dengan waktu yang lama karena algoritma Brute Force akan mencoba semua kemungkinan kasus untuk mendapatkan solusi optimal nya. IQ Puzzler Pro merupakan salah satu permainan yang dapat diselesaikan dengan algoritma Brute Force dengan cara mencoba segala kemungkinan dari urutan penempatan blok atau *pieces* pada papan dengan setiap blok juga diuji semua kemungkinan posisi atau bentuk tranformasi (rotasi dan pencerminan) yang dapat ditempatkan pada papan. Algoritma *Brute Force* yang digunakan untuk menyelesaikan permainan IQ Puzzler Pro dilakukan dengan cara berikut:

1. Pastikan input berupa ukuran papan dan blok-bloknya valid dalam permainan. Blok awalnya di simpan pada program sebagai map dengan *key* nya berupa huruf balok dan nilai nya adalah *list* koordinat dari setiap unit blok yang kemudian diubah menjadi objek blok. Sebelum dilakukan algoritma brute force, urutan-urutan penempatan blok pada papan dicari dengan menggunakan permutasi dan disimpan sebagai *list of list of block objects* atau list yang mengandung urutan dari penempatan blok-blok, seperti [[A,B,C],[A,C,B],[B,A,C]. List ini akan menjadi dasar dari proses *looping* terluar pada algoritma brute force yang akan dilakukan.
2. Algoritma *Brute force* dimulai dari kombinasi urutan blok yang paling pertama dari *loop* hingga urutan blok terakhir dari permutasi. Setiap blok dari kombinasi urutannya ditempatkan satu per satu pada papan dengan setiap blok akan dicari semua bentuknya transformasi (rotasi atau pencerminan) dan disimpan sebagai *list of block objects*. Setiap Blok akan diiterasi dari *list* bentuk transformasi blok yang dimiliki dengan blok dicoba untuk ditempatkan dari posisi paling atas kanan papan (i = 0, j = 0) kemudian digeser secara kolom kemudian baris hingga posisi paling bawah kanan dari papan (i = N, j = M) sampai dapat ditempatkan pada papan. Blok lain akan ditempatkan ke pada papan sesuai urutan blok setelah blok sebelumnya dapat ditempatkan kepada papan kemudian diulangi proses yang dilakukan seperti sebelumnya. Apabila blok gagal ditempatkan pada semua posisi dalam papan, bentuk transformasi lain dari blok (berupa rotasi atau pencerminan dari blok) tersebut akan dicoba untuk ditempatkan pada papan kemudian diulangi proses penempatan blok seperti langkah sebelumnya.
3. Ketika suatu blok gagal ditempatkan pada semua posisi dalam papan meskipun sudah dicoba semua bentuk rotasi/ dan atau pencerminan dari blok. Algoritma *Brute Force* dalam program akan mundur ke blok sebelumnya yang sudah ditempatkan (*backtracking)* kemudian blok sebelumnya tersebut dirotasi pada papan. Proses *backtracking* ini tetap tergolong pada Algoritma *Brute Force* karena tidak ada teknik heuristik (melakukan *educated guess* atau mencari langkah optimal selanjutnya) yang dipakai dalam program. Proses ini dapat terjadi karena program menyalin papan

untuk setiap penempatan blok dengan fungsi rekursif untuk menempatkan blok selanjutnya dalam papan salinan tersebut, artinya pada proses *backtracking*, program kembali memakai papan atau salinan papan lama yang ditempatkan oleh blok sebelumnya. Proses rekursif pada program dapat dijelakan sebagai berikut :

- Basis: pengecekan apakah semua blok telah mengisi papan dan papan sudah terisi pada fungsi *parent tryOrder*.

Pada Fungsi *tryOrder,* selain pengecekan sebelumnya. bentuk transformasi dari suatu blok didapatkan kemudian diiterasi dengan memanggil fungsi *tryPlaceBlock* pada tiap iterasi dengan membentuk papan *temporary* baru dan akan dicoba mengiterasi setiap blok pada posisi papan baru tersebut

- Rekurens: pada setiap kasus blok dapat ditempatkan pada papan dengan fungsi *tryPlaceBlock*. Fungsi tersebut akan membuat salinan dari papan kemudian memanggil fungsi *parent tryOrder* untuk penempatan blok selanjutnya pada salinan papan tersebut.

4. Jika seluruh bentuk transformasi dari suatu urutan blok tidak dapat mengisi seluruh bagian papan, program akan beralih pada urutan blok yang lain pada *loop* urutan blok kemudian mengulangi proses yang dilakukan pada langkah No. 2 dan No. 3.
5. Dalam hal ini, jika seluruh blok dapat ditempatkan pada salinan papan terakhir dan tidak ada posisi kosong pada papan, kondisi papan terakhir akan disimpan sebagai solusi yang telah ditemukan. Jika masih ada posisi kosong pada papan, solusi dinyatakan tidak ada.

BAGIAN II
SOURCE PROGRAM

Program utama ditulis dalam bahasa C dengan memakai *packages* di antaranya:

1. Java.util
2. java.io
3. Javax.swing (GUI)
4. Javax.awt (GUI)

Program ini terdiri atas 4 *files* yang bersesuaian dengan kelas nya dan 1 *file* sebagai GUI atau Main

- Files
- Board
- Block
- Solver
- Main (GUI)

Kelas Solver

```
import java.util.*;

public class Solver {
    private char[][] board;
    private List<Block> blocks;
    private int casesCount;

    public Solver(char[][] board, Map<Character, List<int[]>> mapBlocks) {
        this.board = board;
        this.blocks = new ArrayList<>();
        this.casesCount = 0;

        // Ubah map daftar Blok menjadi list dari objek block
        for (Map.Entry<Character, List<int[]>> entry : mapBlocks.entrySet()) {
            blocks.add(new Block(entry.getKey(), entry.getValue()));
        }
    }

    // Getter
    public Board getSolvedBoard() {
        return new Board(this.board);
    }
    public int getCasesCount() {
        return this.casesCount;
    }
```

```java
public boolean bruteForceSolve() {
    List<List<Block>> blockPermutations = getBlockPermutations();

    // Coba semua urutan permutasi blok
    for (List<Block> blockOrder : blockPermutations) {
        if (tryOrder(blockOrder, new Board(this.board), 0)) {
            return true;
        }
    }
    System.out.println("Tidak ada solusi yang ditemukan.");
    return false;
}

// Mencoba menempatkan blok dari urutan permutasi blok
private boolean tryOrder(List<Block> blockOrder, Board currentBoard, int blockIdx) {
    if (blockIdx >= blockOrder.size()) {
        casesCount++;
        if (!currentBoard.hasEmptySpace()) {
            this.board = currentBoard.getBoard();
            return true;
        }
        return false; // Semua blok telah ditempatkan tapi papan tidak penuh
    }

    Block block = blockOrder.get(blockIdx);
    List<List<int[]>> blockTransformations = block.getBlockTransformations();
```

```java
    // Coba semua transformasi (Rotasi + mirror) dari blok
    for (List<int[]> blockTransformation : blockTransformations) {
        if (tryPlaceBlock(blockOrder, currentBoard, blockIdx, blockTransformation)) {
            return true;
        }
    }

    return false;
}
private boolean tryPlaceBlock(List<Block> blockOrder, Board currentBoard, int blockIdx, List<int[]> blockTransformation) {
    boolean placed = false;
    for (int y = 0; y < board.length; y++) {
        for (int x = 0; x < board[0].length; x++) {
            if (currentBoard.canPlaceBlock(blockTransformation, y, x)) {
                placed = true;
                Board newBoard = new Board(currentBoard); // copy temporary papan baru
                newBoard.placeBlock(blockTransformation, y, x, blockOrder.get(blockIdx).getAbjad());

                if (tryOrder(blockOrder, newBoard, blockIdx + 1)) {
                    return true;
                }
            }
        }
    }
    if (!placed) { // Jika blok tidak bisa ditempatkan dalam setiap posisi
        casesCount++;
```

```
        return false;
    }

    // cari semua permutasi urutan blok
    private List<List<Block>> getBlockPermutations() {
        List<List<Block>> permutations = new ArrayList<>();
        permutationHelper(blocks, 0, permutations);
        return permutations;
    }

    private void permutationHelper(List<Block> blocks, int level, List<List<Block>> permutations) {
        if (level >= blocks.size()) {
            permutations.add(new ArrayList<>(blocks));
            return;
        }

        for (int i = level; i < blocks.size(); i++) {
            Collections.swap(blocks, level, i);
            permutationHelper(blocks, level + 1, permutations);
            Collections.swap(blocks, level, i);
        }
    }
}
```

Kelas Board

```
import java.util.Arrays;
import java.util.List;

public class Board {
    private int N, M;
    private char[][] board;


    // Buat objek board baru dari kolom dan baris
    public Board(int N, int M) {
        this.N = N;
        this.M = M;
        this.board = new char[N][M];
        for (int row = 0; row < N; row++) {
            for (int unit = 0; unit < M; unit++) {
                board[row][unit] = '0';
            }
        }
    }

    // Buat objek board baru dari array board
    public Board(char[][] board) {
        this.N = board.length;
        this.M = board[0].length;
        this.board = new char[N][M];
        for (int i = 0; i < N; i++) {
            System.arraycopy(board[i], 0, this.board[i], 0, M);
        }
    }
```

```java
    // Buat objek board baru dari board lain
    public Board(Board baseBoard) {
        this.N = baseBoard.N;
        this.M = baseBoard.M;
        this.board = new char[N][M];
        for (int i = 0; i < N; i++) {
            System.arraycopy(baseBoard.board[i], 0, this.board[i], 0, M);
        }

    }

    // Getter
    public char[][] getBoard() {
        return board;
    }

    // Cek blok dapat ditempatkan pada lokasi yang kosong dalam board
    public boolean canPlaceBlock(List<int[]> block, int y, int x) {
        for (int[] unit : block) {
            int yPosition = unit[0] + y;
            int xPosition = unit[1] + x;
            if ((yPosition >= N || xPosition >= M || board[yPosition][xPosition] != '0'
            || board[yPosition][xPosition] == '.')) {
                return false;
            }
        }
        return true;
    }
```

```java
    public void placeBlock(List<int[]> block, int y, int x, char abjad) {
        for (int[] unit : block) {
            board[unit[0] + y][unit[1] + x] = abjad;
        }
    }

    // Cek apakah papan masih ada space kosong
    public boolean hasEmptySpace() {
        for (int row = 0; row < N; row++) {
            for (int unit = 0; unit < M; unit++) {
                if (board[row][unit] == '0') {
                    return true;
                }
            }
        }
        return false;
    }


}
```

Kelas Block

```java
import java.util.ArrayList;
import java.util.List;

public class Block {
    private char abjad;
    private List<int[]> block;

    public Block(char abjad, List<int[]> block) {
        this.abjad = abjad;
        this.block = normalizeBlockCoords(block);
    }

    // Getter
    public char getAbjad() {
        return abjad;
    }

    public List<int[]> getBlock() {
        return block;
    }

    // Normalisasi koordinat blok ke titik (0,0) dari transformasi blok
    private List<int[]> normalizeBlockCoords(List<int[]> block) {
        int minX = 99999;
        int minY = 99999;

        for (int[] blockCoord : block) {
            minY = Math.min(minX, blockCoord[0]);
            minX = Math.min(minY, blockCoord[1]);
        }

        List<int[]> normalizedBlockCoords = new ArrayList<>();
        for (int[] blockCoord : block) {
            normalizedBlockCoords.add(new int[]{blockCoord[0] - minY, blockCoord[1] - minX});
        }
        return normalizedBlockCoords;
    }

    // Rotasi koordinat blok 90 derajat searah jarum jam
    private List<int[]> rotateBlock90(List<int[]> block) {
        List<int[]> rotatedBlockCoords = new ArrayList<>();
        for (int[] coord : block) {
            rotatedBlockCoords.add(new int[]{coord[1], -coord[0]}); // (y,x) -> (x,-y)
        }
        return normalizeBlockCoords(rotatedBlockCoords);
    }

    private List<int[]> mirrorYBlock(List<int[]> block) {
        List<int[]> mirroredBlockCoords = new ArrayList<>();

        int maxY = -99999;

        for (int[] blockCoord : block) {
            maxY = Math.max(maxY, blockCoord[0]);
        }
```

```
        for (int[] coord : block) {
            mirroredBlockCoords.add(new int[]{maxY - coord[0], coord[1]});
        }
        return normalizeBlockCoords(mirroredBlockCoords);
    }

    private List<int[]> mirrorXBlock(List<int[]> block) {
        List<int[]> mirroredBlockCoords = new ArrayList<>();

        int maxX = -99999;

        for (int[] blockCoord : block) {
            maxX = Math.max(maxX, blockCoord[1]);
        }
        for (int[] coord : block) {
            mirroredBlockCoords.add(new int[]{coord[0], maxX - coord[1]});
        }
        return normalizeBlockCoords(mirroredBlockCoords);
    }
```

```
    // Cek blok sudah ada di list transformasi block
    private boolean containSameBlock(List<List<int[]>> blockTransformations, List<int[]> newBlock) {
        for (List<int[]> blockTransformation : blockTransformations) {
            if (isBlockPositionSame(blockTransformation, newBlock)) {
                return true;
            }
        }
        return false;
    }

    // Rotasi blok 0, 90, 180, 270 derajat
    private void allBlockRotation(List<List<int[]>> blockTransformations, List<int[]> currentBlock) {
        for (int i = 0; i < 4; i++) {
            if (!containSameBlock(blockTransformations, currentBlock)) {
                blockTransformations.add(new ArrayList<>(currentBlock));
            }
            currentBlock = rotateBlock90(currentBlock);
        }

    }
```

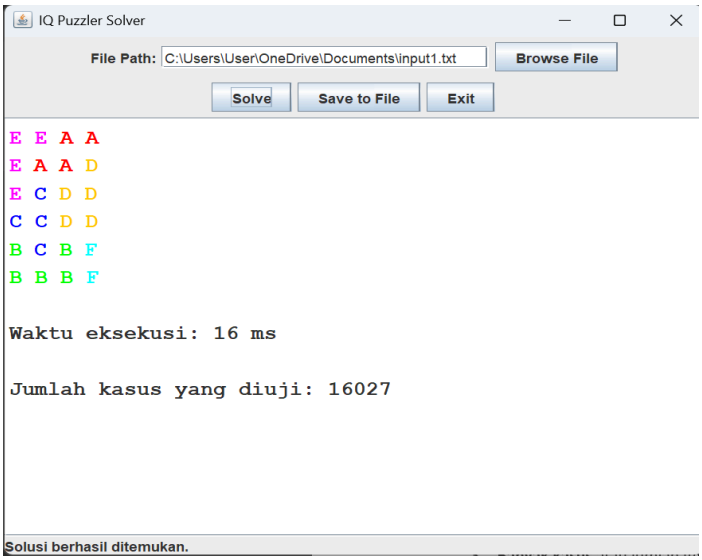
```
    public List<List<int[]>> getBlockTransformations() {
        List<List<int[]>> blockTransformations = new ArrayList<>();

        List<int[]> mirrorX = mirrorXBlock(block);
        List<int[]> mirrorY = mirrorYBlock(block);
        List<int[]> mirrorXY = mirrorXBlock(mirrorY);

        allBlockRotation(blockTransformations, block);

        // Transformasi untuk kemungkinan mirror block
        if(!isBlockPositionSame(block, mirrorX)) {
            allBlockRotation(blockTransformations, mirrorX);
        }
        if(!isBlockPositionSame(block, mirrorY)) {
            allBlockRotation(blockTransformations, mirrorY);
        }
        if(!isBlockPositionSame(block, mirrorXY)) {
            allBlockRotation(blockTransformations, mirrorXY);
        }

        return blockTransformations;
    }

}
```

BAGIAN III
SCREENSHOT HASIL TEST

3.1 Input file: input1.txt

```
6 4 6
DEFAULT
 AA
AA
B B
BBB
 C
CCC
D
DD
DD
E
EEE
FF
```

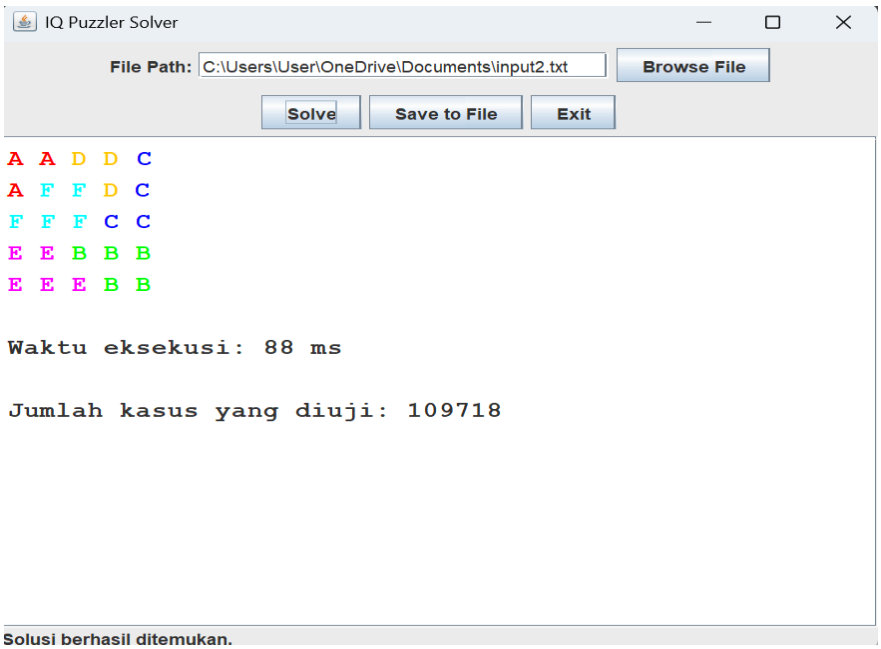
Hasil :



3.2 Input file: input2.txt

```
5 5 6
DEFAULT
BB
BBB
CC
 C
 C
A
AA
D
DD
FFF
FF
EE
EE
E
```

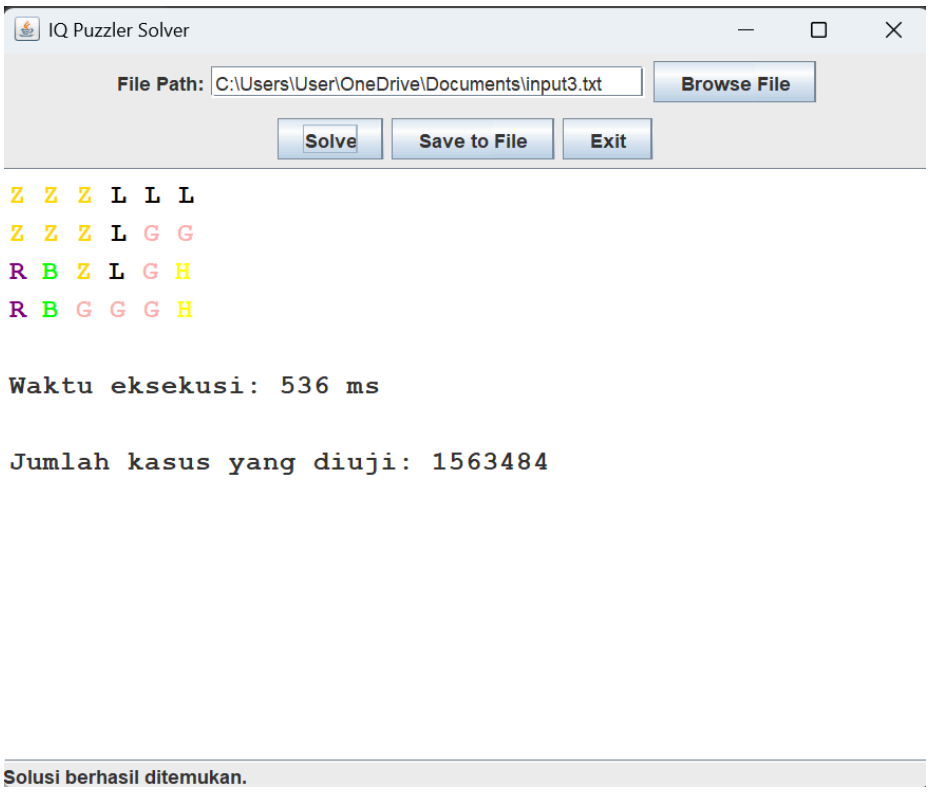
Hasil :



3.3 Input file: input3.txt

```
4 6 6
DEFAULT
Z
ZZZ
ZZZ
  L
  L
LLL
R
R
GGG
  G
  GG
HH
B
B
```

Hasil:



3.4 Input file: input4.txt

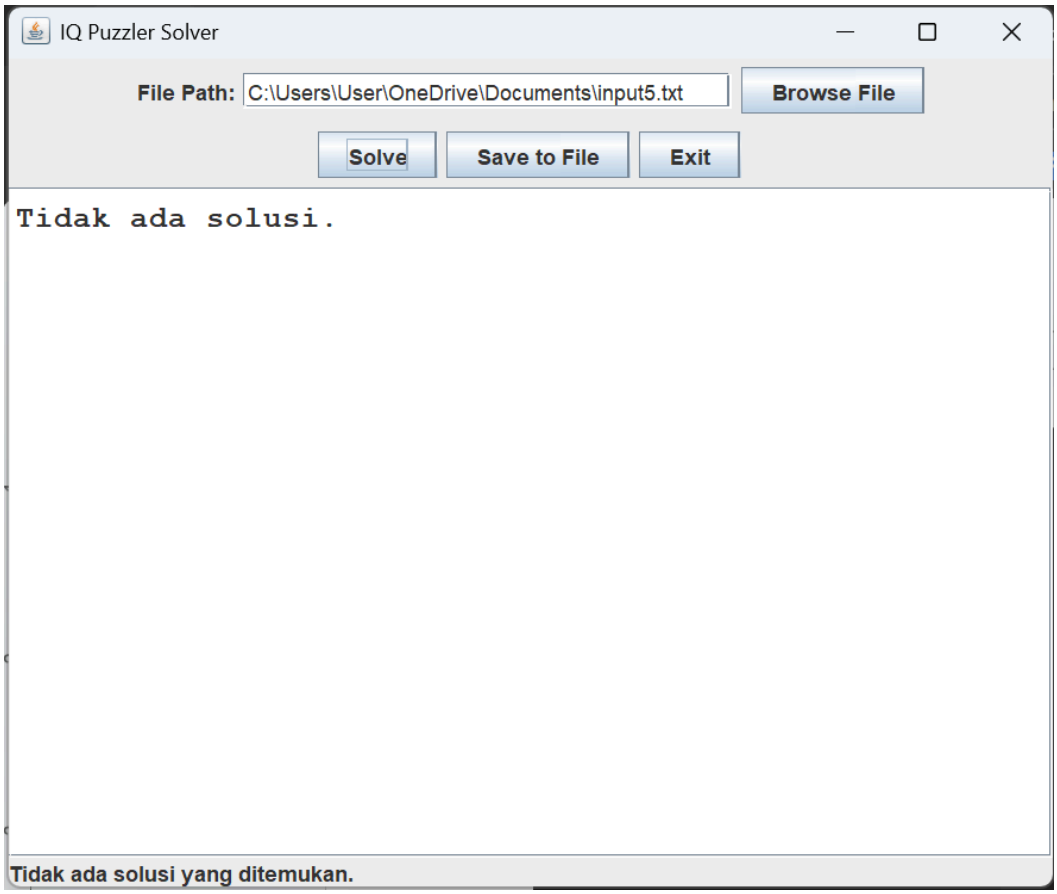
```
5 7 8
DEFAULT
FFF
F
FF
A
A
A
HHHH
H H
H
DDD
 DD
YYY
KK
K
KK
 R
RRR
O
O
```

Hasil :

3.5 Input file: input5.txt

```
4 4 5
DEFAULT
 AA
AA
A
CCC
C
DD
 D
HHH
B
B
```

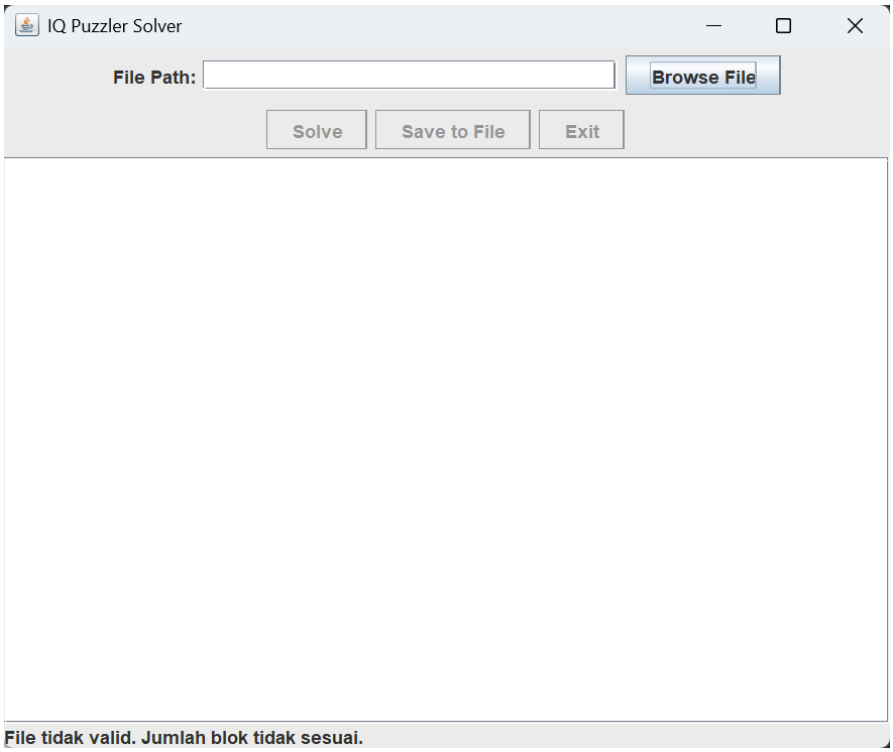
Hasil :



3.6 Input file: input6.txt

```
4 5 6
DEFAULT
JJJJ
H
HHH
H
C
CCC
A
B
B
GGG
D
DDD
```

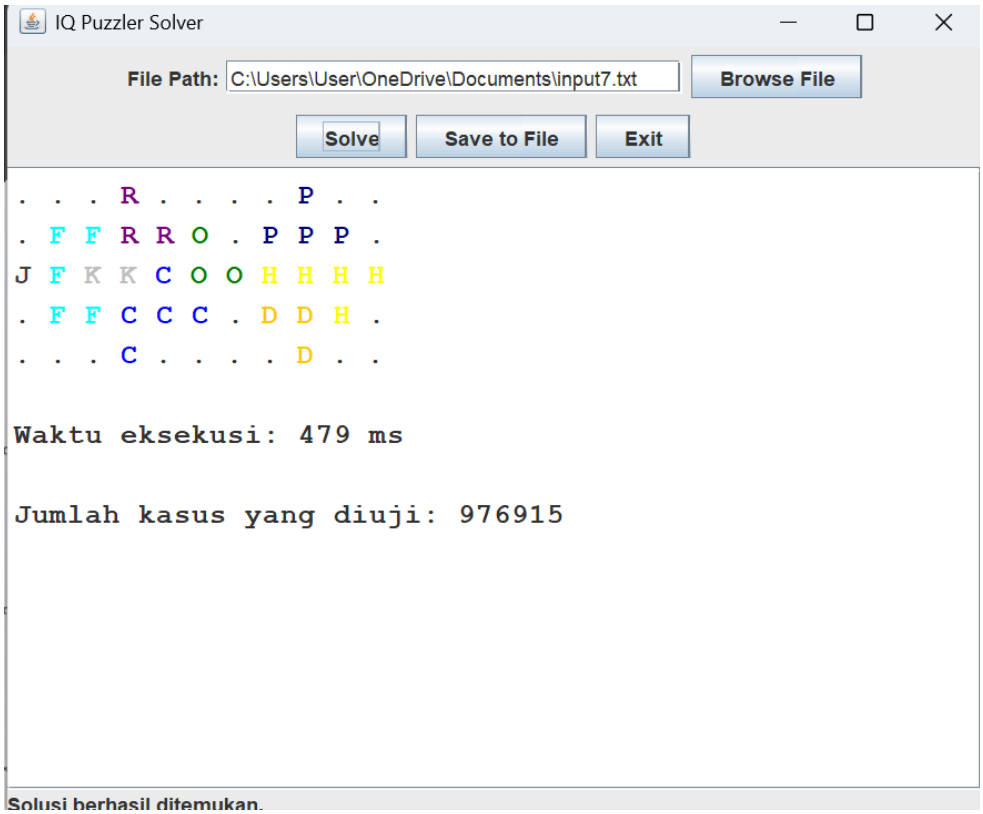
Hasil :



3.7 Input file: input7.txt

```
5 11 9
CUSTOM
...X....X..
.XXXXX.XXX.
XXXXXXXXXXX
.XXXXX.XXX.
...X....X..
 C
CC
 CC
 H
HHHH
FF
F
FF
OO
O
J
KK
PPP
 P
D
DD
RR
R
```

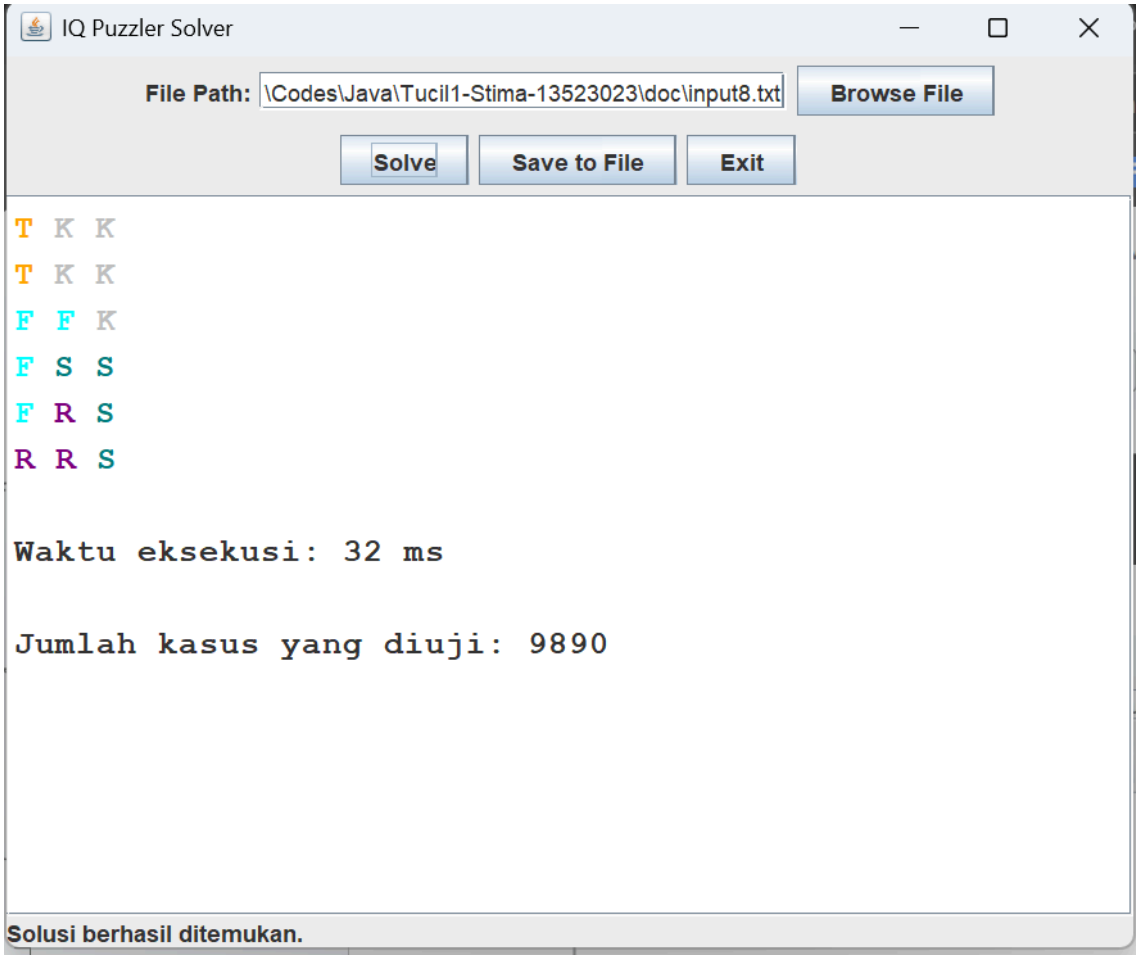
Hasil :



3.8 Input file: input8.txt

```
6 3 5
DEFAULT
F
FFF
K
KK
KK
R
RR
SSS
  S
TT
```

Hasil :



LINK REPOSITORY

https://github.com/AgungLucker/Tucil1_13523023

TABEL CHECKLIST

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	✓	
2	Program berhasil dijalankan	✓	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4	Program dapat membaca masukan berkas .txt serta menyimpan solusi dalam berkas .txt	✓	
5	Program memiliki Graphical User Interface (GUI)	✓	
6	Program dapat menyimpan solusi dalam bentuk file gambar		✓
7	Program dapat menyelesaikan kasus konfigurasi custom	✓	
8	Program dapat menyelesaikan kasus konfigurasi Piramida (3D)		✓
9	Program dibuat oleh saya sendiri	✓	