LAPORAN TUGAS KECIL 01
IF2211 STRATEGI ALGORITMA

Penyelesaian **IQ Puzzler Pro** dengan Algoritma Brute Force

Disusun oleh:

Danendra Shafi Athallah

13523136

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG 40132

2024

BAB I

ALGORITMA BRUTE FORCE

Berdasarkan materi kuliah IF2211 Strategi Algoritma, algoritma *brute force* merupakan pendekatan penyelesaian masalah yang bersifat straightforward (langsung). Algoritma ini memecahkan persoalan secara sangat sederhana, langsung, dan dengan cara yang jelas. Seperti dijelaskan dalam materi, algoritma *brute force* dapat langsung ditulis berdasarkan pernyataan dalam persoalan atau definisi konsep yang terlibat. Algoritma *brute force* memiliki beberapa karakteristik utama. Pertama, kesederhanaan dalam implementasi dan pemahaman. Kedua, kemampuan untuk selalu menemukan solusi (jika ada) karena memeriksa seluruh kemungkinan. Ketiga, pendekatan ini umumnya kurang efisien untuk masalah berukuran besar karena kompleksitasnya yang tinggi, seringkali eksponensial atau faktorial.

Dalam konteks penyelesaian IQ Puzzler Pro, algoritma *brute force* diterapkan dengan mencoba secara sistematis semua kemungkinan penempatan blok puzzle pada papan. Setiap blok puzzle dicoba pada setiap posisi yang mungkin, dengan setiap orientasi yang mungkin (rotasi dan pencerminan), hingga ditemukan konfigurasi yang memenuhi kriteria penyelesaian puzzle. Pendekatan exhaustive search ini menjamin ditemukannya solusi jika memang ada, meskipun mungkin membutuhkan waktu eksekusi yang cukup lama untuk puzzle dengan dimensi besar. Algoritma *brute force* yang diimplementasikan untuk penyelesaian IQ Puzzler Pro menggunakan pendekatan rekursif dengan backtracking untuk mencoba semua kemungkinan penempatan blok puzzle. Pendekatan ini memastikan bahwa semua kemungkinan solusi dieksplorasi secara sistematis hingga solusi ditemukan atau semua kemungkinan telah dievaluasi. Berikut adalah langkah-langkah detail dari algoritma yang diimplementasikan:

1. Inisialisasi dimulai dengan membaca input dari file berupa dimensi papan (N x M), jumlah blok puzzle (P), jenis kasus (DEFAULT/CUSTOM/PYRAMID), dan bentuk blok puzzle. Kemudian, papan kosong berukuran N x M dipersiapkan dan semua blok puzzle diproses dan disimpan dalam struktur data yang sesuai. Semua orientasi yang mungkin (rotasi dan pencerminan) untuk setiap blok puzzle juga dihitung pada tahap ini.
2. Selanjutnya, algoritma mencoba menempatkan blok puzzle satu per satu pada papan. Setiap blok dicoba pada setiap posisi yang mungkin dengan setiap orientasi yang mungkin. Proses ini menghasilkan jumlah iterasi yang sangat besar, yang merupakan karakteristik utama dari pendekatan *brute force*. Secara spesifik, algoritma mencoba menempatkan blok puzzle satu per satu pada papan dengan metode rekursif. Untuk setiap blok puzzle yang akan ditempatkan, algoritma mencoba semua kemungkinan yaitu semua posisi pada papan (i,j) dari (0,0) hingga (N-1, M-1) dan semua orientasi (rotasi dan pencerminan) yang mungkin untuk blok tersebut.
3. Lalu dilakukan pengecekan apakah suatu blok dengan orientasi tertentu dapat ditempatkan pada posisi (row, col). Untuk kasus DEFAULT, program memeriksa apakah blok tidak keluar dari batas papan dan tidak tumpang tindih dengan blok lain. Untuk kasus CUSTOM, terdapat pengecekan tambahan bahwa blok harus menutupi setidaknya satu target ('X') dan tidak menutupi sel non-target.
4. Setelah itu dicek kelengkapan dengan memeriksa apakah kondisi penyelesaian sudah terpenuhi. Untuk kasus DEFAULT, semua sel pada papan harus terisi. Untuk kasus CUSTOM, semua sel target ('X') harus terisi.
5. Program menghitung jumlah iterasi yang dilakukan oleh algoritma dan mengukur waktu eksekusi algoritma. Jumlah iterasi yang besar dalam pencarian *brute force* ini mencerminkan kompleksitas dengan solusi yang komprehensif.
6. Setelah solusi ditemukan melalui pencarian *brute force* yang menyeluruh, program menampilkan solusi dengan warna berbeda untuk setiap blok puzzle. Untuk kasus GUI, papan dan blok puzzle divisualisasikan dengan warna berbeda.

BAB II

SOURCE CODE PROGRAM

2.1. Library

Dalam pengembangan program IQ Puzzler Pro Solver yang mengimplementasikan algoritma *brute force,* sejumlah library digunakan untuk mendukung berbagai fungsionalitas yang diperlukan. Implementasi ini membutuhkan dukungan untuk operasi input/output, penanganan struktur data, pengukuran waktu, serta pengembangan antarmuka grafis yang interaktif dan menarik. Beberapa library standar Java dimanfaatkan untuk memenuhi kebutuhan tersebut, sementara library tambahan digunakan untuk meningkatkan kemampuan visualisasi dan interaksi dengan pengguna. Berikut merupakan library yang digunakan pada program IQ Puzzler Pro Solver:

- **java.io** : Library ini menyediakan berbagai kelas untuk melakukan operasi input/output yang menjadi komponen penting dalam pembacaan file konfigurasi puzzle dan penyimpanan hasil solusi ke dalam format teks.
- **java.util** : Koleksi struktur data dan utilitas dari library ini menjadi fondasi dalam pengorganisasian blok puzzle, representasi papan, dan pengelolaan berbagai operasi algoritma *brute force*.
- **java.time** : Pengukuran performa algoritma menjadi aspek penting dalam evaluasi solusi, dan library ini menyediakan mekanisme yang presisi untuk mengukur durasi eksekusi algoritma.
- **java.awt** : Untuk visualisasi dan implementasi antarmuka grafis, library ini menyediakan komponen dasar yang mendukung representasi visual papan dan blok puzzle.
- **javax.swing** : Library GUI ini menawarkan komponen antarmuka pengguna yang lebih canggih dan konsisten antar platform, memungkinkan interaksi yang lebih intuitif.
- **Java.nio :** Library ini digunakan untuk menangani operasi file dengan pendekatan yang lebih modern dan efisien.
- **javafx** Framework GUI untuk pengembangan aplikasi desktop dengan dukungan untuk berbagai komponen antarmuka dan efek visual yang menarik.
- **javax.imageio.ImageIO :** Fungsionalitas ekspor gambar menjadi komponen bonus dalam spesifikasi tugas, dan library ini mendukung kemampuan tersebut.
- **javafx.embed.swing.SwingFXUtils** : Library untuk integrasi antara komponen JavaFX dan Swing, memungkinkan konversi antara kedua framework UI tersebut.
- **javafx.scene.image.WritableImage** : Dimanfaatkan untuk representasi visual yang dapat dimanipulasi dan disimpan sebagai file gambar.
- **chrono** : Kemampuan pengukuran waktu dengan presisi tinggi merupakan kontribusi penting dari library ini, yang membantu evaluasi kinerja algoritma.

2.2. Program Utama

Berikut merupakan beberapa program dengan fokus pada pengaplikasian *brute force* untuk IQ Puzzler Pro Solver :

2.2.1. Main.java

```java
package src;

import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import javafx.application.Application;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("IQ Puzzler Pro Solver");
        System.out.print("Choose view mode (1: Terminal, 2: GUI): ");

        int choice = scanner.nextInt();
        scanner.nextLine();

        if (choice == 1) {
            runTerminalMode(scanner);
        } else if (choice == 2) {
            Application.launch(GUI.class, args);
        } else {
            System.out.println("Invalid choice!");
        }

        scanner.close();
    }

    private static void runTerminalMode(Scanner scanner) {
        try {
            System.out.print("Enter file path: ");
            String filePath = scanner.nextLine().trim();
            File file = new File(filePath);

            if (!file.exists()) {
                System.out.println("Error: File not found");
                return;
            }

            Board board = Board.fromFile(file);
            Solver solver = new Solver(board);
            solver.setDebugMode(true);
            System.out.println("\nSolving puzzle...");
            long startTime = System.currentTimeMillis();
            boolean solved = solver.solve();
            long endTime = System.currentTimeMillis();

            System.out.println("\nResults:");
```

```
            System.out.println("=========");
            if (solved) {
                System.out.println("\nSolution found:");
                board.print();
            } else {
                System.out.println("No solution exists.");
            }

              System.out.println("\nWaktu pencarian: " + (endTime - startTime) + " ms");
                          System.out.println("Banyak kasus yang ditinjau: " + solver.getIterationCount());

             System.out.print("\nApakah anda ingin menyimpan solusi? (ya/tidak): ");
            String save = scanner.nextLine().trim().toLowerCase();
            if (save.startsWith("y")) {
                System.out.print("Enter output file path: ");
                String outputPath = scanner.nextLine().trim();
                board.saveToFile(outputPath);
                System.out.println("Solution saved to: " + outputPath);
            }

        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

2.2.2. Board.java

```
package src;

import java.io.*;
import java.util.*;

public class Board {
    private final int rows;
    private final int cols;
    private final char[][] grid;
    private final List<Piece> pieces;
    private String puzzleType;
    private char[][] targetConfig;
```

```
13.             private   static   final   Set<String>   VALID_PUZZLE_TYPES   =   new
    HashSet<>(Arrays.asList("DEFAULT", "CUSTOM", "PYRAMID"));
14.
15.     // ANSI color buat terminal
16.     private static final String[] COLORS = {
17.         "\u001B[31m", // Red
18.         "\u001B[32m", // Green
19.         "\u001B[33m", // Yellow
20.         "\u001B[34m", // Blue
21.         "\u001B[35m", // Magenta
22.         "\u001B[36m", // Cyan
23.         "\u001B[91m", // Bright Red
24.         "\u001B[92m", // Bright Green
25.         "\u001B[93m", // Bright Yellow
26.         "\u001B[94m", // Bright Blue
27.         "\u001B[95m", // Bright Magenta
28.         "\u001B[96m", // Bright Cyan
29.         "\u001B[41m", // Red Background
30.         "\u001B[42m", // Green Background
31.         "\u001B[43m", // Yellow Background
32.         "\u001B[44m", // Blue Background
33.         "\u001B[45m", // Magenta Background
34.         "\u001B[46m", // Cyan Background
35.         "\u001B[101m", // Bright Red Background
36.         "\u001B[102m", // Bright Green Background
37.         "\u001B[103m", // Bright Yellow Background
38.         "\u001B[104m", // Bright Blue Background
39.         "\u001B[105m", // Bright Magenta Background
40.         "\u001B[106m", // Bright Cyan Background
41.         "\u001B[90m", // Dark Gray
42.         "\u001B[97m"  // White
43.     };
44.     private static final String RESET = "\u001B[0m";
45.
46.     public Board(int rows, int cols) {
47.         if (rows <= 0 || cols <= 0) {
48.                 throw new IllegalArgumentException("Board dimensions must be
    positive");
49.         }
50.         this.rows = rows;
51.         this.cols = cols;
52.         this.grid = new char[rows][cols];
53.         this.pieces = new ArrayList<>();
54.         this.puzzleType = "DEFAULT";
55.         initializeGrid();
56.     }
57.
```

```
58.     private void initializeGrid() {
59.         for (int i = 0; i < rows; i++) {
60.             Arrays.fill(grid[i], '.');
61.         }
62.     }
63.
64.     public static Board fromFile(File file) throws IOException {
65.          try (BufferedReader reader = new BufferedReader(new FileReader(file)))
    {
66.             String[] dimensions = reader.readLine().trim().split("\\s+");
67.             if (dimensions.length != 3) {
68.                     throw new IOException("First line must contain exactly 3
    numbers (N M P)");
69.             }
70.
71.             int rows = Integer.parseInt(dimensions[0]);
72.             int cols = Integer.parseInt(dimensions[1]);
73.             int expectedPieces = Integer.parseInt(dimensions[2]);
74.
75.             if (expectedPieces > 26) {
76.                 throw new IOException("Number of pieces cannot exceed 26");
77.             }
78.
79.             String puzzleType = reader.readLine().trim();
80.             if (!VALID_PUZZLE_TYPES.contains(puzzleType)) {
81.                 throw new IOException("Invalid puzzle type: " + puzzleType);
82.             }
83.
84.             Board board = new Board(rows, cols);
85.             board.puzzleType = puzzleType;
86.
87.             if ("CUSTOM".equals(puzzleType)) {
88.                 board.targetConfig = new char[rows][cols];
89.                 for (int i = 0; i < rows; i++) {
90.                     String line = reader.readLine();
91.                     if (line == null || line.length() < cols) {
92.                             throw new IOException("Invalid custom configuration
    data");
93.                     }
94.                     System.out.println("Reading target config line: " + line);
95.                     for (int j = 0; j < cols; j++) {
96.                         board.targetConfig[i][j] = line.charAt(j);
97.                     }
98.                 }
99.             }
100.
101.            List<String> pieceInput = new ArrayList<>();
```

```
            String line;
            while ((line = reader.readLine()) != null) {
                pieceInput.add(line);
            }

            List<List<String>> piecesData = new ArrayList<>();
            List<String> currentPiece = new ArrayList<>();
            char currentPieceId = 0;

            for (String pieceLine : pieceInput) {
                if (pieceLine.trim().isEmpty()) {
                    if (!currentPiece.isEmpty()) {
                        piecesData.add(new ArrayList<>(currentPiece));
                        currentPiece.clear();
                        currentPieceId = 0;
                    }
                    continue;
                }
                char firstChar = 0;
                for (char c : pieceLine.toCharArray()) {
                    if (Character.isLetter(c)) {
                        firstChar = c;
                        break;
                    }
                }

                if (firstChar != 0) {
                       if (currentPieceId != 0 && firstChar != currentPieceId &&
    !currentPiece.isEmpty()) {
                        piecesData.add(new ArrayList<>(currentPiece));
                        currentPiece.clear();
                    }
                    currentPieceId = firstChar;
                    currentPiece.add(pieceLine);
                } else if (!currentPiece.isEmpty()) {
                    currentPiece.add(pieceLine);
                }
            }

            if (!currentPiece.isEmpty()) {
                piecesData.add(currentPiece);
            }

            for (List<String> pieceData : piecesData) {
                board.pieces.add(new Piece(pieceData));
            }

```

```
148.             if (board.pieces.size() != expectedPieces) {
149.                 throw new IOException("Expected " + expectedPieces + " pieces
    but found " + board.pieces.size());
150.             }
151.
152.             System.out.println("Pieces read from file:");
153.             for (Piece piece : board.pieces) {
154.                 System.out.println("Piece " + piece.getIdentifier() + ":");
155.                 System.out.println(piece.toString());
156.             }
157.
158.             return board;
159.         } catch (NumberFormatException e) {
160.             throw new IOException("Invalid number format in dimensions");
161.         }
162.     }
163.
164.     public boolean placePiece(Piece piece, int row, int col) {
165.         if (!canPlacePiece(piece, row, col)) {
166.             return false;
167.         }
168.
169.         char[][] shape = piece.getShape();
170.         for (int i = 0; i < shape.length; i++) {
171.             for (int j = 0; j < shape[0].length; j++) {
172.                 if (shape[i][j] == piece.getIdentifier()) {
173.                     grid[row + i][col + j] = shape[i][j];
174.                 }
175.             }
176.         }
177.         return true;
178.     }
179.
180.     public void removePiece(Piece piece, int row, int col) {
181.         char[][] shape = piece.getShape();
182.         for (int i = 0; i < shape.length; i++) {
183.             for (int j = 0; j < shape[0].length; j++) {
184.                 if (shape[i][j] == piece.getIdentifier()) {
185.                     grid[row + i][col + j] = '.';
186.                 }
187.             }
188.         }
189.     }
190.
191.     public boolean canPlacePiece(Piece piece, int row, int col) {
192.         char[][] shape = piece.getShape();
```

```
193.             if (row < 0 || col < 0 || row + shape.length > rows || col +
    shape[0].length > cols) {
194.             return false;
195.         }
196.
197.         for (int i = 0; i < shape.length; i++) {
198.             for (int j = 0; j < shape[0].length; j++) {
199.                 if (shape[i][j] == piece.getIdentifier()) {
200.                     if (row + i >= rows || col + j >= cols) {
201.                         return false;
202.                     }
203.                     if (grid[row + i][col + j] != '.') {
204.                         return false;
205.                     }
206.                 }
207.             }
208.         }
209.         return true;
210.     }
211.
212.     public boolean isValidForCustom() {
213.         if (!"CUSTOM".equals(puzzleType) || targetConfig == null) {
214.             return isComplete();
215.         }
216.
217.         for (int i = 0; i < rows; i++) {
218.             for (int j = 0; j < cols; j++) {
219.                 if (targetConfig[i][j] == 'X' && grid[i][j] == '.') {
220.                     return false;
221.                 }
222.                 if (targetConfig[i][j] == '.' && grid[i][j] != '.') {
223.                     return false;
224.                 }
225.             }
226.         }
227.         return true;
228.     }
229.
230.     public boolean isComplete() {
231.         if ("CUSTOM".equals(puzzleType)) {
232.             return isValidForCustom();
233.         }
234.
235.         for (int i = 0; i < rows; i++) {
236.             for (int j = 0; j < cols; j++) {
237.                 if (grid[i][j] == '.') {
238.                     return false;
```

```
                }
            }
        }
        return true;
    }

    public void print() {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] != '.') {
                    int colorIndex = grid[i][j] - 'A';
                        System.out.print(COLORS[colorIndex % COLORS.length] +
    grid[i][j] + RESET);
                } else {
                    System.out.print(grid[i][j]);
                }
            }
            System.out.println();
        }
    }

    public void saveToFile(String filepath) throws IOException {
        try (PrintWriter writer = new PrintWriter(new FileWriter(filepath))) {
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                    writer.print(grid[i][j]);
                }
                writer.println();
            }
        }
    }

    public int getRows() { return rows; }
    public int getCols() { return cols; }
    public char[][] getGrid() { return grid; }
                        public     List<Piece>     getPieces()     {     return
    Collections.unmodifiableList(pieces); }
    public String getPuzzleType() { return puzzleType; }
    public char[][] getTargetConfig() { return targetConfig; }

    public void setPuzzleType(String type) {
        if (!VALID_PUZZLE_TYPES.contains(type)) {
            throw new IllegalArgumentException("Invalid puzzle type: " + type);
        }
        this.puzzleType = type;
    }
}
```

2.2.3. Piece.java

```java
package src;
import java.util.*;

public class Piece {
    private char[][] shape;
    private final char identifier;
    private int rows;
    private int cols;

    public Piece(List<String> lines) {
        if (lines == null || lines.isEmpty()) {
            throw new IllegalArgumentException("Empty piece definition");
        }

        this.identifier = findIdentifier(lines);
        List<String> normalizedLines = new ArrayList<>();
        for (String line : lines) {
            line = line.replace(' ', '.');
            normalizedLines.add(line);
        }

        int maxCols = 0;
        for (String line : normalizedLines) {
            maxCols = Math.max(maxCols, line.length());
        }

        this.rows = normalizedLines.size();
        this.cols = maxCols;
        this.shape = new char[rows][cols];

        for (int i = 0; i < rows; i++) {
            Arrays.fill(shape[i], '.');
        }

        for (int i = 0; i < rows; i++) {
            String line = normalizedLines.get(i);
            for (int j = 0; j < line.length(); j++) {
                if (line.charAt(j) == identifier) {
                    shape[i][j] = identifier;
                }
            }
        }

```

```
44.         trimEmptySpace();
45.     }
46.
47.     private char findIdentifier(List<String> lines) {
48.         for (String line : lines) {
49.             for (char c : line.toCharArray()) {
50.                 if (c != '.' && c != ' ') {
51.                     return c;
52.                 }
53.             }
54.         }
55.         throw new IllegalArgumentException("No identifier found in piece");
56.     }
57.
58.     private void trimEmptySpace() {
59.         int minRow = rows, maxRow = -1;
60.         int minCol = cols, maxCol = -1;
61.         for (int i = 0; i < rows; i++) {
62.             for (int j = 0; j < cols; j++) {
63.                 if (shape[i][j] == identifier) {
64.                     minRow = Math.min(minRow, i);
65.                     maxRow = Math.max(maxRow, i);
66.                     minCol = Math.min(minCol, j);
67.                     maxCol = Math.max(maxCol, j);
68.                 }
69.             }
70.         }
71.
72.         if (maxRow < minRow || maxCol < minCol) {
73.                     throw new IllegalArgumentException("Invalid piece: no cells
    found");
74.         }
75.         rows = maxRow - minRow + 1;
76.         cols = maxCol - minCol + 1;
77.         char[][] trimmed = new char[rows][cols];
78.
79.         for (int i = 0; i < rows; i++) {
80.             for (int j = 0; j < cols; j++) {
81.                 trimmed[i][j] = shape[i + minRow][j + minCol];
82.             }
83.         }
84.
85.         this.shape = trimmed;
86.     }
87.
88.     public List<Piece> getAllOrientations() {
89.         Set<String> seen = new HashSet<>();
```

```
90.         List<Piece> orientations = new ArrayList<>();
91.         Piece current = this;
92.         for (int rot = 0; rot < 4; rot++) {
93.             addUniqueOrientation(current, orientations, seen);
94.             Piece flipped = current.flip();
95.             addUniqueOrientation(flipped, orientations, seen);
96.             current = current.rotate();
97.         }
98.
99.         return orientations;
100.        }
101.
102.        private void addUniqueOrientation(Piece piece, List<Piece> orientations,
    Set<String> seen) {
103.            String key = piece.toString();
104.            if (!seen.contains(key)) {
105.                seen.add(key);
106.                orientations.add(piece);
107.            }
108.        }
109.
110.        public Piece rotate() {
111.            char[][] rotated = new char[cols][rows];
112.            for (int i = 0; i < rows; i++) {
113.                for (int j = 0; j < cols; j++) {
114.                    rotated[j][rows - 1 - i] = shape[i][j];
115.                }
116.            }
117.            return new Piece(Arrays.asList(shapeToStrings(rotated)));
118.        }
119.
120.        public Piece flip() {
121.            char[][] flipped = new char[rows][cols];
122.            for (int i = 0; i < rows; i++) {
123.                for (int j = 0; j < cols; j++) {
124.                    flipped[i][cols - 1 - j] = shape[i][j];
125.                }
126.            }
127.            return new Piece(Arrays.asList(shapeToStrings(flipped)));
128.        }
129.
130.        private String[] shapeToStrings(char[][] shape) {
131.            String[] result = new String[shape.length];
132.            for (int i = 0; i < shape.length; i++) {
133.                StringBuilder sb = new StringBuilder();
134.                for (int j = 0; j < shape[i].length; j++) {
135.                    sb.append(shape[i][j] == 0 ? identifier : shape[i][j]);
```

```
                }
                result[i] = sb.toString();
            }
            return result;
        }

        public char[][] getShape() {
            return shape;
        }

        public char getIdentifier() {
            return identifier;
        }

        public int getRows() {
            return rows;
        }

        public int getCols() {
            return cols;
        }

        @Override
        public String toString() {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                    sb.append(shape[i][j]);
                }
                if (i < rows - 1) sb.append('\n');
            }
            return sb.toString();
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;
            Piece piece = (Piece) o;
            return Arrays.deepEquals(shape, piece.shape);
        }

        @Override
        public int hashCode() {
            return Arrays.deepHashCode(shape);
        }
    }
```

2.2.4. Solver.java

```java
package src;

import javafx.application.Platform;
import javafx.scene.layout.GridPane;
import java.util.*;
import java.util.concurrent.atomic.AtomicBoolean;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

public class Solver {
    private final Board board;
    private final GridPane boardGrid;
    private long iterationCount;
    private long startTime;
    private long delayMs = 0;
    private final AtomicBoolean isSolving;
    private List<SolverListener> listeners;
    private boolean debugMode = false;
    private boolean showSteps = false;
    private char[][] targetConfig;
    private Set<Character> usedPieces;
    private int targetCells = 0;

    public interface SolverListener {
        void onIterationComplete(long iteration, String message);
        void onSolutionFound(boolean found, long iterations, long timeTaken);
    }

    public Solver(Board board) {
        this(board, null);
    }

    public Solver(Board board, GridPane boardGrid) {
        this.board = board;
        this.boardGrid = boardGrid;
        this.iterationCount = 0;
        this.startTime = 0;
        this.isSolving = new AtomicBoolean(false);
        this.listeners = new ArrayList<>();
        this.usedPieces = new HashSet<>();

        if ("CUSTOM".equals(board.getPuzzleType())) {
            this.targetConfig = board.getTargetConfig();
```

```
            // Count target cells
            if (targetConfig != null) {
                for (int i = 0; i < board.getRows(); i++) {
                    for (int j = 0; j < board.getCols(); j++) {
                        if (targetConfig[i][j] == 'X') {
                            targetCells++;
                        }
                    }
                }
                System.out.println("Target cells count: " + targetCells);
            }
        }
    }

    public boolean solve() {
        if (isSolving.get()) return false;

        isSolving.set(true);
        startTime = System.currentTimeMillis();
        iterationCount = 0;
        usedPieces.clear();
        List<Piece> pieces = new ArrayList<>(board.getPieces());
        boolean result = tryAllPieces(pieces, 0);
        long timeTaken = System.currentTimeMillis() - startTime;
        notifyListeners(result, iterationCount, timeTaken);
        isSolving.set(false);

        return result;
    }

     private void notifyListeners(boolean solutionFound, long iterations, long
  timeTaken) {
        for (SolverListener listener : listeners) {
            listener.onSolutionFound(solutionFound, iterations, timeTaken);
        }
    }

    private boolean tryAllPieces(List<Piece> pieces, int currentIndex) {
        if (!isSolving.get()) return false;
        if (currentIndex >= pieces.size()) {
            return isComplete();
        }
        Piece currentPiece = pieces.get(currentIndex);
        if (usedPieces.contains(currentPiece.getIdentifier())) {
            return tryAllPieces(pieces, currentIndex + 1);
        }

```

```
90.         List<int[]> positions = getAllPositions();
91.         List<Piece> orientations = currentPiece.getAllOrientations();
92.
93.         for (int[] pos : positions) {
94.             int row = pos[0];
95.             int col = pos[1];
96.
97.             for (Piece orientation : orientations) {
98.                 iterationCount++;
99.
100.                    if (iterationCount % 100000 == 0 && debugMode) {
101.                            debugPrint("Iteration " + iterationCount + ": Trying
    piece " + orientation.getIdentifier() + " at (" + row + "," + col + ")");
102.                    }
103.
104.                    if (canPlacePiece(orientation, row, col)) {
105.                        board.placePiece(orientation, row, col);
106.                        usedPieces.add(orientation.getIdentifier());
107.
108.                        if (showSteps) {
109.                            updateBoardDisplay();
110.                        }
111.
112.                        if (tryAllPieces(pieces, currentIndex + 1)) {
113.                            return true;
114.                        }
115.
116.                        board.removePiece(orientation, row, col);
117.                        usedPieces.remove(orientation.getIdentifier());
118.                    }
119.
120.                    if (!isSolving.get()) return false;
121.                }
122.            }
123.
124.            return false;
125.        }
126.
127.        private void debugPrint(String message) {
128.            if (debugMode) {
129.                System.out.println("[DEBUG] " + message);
130.                for (SolverListener listener : listeners) {
131.                    listener.onIterationComplete(iterationCount, message);
132.                }
133.            }
134.        }
135.
```

```
136.        private List<int[]> getAllPositions() {
137.            List<int[]> positions = new ArrayList<>();
138.            for (int i = 0; i < board.getRows(); i++) {
139.                for (int j = 0; j < board.getCols(); j++) {
140.                    positions.add(new int[]{i, j});
141.                }
142.            }
143.            return positions;
144.        }
145.
146.        // cek apakah piece dapat ditempatkan pada posisi (row, col)
147.        private boolean canPlacePiece(Piece piece, int row, int col) {
148.              // cek apakah piece dapat ditempatkan secara tidak tumpang tindih
    atau keluar batas
149.            if (!board.canPlacePiece(piece, row, col)) {
150.                return false;
151.            }
152.
153.                // kasus CUSTOM, periksa apakah piece menutupi setidaknya satu
    target
154.             if ("CUSTOM".equals(board.getPuzzleType()) && targetConfig != null)
    {
155.                char[][] shape = piece.getShape();
156.                boolean coversTarget = false;
157.
158.                for (int i = 0; i < shape.length; i++) {
159.                    for (int j = 0; j < shape[0].length; j++) {
160.                        if (shape[i][j] == piece.getIdentifier()) {
161.                            int boardRow = row + i;
162.                            int boardCol = col + j;
163.                            if (isValidPosition(boardRow, boardCol)) {
164.                                if (targetConfig[boardRow][boardCol] == 'X') {
165.                                    coversTarget = true;
166.                                } else if (targetConfig[boardRow][boardCol] ==
    '.') {
167.                                    // piece mengenai non-target ini tidak valid
    untuk kasus CUSTOM
168.                                    return false;
169.                                }
170.                            }
171.                        }
172.                    }
173.                }
174.                return coversTarget;
175.            }
176.
177.            return true;
```

```
178.        }
179.
180.        private boolean isValidPosition(int row, int col) {
181.              return row >= 0 && row < board.getRows() && col >= 0 && col <
    board.getCols();
182.        }
183.
184.        private boolean isComplete() {
185.             if ("CUSTOM".equals(board.getPuzzleType()) && targetConfig != null)
    {
186.                char[][] grid = board.getGrid();
187.                for (int i = 0; i < board.getRows(); i++) {
188.                    for (int j = 0; j < board.getCols(); j++) {
189.                        if (targetConfig[i][j] == 'X' && grid[i][j] == '.') {
190.                            return false;
191.                        }
192.                    }
193.                }
194.
195.                if (debugMode) {
196.                    System.out.println("Board state at completion check:");
197.                    board.print();
198.                }
199.
200.                return true;
201.            }
202.            return board.isComplete();
203.        }
204.
205.        private void updateBoardDisplay() {
206.            if (boardGrid == null || !showSteps) return;
207.
208.            Platform.runLater(() -> {
209.                boardGrid.getChildren().clear();
210.                char[][] grid = board.getGrid();
211.                for (int i = 0; i < board.getRows(); i++) {
212.                    for (int j = 0; j < board.getCols(); j++) {
213.                        Rectangle cell = createCell(grid[i][j]);
214.                        boardGrid.add(cell, j, i);
215.                    }
216.                }
217.            });
218.
219.            if (delayMs > 0) {
220.                try {
221.                    Thread.sleep(delayMs);
222.                } catch (InterruptedException e) {
```

```
                    Thread.currentThread().interrupt();
                }
            }
        }

        private Rectangle createCell(char value) {
            Rectangle cell = new Rectangle(40, 40);
            if (value == '.') {
                cell.setFill(Color.WHITE);
            } else {
                int colorIndex = value - 'A';
                cell.setFill(getColorForPiece(colorIndex));
            }
            cell.setStroke(Color.BLACK);
            cell.setStrokeWidth(0.5);
            return cell;
        }

        private Color getColorForPiece(int index) {
            Color[] colors = {
                Color.RED, Color.BLUE, Color.GREEN, Color.YELLOW,
                Color.PURPLE, Color.CYAN, Color.ORANGE, Color.PINK,
                Color.BROWN, Color.GRAY
            };
            return colors[index % colors.length];
        }

        public void addListener(SolverListener listener) {
            listeners.add(listener);
        }

        public void stop() {
            isSolving.set(false);
        }

        public long getIterationCount() {
            return iterationCount;
        }

        public void setDelay(long milliseconds) {
            this.delayMs = milliseconds;
        }

        public void setShowSteps(boolean enabled) {
            this.showSteps = enabled;
        }

```

```
270.        public void setDebugMode(boolean enabled) {
271.            this.debugMode = enabled;
272.        }
273.
274.        public long getElapsedTime() {
275.            return System.currentTimeMillis() - startTime;
276.        }
277.
278.        public boolean isRunning() {
279.            return isSolving.get();
280.        }
281.    }
```

2.2.5. GUI.java

```
1.  package src;
2.
3.  import javafx.application.Application;
4.  import javafx.application.Platform;
5.  import javafx.geometry.Insets;
6.  import javafx.geometry.Pos;
7.  import javafx.scene.Scene;
8.  import javafx.scene.control.*;
9.  import javafx.scene.layout.*;
10. import javafx.scene.paint.Color;
11. import javafx.scene.shape.Rectangle;
12. import javafx.stage.FileChooser;
13. import javafx.stage.Stage;
14. import javafx.scene.image.WritableImage;
15. import javafx.embed.swing.SwingFXUtils;
16. import javafx.scene.control.Alert.AlertType;
17. import javafx.scene.input.MouseEvent;
18. import javafx.scene.text.Text;
19. import javafx.beans.property.SimpleStringProperty;
20. import javafx.beans.property.StringProperty;
21.
22. import javax.imageio.ImageIO;
23. import java.io.File;
24. import java.util.logging.*;
25. import java.util.ArrayList;
26. import java.util.List;
27.
28. public class GUI extends Application implements Solver.SolverListener {
29.     private Board board;
30.     private GridPane boardGrid;
31.     private GridPane piecesPreviewGrid;
```

```
32.     private Label statusLabel;
33.     private Label timeLabel;
34.     private Label movesLabel;
35.     private Label iterationsLabel;
36.     private ToggleGroup modeGroup;
37.     private Slider speedSlider;
38.     private CheckBox debugModeCheckBox;
39.     private CheckBox showStepsCheckBox;
40.     private Button solveButton;
41.     private Button loadButton;
42.     private Button resetButton;
43.     private Button saveImageButton;
44.     private Button stopButton;
45.     private ProgressBar progressBar;
46.     private TextArea debugOutput;
47.     private double cellSize = 40;
48.     private List<Piece> availablePieces = new ArrayList<>();
49.                 private    StringProperty    currentModeProperty    =    new
    SimpleStringProperty("DEFAULT");
50.     private static final Logger logger = Logger.getLogger(GUI.class.getName());
51.     private Solver currentSolver;
52.
53.     @Override
54.      public void onSolutionFound(boolean found, long iterations, long timeTaken)
    {
55.         Platform.runLater(() -> {
56.                 statusLabel.setText(found ? "Solution found!" : "No solution
    exists");
57.             timeLabel.setText("Time: " + timeTaken + " ms");
58.             iterationsLabel.setText("Iterations: " + iterations);
59.         });
60.     }
61.
62.     @Override
63.     public void onIterationComplete(long iteration, String message) {
64.         Platform.runLater(() -> {
65.             iterationsLabel.setText("Iterations: " + iteration);
66.             debugOutput.appendText(message + "\n");
67.         });
68.     }
69.
70.     private void updateBoardDisplay() {
71.         if (board == null || boardGrid == null) return;
72.
73.         boardGrid.getChildren().clear();
74.         char[][] grid = board.getGrid();
75.
```

```
76.         for (int i = 0; i < board.getRows(); i++) {
77.             for (int j = 0; j < board.getCols(); j++) {
78.                 Rectangle cell = createBoardCell(grid[i][j], i, j);
79.                 boardGrid.add(cell, j, i);
80.             }
81.         }
82.     }
83.
84.     private Rectangle createBoardCell(char value, int row, int col) {
85.         Rectangle cell = new Rectangle(cellSize, cellSize);
86.         if (value == '.') {
87.             cell.setFill(Color.WHITE);
88.         } else {
89.             int colorIndex = value - 'A';
90.             cell.setFill(getColorForPiece(colorIndex));
91.         }
92.         cell.setStroke(Color.BLACK);
93.         cell.setStrokeWidth(0.5);
94.         return cell;
95.     }
96.
97.     private Color getColorForPiece(int index) {
98.         Color[] colors = {
99.             Color.RED, Color.BLUE, Color.GREEN, Color.YELLOW, Color.PURPLE,
100.                 Color.CYAN, Color.ORANGE, Color.PINK, Color.BROWN, Color.GRAY
101.             };
102.             return colors[index % colors.length];
103.         }
104.
105.         private void updatePiecesPreview() {
106.             if (piecesPreviewGrid == null || availablePieces == null) return;
107.
108.             piecesPreviewGrid.getChildren().clear();
109.             int row = 0;
110.             for (Piece piece : availablePieces) {
111.                 GridPane pieceGrid = createPiecePreview(piece);
112.                 piecesPreviewGrid.add(pieceGrid, 0, row++);
113.             }
114.         }
115.
116.         private GridPane createPiecePreview(Piece piece) {
117.             GridPane grid = new GridPane();
118.             grid.setHgap(1);
119.             grid.setVgap(1);
120.
121.             char[][] shape = piece.getShape();
122.             for (int i = 0; i < shape.length; i++) {
```

```
123.                 for (int j = 0; j < shape[0].length; j++) {
124.                     if (shape[i][j] == piece.getIdentifier()) {
125.                         Rectangle cell = new Rectangle(cellSize/2, cellSize/2);
126.                             cell.setFill(getColorForPiece(piece.getIdentifier() -
    'A'));
127.                         cell.setStroke(Color.BLACK);
128.                         cell.setStrokeWidth(0.5);
129.                         grid.add(cell, j, i);
130.                     }
131.                 }
132.             }
133.             return grid;
134.         }
135.
136.         private void resetStatistics() {
137.             timeLabel.setText("Time: 0 ms");
138.             iterationsLabel.setText("Iterations: 0");
139.             movesLabel.setText("Moves: 0");
140.             debugOutput.clear();
141.         }
142.
143.         private void enableControls(boolean enable) {
144.             solveButton.setDisable(!enable);
145.             loadButton.setDisable(!enable);
146.             resetButton.setDisable(!enable);
147.             saveImageButton.setDisable(!enable);
148.         }
149.
150.         private void showError(String title, String message) {
151.             Alert alert = new Alert(AlertType.ERROR);
152.             alert.setTitle(title);
153.             alert.setHeaderText(null);
154.             alert.setContentText(message);
155.             alert.showAndWait();
156.         }
157.
158.         private void saveAsImage() {
159.             if (board == null) {
160.                 showError("Error", "No solution to save");
161.                 return;
162.             }
163.
164.             FileChooser fileChooser = new FileChooser();
165.             fileChooser.getExtensionFilters().add(
166.                 new FileChooser.ExtensionFilter("PNG files", "*.png")
167.             );
168.
```

```
169.            File file = fileChooser.showSaveDialog(null);
170.            if (file != null) {
171.                try {
172.                    WritableImage image = boardGrid.snapshot(null, null);
173.                     ImageIO.write(SwingFXUtils.fromFXImage(image, null), "png",
    file);
174.                    statusLabel.setText("Image saved to: " + file.getName());
175.                } catch (Exception e) {
176.                    showError("Error saving image", e.getMessage());
177.                }
178.            }
179.        }
180.
181.        private void resetBoard() {
182.            if (board != null) {
183.                for (int i = 0; i < board.getRows(); i++) {
184.                    for (int j = 0; j < board.getCols(); j++) {
185.                        board.getGrid()[i][j] = '.';
186.                    }
187.                }
188.                updateBoardDisplay();
189.                resetStatistics();
190.            }
191.        }
192.
193.        private void stopSolving() {
194.            if (currentSolver != null) {
195.                currentSolver.stop();
196.                statusLabel.setText("Solving stopped");
197.                enableControls(true);
198.            }
199.        }
200.
201.        @Override
202.        public void start(Stage primaryStage) {
203.            setupLogger();
204.
205.            BorderPane root = new BorderPane();
206.            root.setPadding(new Insets(10));
207.
208.            VBox topSection = createTopSection();
209.            root.setTop(topSection);
210.
211.            VBox controlSection = createControlSection();
212.            root.setLeft(controlSection);
213.
214.            VBox boardSection = createBoardSection();
```

```
215.            root.setCenter(boardSection);
216.
217.            VBox piecesSection = createPiecesSection();
218.            root.setRight(piecesSection);
219.
220.            VBox bottomSection = createBottomSection();
221.            root.setBottom(bottomSection);
222.
223.            Scene scene = new Scene(root, 1200, 800);
224.            primaryStage.setTitle("IQ Puzzler Pro Solver");
225.            primaryStage.setScene(scene);
226.
227.            board = new Board(5, 5);
228.            updateBoardDisplay();
229.
230.            primaryStage.show();
231.        }
232.
233.        private VBox createTopSection() {
234.            VBox topSection = new VBox(10);
235.            topSection.setAlignment(Pos.CENTER);
236.
237.            Label titleLabel = new Label("IQ Puzzler Pro Solver");
238.            titleLabel.setStyle("-fx-font-size: 24px; -fx-font-weight: bold;");
239.
240.            HBox modeSelection = createModeSelection();
241.
242.            topSection.getChildren().addAll(titleLabel, modeSelection);
243.            return topSection;
244.        }
245.
246.        private HBox createModeSelection() {
247.            HBox modeBox = new HBox(20);
248.            modeBox.setAlignment(Pos.CENTER);
249.            modeBox.setPadding(new Insets(10));
250.
251.            modeGroup = new ToggleGroup();
252.
253.            RadioButton defaultMode = new RadioButton("Default Mode");
254.            RadioButton customMode = new RadioButton("Custom Mode");
255.            RadioButton pyramidMode = new RadioButton("Pyramid Mode");
256.
257.            defaultMode.setToggleGroup(modeGroup);
258.            customMode.setToggleGroup(modeGroup);
259.            pyramidMode.setToggleGroup(modeGroup);
260.
261.            defaultMode.setSelected(true);
```

```
262.
263.            modeGroup.selectedToggleProperty().addListener((obs, oldVal, newVal)
    -> {
264.                if (newVal != null) {
265.                    RadioButton selected = (RadioButton) newVal;
266.                        String mode = selected.getText().toUpperCase().replace("
    MODE", "");
267.                    currentModeProperty.set(mode);
268.                    if (board != null) {
269.                        board.setPuzzleType(mode);
270.                    }
271.                    logger.info("Mode changed to: " + mode);
272.                }
273.            });
274.
275.            modeBox.getChildren().addAll(defaultMode, customMode, pyramidMode);
276.            return modeBox;
277.        }
278.
279.        private VBox createControlSection() {
280.            VBox controlSection = new VBox(10);
281.            controlSection.setPadding(new Insets(10));
282.            controlSection.setPrefWidth(250);
283.            TitledPane fileControls = createFileControlsGroup();
284.            TitledPane solverControls = createSolverControlsGroup();
285.            TitledPane displaySettings = createDisplaySettingsGroup();
286.            TitledPane debugSettings = createDebugSettingsGroup();
287.               controlSection.getChildren().addAll(fileControls, solverControls,
    displaySettings, debugSettings);
288.            return controlSection;
289.        }
290.
291.        private TitledPane createFileControlsGroup() {
292.            VBox controls = new VBox(10);
293.            controls.setPadding(new Insets(10));
294.
295.            loadButton = new Button("Load Puzzle");
296.            saveImageButton = new Button("Save as Image");
297.            resetButton = new Button("Reset Board");
298.
299.            loadButton.setMaxWidth(Double.MAX_VALUE);
300.            saveImageButton.setMaxWidth(Double.MAX_VALUE);
301.            resetButton.setMaxWidth(Double.MAX_VALUE);
302.
303.            loadButton.setOnAction(e -> loadPuzzle());
304.            saveImageButton.setOnAction(e -> saveAsImage());
305.            resetButton.setOnAction(e -> resetBoard());
```

```
306.
307.                     controls.getChildren().addAll(loadButton,  saveImageButton,
    resetButton);
308.
309.           TitledPane fileControls = new TitledPane("File Controls", controls);
310.           fileControls.setCollapsible(false);
311.           return fileControls;
312.       }
313.
314.       private TitledPane createSolverControlsGroup() {
315.           VBox controls = new VBox(10);
316.           controls.setPadding(new Insets(10));
317.
318.           solveButton = new Button("Start Solving");
319.           stopButton = new Button("Stop Solving");
320.           stopButton.setDisable(true);
321.
322.           Label speedLabel = new Label("Solving Speed:");
323.           speedSlider = new Slider(0, 1000, 100);
324.           speedSlider.setShowTickLabels(true);
325.           speedSlider.setShowTickMarks(true);
326.
327.           solveButton.setMaxWidth(Double.MAX_VALUE);
328.           stopButton.setMaxWidth(Double.MAX_VALUE);
329.
330.           solveButton.setOnAction(e -> startSolving());
331.           stopButton.setOnAction(e -> stopSolving());
332.
333.             controls.getChildren().addAll(solveButton, stopButton, speedLabel,
    speedSlider);
334.
335.               TitledPane  solverControls  =  new  TitledPane("Solver  Controls",
    controls);
336.           solverControls.setCollapsible(false);
337.           return solverControls;
338.       }
339.
340.       private TitledPane createDisplaySettingsGroup() {
341.           VBox settings = new VBox(10);
342.           settings.setPadding(new Insets(10));
343.
344.           Label sizeLabel = new Label("Cell Size:");
345.           Slider sizeSlider = new Slider(20, 60, 40);
346.           sizeSlider.setShowTickLabels(true);
347.           sizeSlider.setShowTickMarks(true);
348.
349.           sizeSlider.valueProperty().addListener((obs, oldVal, newVal) -> {
```

```
                cellSize = newVal.doubleValue();
                updateBoardDisplay();
                updatePiecesPreview();
            });

            settings.getChildren().addAll(sizeLabel, sizeSlider);

               TitledPane displaySettings = new TitledPane("Display Settings",
    settings);
            displaySettings.setCollapsible(false);
            return displaySettings;
        }

        private TitledPane createDebugSettingsGroup() {
            VBox settings = new VBox(10);
            settings.setPadding(new Insets(10));

            debugModeCheckBox = new CheckBox("Debug Mode");
            showStepsCheckBox = new CheckBox("Show Steps");

            debugModeCheckBox.setSelected(true);
            showStepsCheckBox.setSelected(true);

            settings.getChildren().addAll(debugModeCheckBox, showStepsCheckBox);

                  TitledPane debugSettings = new TitledPane("Debug Settings",
    settings);
            debugSettings.setCollapsible(false);
            return debugSettings;
        }

        private VBox createBoardSection() {
            VBox boardSection = new VBox(10);
            boardSection.setAlignment(Pos.CENTER);

            boardGrid = new GridPane();
            boardGrid.setAlignment(Pos.CENTER);
            boardGrid.setHgap(1);
            boardGrid.setVgap(1);
              boardGrid.setStyle("-fx-background-color: white; -fx-padding: 10;
    -fx-border-color: gray; -fx-border-width: 1;");

            ScrollPane scrollPane = new ScrollPane(boardGrid);
            scrollPane.setFitToWidth(true);
            scrollPane.setFitToHeight(true);
            scrollPane.setPrefViewportWidth(500);
            scrollPane.setPrefViewportHeight(500);
```

```
394.
395.            HBox statsBox = new HBox(20);
396.            statsBox.setAlignment(Pos.CENTER);
397.            timeLabel = new Label("Time: 0 ms");
398.            iterationsLabel = new Label("Iterations: 0");
399.            movesLabel = new Label("Moves: 0");
400.                     statsBox.getChildren().addAll(timeLabel,  iterationsLabel,
    movesLabel);
401.
402.            boardSection.getChildren().addAll(scrollPane, statsBox);
403.            return boardSection;
404.        }
405.
406.        private VBox createPiecesSection() {
407.            VBox piecesSection = new VBox(10);
408.            piecesSection.setPadding(new Insets(10));
409.            piecesSection.setPrefWidth(200);
410.
411.            Label piecesLabel = new Label("Available Pieces");
412.            piecesLabel.setStyle("-fx-font-weight: bold;");
413.
414.            piecesPreviewGrid = new GridPane();
415.            piecesPreviewGrid.setAlignment(Pos.CENTER);
416.            piecesPreviewGrid.setHgap(10);
417.            piecesPreviewGrid.setVgap(10);
418.
419.            ScrollPane scrollPane = new ScrollPane(piecesPreviewGrid);
420.            scrollPane.setFitToWidth(true);
421.            scrollPane.setPrefViewportHeight(400);
422.
423.            piecesSection.getChildren().addAll(piecesLabel, scrollPane);
424.            return piecesSection;
425.        }
426.
427.        private VBox createBottomSection() {
428.            VBox bottomSection = new VBox(10);
429.            bottomSection.setPadding(new Insets(10));
430.
431.            HBox statusBar = new HBox(10);
432.            statusBar.setAlignment(Pos.CENTER_LEFT);
433.
434.            statusLabel = new Label("Ready");
435.            progressBar = new ProgressBar(0);
436.            progressBar.setPrefWidth(200);
437.
438.            statusBar.getChildren().addAll(statusLabel, progressBar);
439.
```

```
            debugOutput = new TextArea();
            debugOutput.setPrefRowCount(5);
            debugOutput.setEditable(false);
            debugOutput.setWrapText(true);

            bottomSection.getChildren().addAll(statusBar, debugOutput);
            return bottomSection;
        }

        private void loadPuzzle() {
            FileChooser fileChooser = new FileChooser();
            fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Text Files", "*.txt")
            );

            File file = fileChooser.showOpenDialog(null);
            if (file != null) {
                try {
                    board = Board.fromFile(file);
                    board.setPuzzleType(currentModeProperty.get());
                    availablePieces = new ArrayList<>(board.getPieces());
                    updateBoardDisplay();
                    updatePiecesPreview();
                    resetStatistics();
                    enableControls(true);
                    statusLabel.setText("Puzzle loaded: " + file.getName());
                                        logger.info("Loaded  puzzle  from:  "  +
    file.getAbsolutePath());
                } catch (Exception e) {
                    showError("Error loading puzzle", e.getMessage());
                    logger.severe("Error loading puzzle: " + e.getMessage());
                }
            }
        }

        private void startSolving() {
            if (board == null) {
                showError("Error", "No puzzle loaded");
                return;
            }

            solveButton.setDisable(true);
            stopButton.setDisable(false);
            loadButton.setDisable(true);
            resetButton.setDisable(true);
            statusLabel.setText("Solving puzzle...");
            progressBar.setProgress(ProgressBar.INDETERMINATE_PROGRESS);
```

```
486.
487.             currentSolver = new Solver(board, boardGrid);
488.             currentSolver.addListener(this);
489.             currentSolver.setDelay((long) speedSlider.getValue());
490.             currentSolver.setDebugMode(debugModeCheckBox.isSelected());
491.             currentSolver.setShowSteps(showStepsCheckBox.isSelected());
492.
493.             Thread solverThread = new Thread(() -> {
494.                 try {
495.                     boolean solved = currentSolver.solve();
496.                     Platform.runLater(() -> {
497.                         solveButton.setDisable(false);
498.                         stopButton.setDisable(true);
499.                         loadButton.setDisable(false);
500.                         resetButton.setDisable(false);
501.                         progressBar.setProgress(0);
502.                             statusLabel.setText(solved ? "Solution found!" : "No
    solution exists");
503.
504.                         Alert alert = new Alert(solved ? AlertType.INFORMATION :
    AlertType.WARNING);
505.                                 alert.setTitle(solved ? "Solution Found" : "No
    Solution");
506.                         alert.setHeaderText(null);
507.                         alert.setContentText(String.format(
508.                             "%s\nTime taken: %d ms\nIterations: %d",
509.                             solved ? "Solution found!" : "No solution exists",
510.                             currentSolver.getElapsedTime(),
511.                             currentSolver.getIterationCount()
512.                         ));
513.                         alert.showAndWait();
514.                     });
515.                 } catch (Exception e) {
516.                     Platform.runLater(() -> {
517.                         showError("Error during solving", e.getMessage());
518.                         enableControls(true);
519.                         progressBar.setProgress(0);
520.                         statusLabel.setText("Error occurred");
521.
522.                                     logger.severe("Error during solving: " +
    e.getMessage());
523.                         e.printStackTrace();
524.                     });
525.                 }
526.             });
527.
528.             solverThread.setDaemon(true);
```

```
529.            solverThread.start();
530.        }
531.
532.        private void setupLogger() {
533.            try {
534.                FileHandler fh = new FileHandler("solver.log", true);
535.                SimpleFormatter formatter = new SimpleFormatter();
536.                fh.setFormatter(formatter);
537.                logger.addHandler(fh);
538.                logger.setLevel(Level.ALL);
539.            } catch (Exception e) {
540.                            System.err.println("Could  not  set  up  logger:  "  +
    e.getMessage());
541.                e.printStackTrace();
542.            }
543.        }
544.
545.        private String formatDuration(long milliseconds) {
546.            if (milliseconds < 1000) {
547.                return milliseconds + " ms";
548.            }
549.            long seconds = milliseconds / 1000;
550.            milliseconds %= 1000;
551.            if (seconds < 60) {
552.                return String.format("%d.%03d s", seconds, milliseconds);
553.            }
554.            long minutes = seconds / 60;
555.            seconds %= 60;
556.                        return  String.format("%d:%02d.%03d",  minutes,  seconds,
    milliseconds);
557.        }
558.
559.        private void saveSolutionToFile() {
560.            if (board == null) {
561.                showError("Error", "No solution to save");
562.                return;
563.            }
564.
565.            FileChooser fileChooser = new FileChooser();
566.            fileChooser.getExtensionFilters().add(
567.                new FileChooser.ExtensionFilter("Text Files", "*.txt")
568.            );
569.
570.            File file = fileChooser.showSaveDialog(null);
571.            if (file != null) {
572.                try {
573.                    board.saveToFile(file.getAbsolutePath());
```

```
574.                     statusLabel.setText("Solution saved to: " + file.getName());
575.                     logger.info("Saved solution to: " + file.getAbsolutePath());
576.                 } catch (Exception e) {
577.                     showError("Error saving solution", e.getMessage());
578.                     logger.severe("Error saving solution: " + e.getMessage());
579.                 }
580.             }
581.         }
582.
583.         private void toggleFullScreen(Stage stage) {
584.             stage.setFullScreen(!stage.isFullScreen());
585.         }
586.
587.         private void showAboutDialog() {
588.             Alert alert = new Alert(AlertType.INFORMATION);
589.             alert.setTitle("About IQ Puzzler Pro Solver");
590.             alert.setHeaderText("IQ Puzzler Pro Solver");
591.             alert.setContentText(
592.                 "Version 1.0\n\n" +
593.                       "A solver for IQ Puzzler Pro puzzles using brute force
    algorithm.\n\n" +
594.                 "Created as part of IF2211 Strategi Algoritma\n" +
595.                 "Institut Teknologi Bandung\n\n" +
596.                 "© 2025"
597.             );
598.             alert.showAndWait();
599.         }
600.
601.         private void showHelpDialog() {
602.             Alert alert = new Alert(AlertType.INFORMATION);
603.             alert.setTitle("Help");
604.             alert.setHeaderText("How to Use IQ Puzzler Pro Solver");
605.             alert.setContentText(
606.                 "1. Load a puzzle file using the 'Load Puzzle' button\n" +
607.                 "2. Select solving mode (Default/Custom/Pyramid)\n" +
608.                 "3. Adjust solver settings if needed:\n" +
609.                 "   - Solving speed\n" +
610.                 "   - Debug mode\n" +
611.                 "   - Show steps\n" +
612.                 "4. Click 'Start Solving' to begin\n" +
613.                 "5. Use 'Stop Solving' to halt the process\n" +
614.                 "6. Save solution as image or text file\n\n" +
615.                   "You can also manually place pieces by dragging them from the
    pieces panel."
616.             );
617.             alert.showAndWait();
618.         }
```

```
619.
620.        @Override
621.        public void stop() {
622.            if (currentSolver != null && currentSolver.isRunning()) {
623.                currentSolver.stop();
624.            }
625.            for (Handler handler : logger.getHandlers()) {
626.                handler.close();
627.            }
628.        }
629.
630.        public static void main(String[] args) {
631.            System.setProperty("javafx.preloader", "true");
632.            launch(args);
633.        }
634.    }
```

2.3. Alur Program

Implementasi algoritma *brute force* dalam program ini berpusat pada class Solver, yang mengorganisir logika pencarian solusi secara sistematis dan menyeluruh. Inti dari implementasi ini terletak pada method tryAllPieces() yang mengeksekusi pencarian *brute force* dengan mencoba setiap kemungkinan penempatan komponen puzzle pada area permainan. Method ini bekerja secara rekursif, dimulai dengan mempersiapkan kondisi awal pencarian melalui method solve(), yang kemudian memanggil tryAllPieces() untuk memulai proses eksplorasi yang komprehensif.

Mekanisme pencarian dalam tryAllPieces() mengikuti pola backtracking yang metodis, dimana untuk setiap piece pada currentIndex, program akan mencoba seluruh kemungkinan orientasi melalui getAllOrientations() dan koordinat melalui getAllPositions(). Setiap percobaan penempatan piece divalidasi menggunakan canPlacePiece() yang memastikan komponen tetap dalam batas area dan tidak tumpang tindih dengan komponen lain. Untuk konfigurasi CUSTOM, terdapat pemeriksaan tambahan yang memastikan kesesuaian dengan pola yang diinginkan. Ketika suatu komponen berhasil ditempatkan dengan placePiece(), algoritma akan melanjutkan secara rekursif ke komponen berikutnya. Jika menemui jalan buntu, program akan melakukan backtrack dengan removePiece() dan mencoba alternatif lain.

Verifikasi hasil akhir dilakukan melalui method isComplete() yang memiliki parameter berbeda bergantung pada jenis puzzle. Untuk konfigurasi DEFAULT, method ini memeriksa kelengkapan pengisian area permainan. Sementara untuk konfigurasi CUSTOM, verifikasi memastikan terpenuhinya pola yang ditentukan. Kompleksitas dari implementasi *brute force* ini menjamin ditemukannya solusi jika memang ada, karena algoritma secara sistematis menjelajahi seluruh ruang kemungkinan penempatan komponen. Setiap langkah pencarian dicatat untuk keperluan analisis performa, memberikan wawasan tentang kompleksitas komputasi yang dibutuhkan untuk menemukan solusi.

BAB III

HASIL PROGRAM

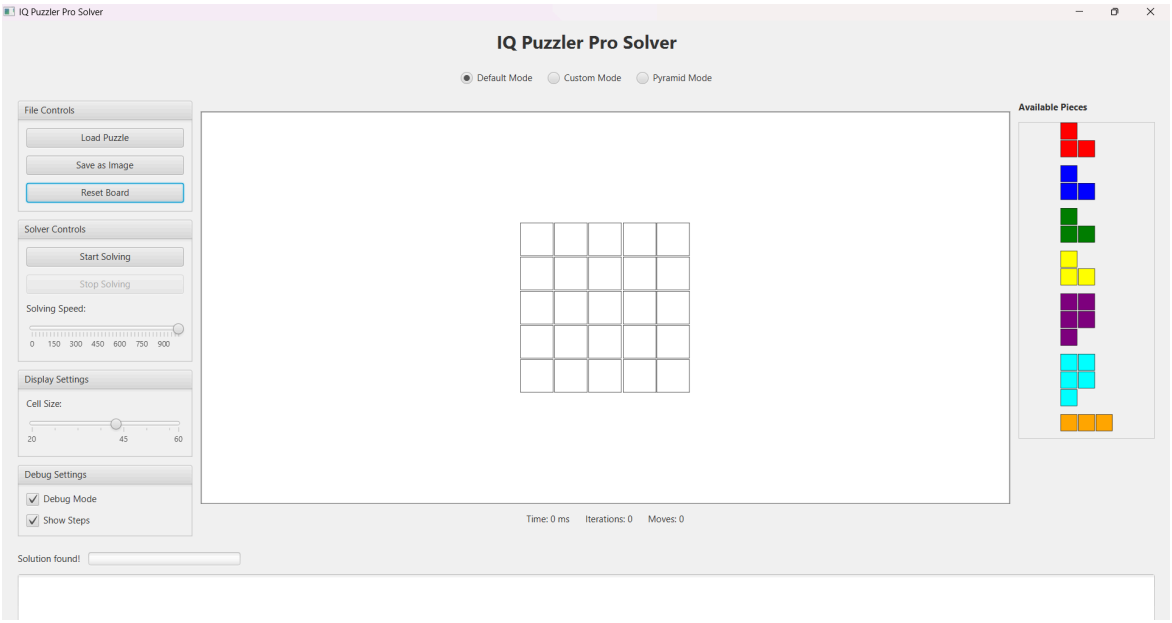
3.1. test1.txt

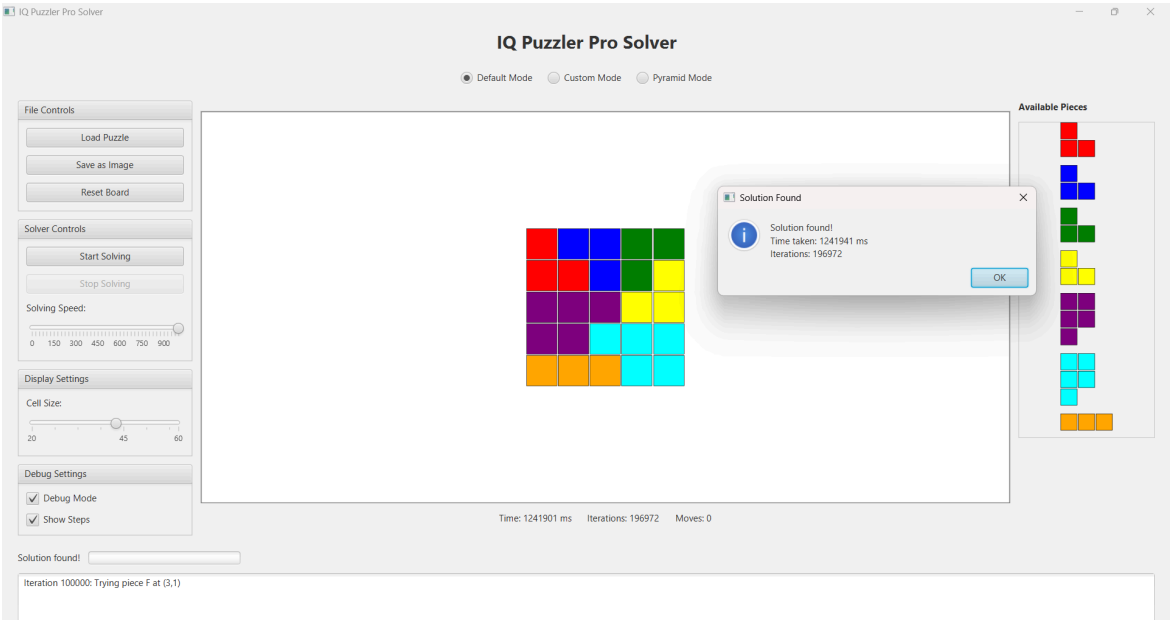
```
5 5 7
DEFAULT
A
AA
B
BB
C
CC
D
DD
EE
EE
E
FF
FF
F
GGG
```

Berikut merupakan hasil output test1.txt via terminal:

```
IQ Puzzler Pro Solver
Choose view mode (1: Terminal, 2: GUI): 1
Enter file path: C:\Users\DANENDRA\OneDrive\Documents\ITB\SEMESTER 4\IF2211 Str
ategi Algoritma\Tucil 1\test\test1.txt
Pieces read from file:
Piece A:
A.
AA
Piece B:
B.
BB
Piece C:
C.
CC
Piece D:
D.
DD
Piece E:
EE
EE
E.
Piece F:
FF
FF
F.
Piece G:
GGG

Solving puzzle...
[DEBUG] Iteration 100000: Trying piece F at (3,1)

Results:
=========

Solution found:
ABBCC
AABCD
EEEDD
EEFFF
GGGFF

Waktu pencarian: 43 ms
Banyak kasus yang ditinjau: 196972

Apakah anda ingin menyimpan solusi? (ya/tidak): ya
Enter output file path: hasil1.txt
Solution saved to: hasil1.txt
```

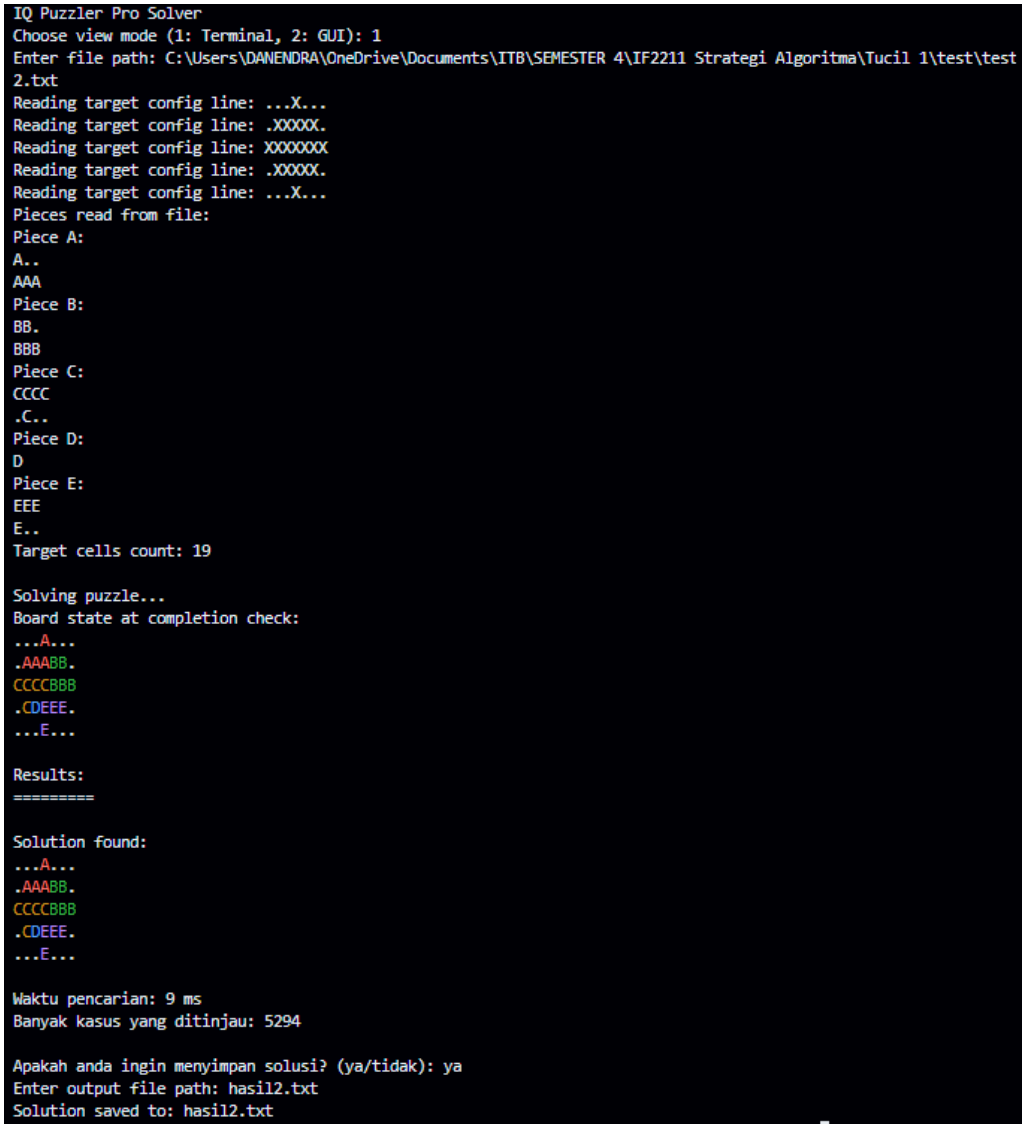
Berikut merupakan hasil output test1.txt via GUI:



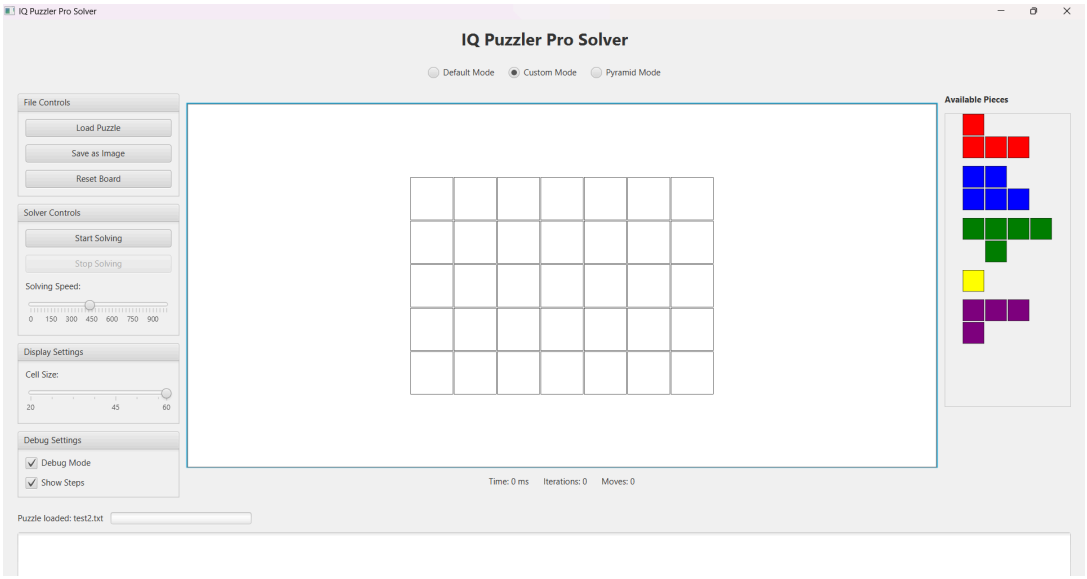


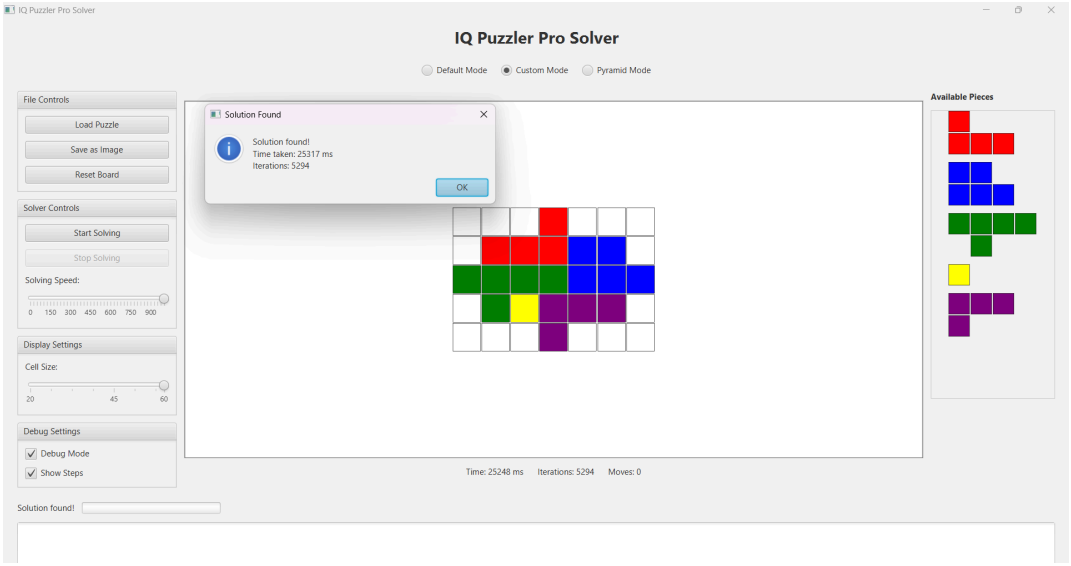
3.2. test2.txt

Berikut merupakan hasil output test1.txt via terminal:



Berikut merupakan hasil output test1.txt via GUI:





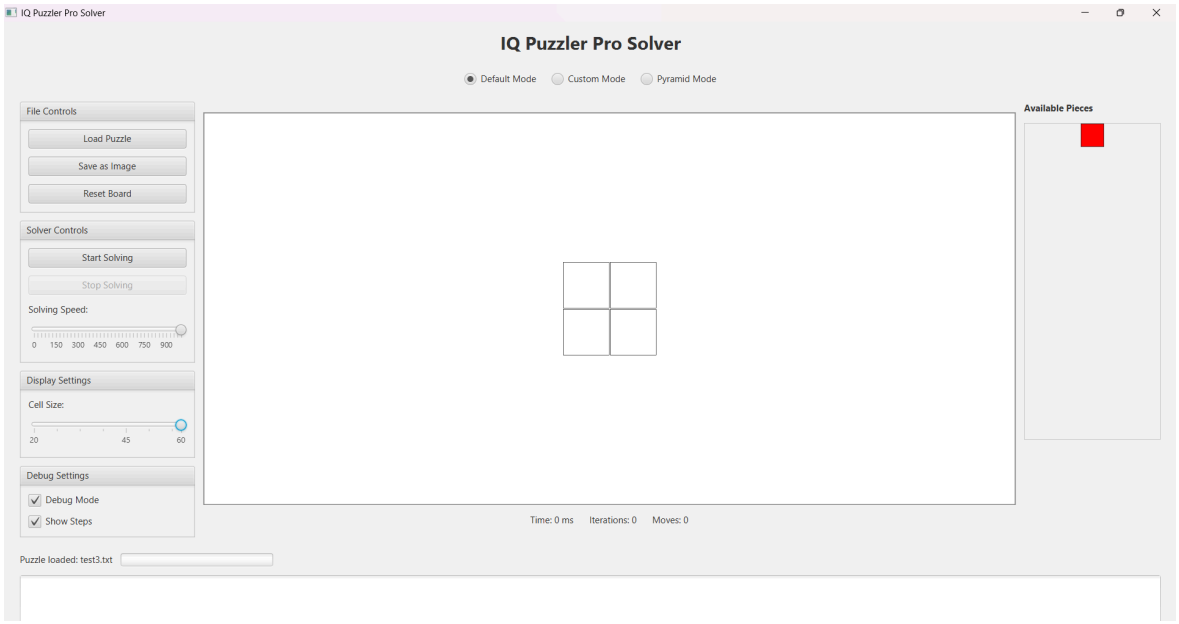
3.3. test3.txt

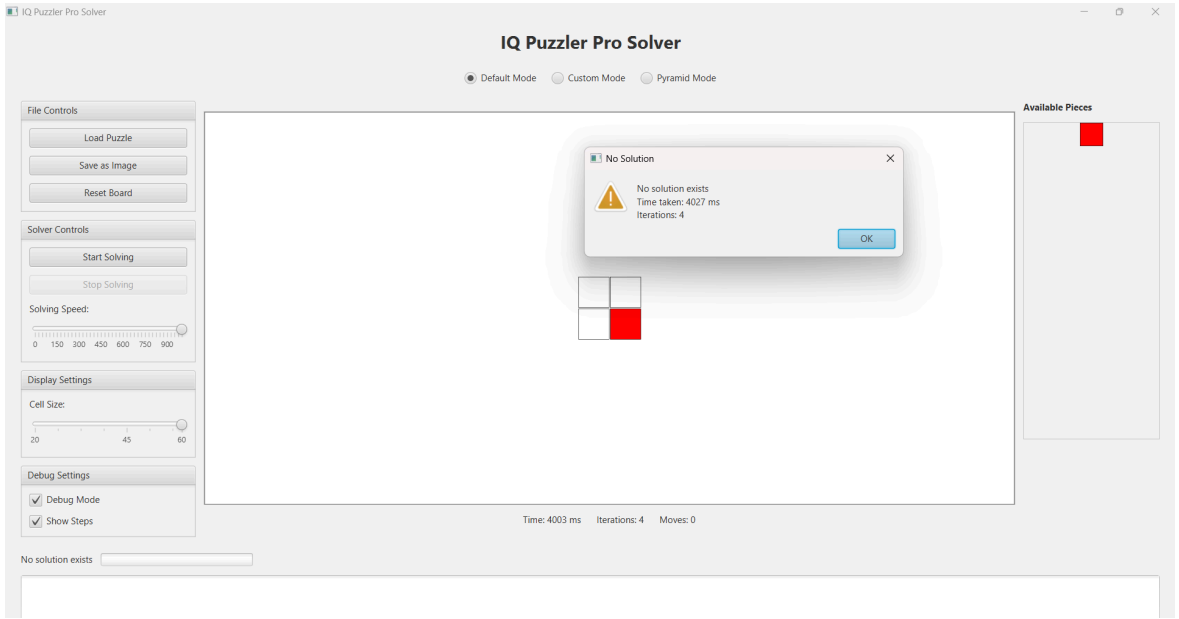
```
2 2 1
DEFAULT
A
```

Berikut merupakan hasil output test1.txt via terminal:

```
IQ Puzzler Pro Solver
Choose view mode (1: Terminal, 2: GUI): 1
Enter file path: C:\Users\DANENDRA\OneDrive\Documents\ITB\SEMESTER 4\IF2211 Strategi Algoritma\Tucil 1\test\test
3.txt
Error reading file: Invalid puzzle type: DEFAULT
```

Berikut merupakan hasil output test1.txt via GUI:





BAB IV

PENUTUP

4.1. Kesimpulan

Implementasi algoritma *brute force* untuk menyelesaikan IQ Puzzler Pro telah berhasil menghadirkan solusi komprehensif dalam menangani kompleksitas permainan puzzle. Melalui pendekatan sistematis yang menguji seluruh kemungkinan penempatan blok puzzle, program yang dikembangkan mampu mengeksplorasi ruang solusi dengan metodologi pencarian menyeluruh yang menjamin kebenaran hasil. Penelitian ini tidak hanya membuktikan efektivitas algoritma *brute force* dalam memecahkan permasalahan kombinatorial, tetapi juga menghadirkan antarmuka pengguna yang memvisualisasikan proses komputasi secara interaktif dan informatif. Implementasi yang dilakukan menunjukkan bahwa meskipun algoritma *brute force* memiliki kompleksitas komputasi tinggi, pendekatan ini tetap menjadi solusi fundamental dalam menyelesaikan persoalan yang membutuhkan penelusuran yang menyeluruh.

4.2. Source Code Repository

Link repository dapat diakses sebagai berikut: https://github.com/danenftyessir/Tucil1_13523136.git

4.1. Checklist Penyelesaian Tugas Kecil

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	✔	