

**Tugas Kecil 3**  
**IF2211 Strategi Algoritma**  
**Penyelesaian Puzzle Rush Hour Menggunakan Algoritma**  
**Pathfinding**



Disusun oleh:  
Samuel Gerrard Hamonangan Girsang / 13523064  
Lutfi Hakim Yusra / 13523084

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**  
**2025**

## I. Pendahuluan



**Gambar 1.** Rush Hour Puzzle

(Sumber: <https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak dapat berputar. Tujuan utama dari permainan ini adalah memindahkan mobil merah ke pintu keluar dengan jumlah langkah seminimal mungkin.

Komponen penting dari permainan Rush Hour terdiri dari:

1. **Papan** – *Papan* merupakan tempat permainan dimainkan.

*Papan* terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi *piece* dan *orientasi*, antara *horizontal* atau *vertikal*.

Hanya **primary piece** yang dapat digerakkan **keluar papan melewati pintu keluar**. *Piece* yang bukan *primary piece* tidak dapat digerakkan keluar papan. Papan memiliki satu *pintu keluar* yang pasti berada di *dinding papan* dan sejajar dengan orientasi *primary piece*.

2. **Piece** – *Piece* adalah sebuah kendaraan di dalam papan. Setiap *piece* memiliki *posisi*, *ukuran*, dan *orientasi*. *Orientasi* sebuah *piece* hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. *Piece* dapat memiliki beragam *ukuran*, yaitu jumlah *cell* yang ditempati oleh *piece*. Secara standar, variasi *ukuran* sebuah *piece* adalah *2-piece* (menempati 2 *cell*) atau *3-piece* (menempati 3 *cell*). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.
3. **Primary Piece** – *Primary piece* adalah kendaraan utama yang harus dikeluarkan dari *papan* (biasanya berwarna merah). Hanya boleh terdapat satu *primary piece*.
4. **Pintu Keluar** – *Pintu keluar* adalah tempat *primary piece* dapat digerakkan keluar untuk menyelesaikan permainan
5. **Gerakan** — *Gerakan* yang dimaksudkan adalah pergeseran *piece* di dalam permainan. *Piece* hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

## II. PENJELASAN ALGORITMA

### 1. Greedy Best-First Search (GBFS)

Sesuai dengan namanya, algoritma pathfinding ini memanfaatkan pendekatan Greedy dalam pencarian solusinya. Greedy yang dilakukan menentukan *best current move* murni berdasarkan heuristik dari setiap gerakan yang mungkin. Heuristik yang digunakan akan memperkirakan jarak dari state ini menuju state goal. Algoritma ini memanfaatkan HashMap hanya untuk melihat apakah pernah mengunjungi state tersebut atau belum. GBFS memanfaatkan Priority Queue untuk menentukan pemilihan langkah berikutnya berdasarkan cost. GBFS cukup cepat dibanding algoritma lainnya jika heuristiknya baik, karena tidak melihat jalur selain yang dianggap terbaik (Greedy). Namun, ini dapat mengakibatkan solusi yang diambil tidak optimal.

Langkah-Langkah:

1. Inisialisasi Priority Queue dari state awal.
2. Ambil state dengan nilai heuristik ( $h(n)$ ) terkecil dari Priority Queue.
  - a. Jika pernah dikunjungi, lewatkan.
  - b. Jika state yang diambil adalah Goal, jalur ditemukan.
  - c. Jika tidak, ekspansi state tersebut dan masukkan tetangga-tetangganya ke dalam Priority Queue.
3. Ulangi langkah 2 hingga mencapai 2b atau hingga Priority Queue kosong, yang berarti tidak ada langkah menuju Goal.

### 2. Uniform Cost Search

Uniform Cost Search (UCS) adalah algoritma pathfinding yang digunakan untuk mencari jalur terpendek di graf yang memiliki biaya. UCS adalah varian dari algoritma Dijkstra yang memfokuskan pada biaya kumulatif dari jalur yang sudah dilalui. UCS memastikan bahwa jalur yang ditemukan adalah jalur dengan biaya terendah, namun tetap mempertimbangkan setiap langkah dengan biaya yang lebih besar. UCS menjamin solusi optimal, tetapi memerlukan memori yang sangat besar jika dibanding GBFS.

Langkah-Langkah:

1. Inisialisasi Priority Queue dari state awal.
2. Ambil state dengan biaya kumulatif terendah dari Priority Queue.

- a. Jika pernah dikunjungi dan memiliki biaya yang lebih tinggi dari yang disimpan, lewatkan.
  - b. Jika state yang diambil adalah Goal, jalur ditemukan.
  - c. Jika tidak, ekspansi state tersebut. Setiap tetangga diperbaharui biayanya berdasarkan sumber ekspansi. Tetangga yang belum pernah dikunjungi atau ditemukan dengan biaya lebih rendah, masukkan ke dalam Priority Queue.
3. Ulangi langkah 2 hingga mencapai 2b atau hingga Priority Queue kosong, yang berarti tidak ada langkah menuju Goal.

### 3. A\*

A\* adalah algoritma pathfinding yang menggabungkan metode pencarian GBFS dan juga UCS dalam pengarahan pathfindingnya. A\* menghitung biaya kumulatif dari awal hingga state ini ( $g(n)$ ) dan perkiraan biaya dari node menuju tujuan ( $h(n)$ ). Fungsi evaluasi yang digunakan untuk mengarahkan pathfindingnya menggabungkan kedua nilai untuk menemukan jalur terpendek yang optimal.

Langkah-Langkah:

1. Inisialisasi Priority Queue dari state awal.
2. Ambil state dengan biaya  $f(n)$  terendah dari Priority Queue.
  - a. Jika pernah dikunjungi dan memiliki biaya yang lebih tinggi dari yang disimpan, lewatkan.
  - b. Jika state yang diambil adalah Goal, jalur ditemukan.
  - c. Jika tidak, ekspansi state tersebut. Setiap tetangga diperbaharui nilai  $f(n)$  berdasarkan kumulatif biaya yang dilewati dan perkiraan biaya menuju Goal. Tetangga yang belum pernah dikunjungi atau ditemukan dengan biaya lebih rendah, masukkan ke dalam Priority Queue.
3. Ulangi langkah 2 hingga mencapai 2b atau hingga Priority Queue kosong, yang berarti tidak ada langkah menuju Goal.

### 4. Iterative Deepening A\* (IDA\*)

Iterative Deepening A\* (IDA\*) adalah sebuah algoritma yang menggabungkan pendekatan Iterative Deepening dengan algoritma A\*. IDA\* bertujuan untuk mengatasi kelemahan A\* dalam hal penggunaan memori yang besar, dengan menggunakan strategi pencarian yang secara bertahap meningkatkan batas  $f(n)$ . IDA\* tidak menyimpan seluruh

ruang pencarian dalam memori, tetapi hanya menyimpan path saat ini yang sedang dieksplorasi.

Langkah-Langkah:

1. Hitung heuristik dari state awal. Nilai tersebut dijadikan threshold pencarian.
2. Inisialisasi Stack dengan state awal.
3. Mengambil state pada stack.
  - a. Jika pernah dikunjungi, lewatkan.
  - b. Jika  $f(n)$  melebihi threshold, lewatkan.
  - c. Jika state yang diambil adalah Goal, jalur ditemukan.
  - d. Jika tidak, ekspansi state tersebut. Setiap tetangga diperbaharui nilai  $f(n)$  berdasarkan kumulatif biaya yang dilewati dan perkiraan biaya menuju Goal. Tetangga yang belum pernah dikunjungi atau memiliki nilai dibawah threshold akan dimasukkan ke Stack.
4. Ulangi langkah 2 hingga mencapai 2b atau hingga Stack kosong. Ketika kosong, ulangi pencarian dengan nilai threshold terkecil yang lebih besar dari threshold sebelumnya.

### III. ANALISIS ALGORITMA

#### 1. Greedy Best-First Search

Algoritma ini memanfaatkan algoritma Greedy dalam pencarian. Pada suatu state, akan melihat tetangganya mana yang memiliki nilai heuristik ( $h(n)$ ) tanpa menghiraukan cost yang digunakan untuk mencapai state tersebut. Dalam pencarian, karena tidak menghiraukan tetangga lainnya, GBFS sangat mengandalkan heuristik untuk menentukan kualitas dari solusi yang diberikan. Jika heuristik mengarahkan pencarian dengan benar, GBFS akan berjalan dengan cepat dan mengembalikan langkah-langkahnya ketika menemukan solusi. Namun, tanpa kemampuan melakukan *backtrack*, GBFS dapat terjebak di solusi local optimum, karena heuristik pada setiap state hanya perkiraan saja, bukan yang pasti.

- $f(n) = h(n)$  (perkiraan jarak menuju Goal)
- Solusi mungkin tidak optimal, berdasarkan heuristik.

#### 2. Uniform Cost Search

Algoritma ini murni memeriksa cost dari satu state menuju state lainnya. Dibandingkan dengan GBFS, UCS akan melakukan pengecekan cost terhadap tetangga dari state berdasarkan jarak yang ditempuh untuk mencapai state tersebut ( $g(n)$ ). UCS tidak membuang tetangga yang bukan merupakan local optimum, tetapi dimasukkan ke Priority Queue. Pendekatan ini menjamin UCS untuk mendapatkan solusi yang optimal. Namun, algoritma ini tidak cocok di permasalahan seperti Rush Hour Puzzle Solver, karena cost dari setiap tetangga menuju tetangga lainnya sama, yaitu satu untuk setiap gerakan balok. Karena cost dari setiap state akan berasal dari jumlah gerakan dari state awal, maka cost ini dapat dianggap sebagai 'level' dalam pencarian. Hal ini menyebabkan UCS dalam permasalahan ini sama dengan pencarian BFS, yang melakukan pencarian per level.

- $f(n) = g(n)$  (Jarak yang telah ditempuh untuk mencapai state)
- Solusi pasti optimal

#### 3. A\*

Dalam algoritma A\*, pencarian dilakukan dengan memanfaatkan nilai  $h(n)$  dan  $g(n)$  dalam pencarian, memanfaatkan jarak yang ditempuh dan perkiraan jarak menuju Goal. Pencarian ini mirip dengan UCS, tetapi cost sebuah state juga dipengaruhi oleh sebuah heuristik, yang menyebabkan pencarian ini lebih 'terarah' dibandingkan UCS. Hal ini menyebabkan pencarian Goal lebih cepat dibanding UCS, apalagi dalam permasalahan Rush Hour, yaitu permasalahan yang jarak antar state tidak mengandung informasi yang signifikan dalam mencapai tujuan karena setiap gerakan sama. Namun, optimalnya algoritma A\* bergantung kepada seberapa *admissible* heuristik yang digunakan tersebut. Sebuah heuristik dikatakan *admissible* jika nilai perkiraan  $h(n)$  tidak pernah melebihi cost optimal sesungguhnya untuk mencapai Goal dari state  $n$ . Heuristik yang digunakan di Rush Hour Puzzle Solver ada 3, yaitu:

#### 1. BlockCount

- Menghitung jumlah balok yang menghalangi balok Player dari Goal.
- *Admissible*, karena untuk memindahkan balok dari arah Player, paling sedikit membutuhkan satu gerakan, sehingga tidak mungkin BlockCount melebihi cost yang nyata.

#### 2. MaxDepth

- Menghitung jumlah dependensi blokir terdalam dari semua balok yang menghalangi balok Player dari Goal (Piece yang diblokir piece yang diblokir...). Balok yang dapat digerak dianggap tidak keblokir.
- *Admissible*, karena untuk memindahkan balok dari arah player, minimal menyelesaikan dependensi terdalam dari salah satu blok yang menghalangi balok Player.

#### 3. Recursive

- Menjumlahkan semua balok yang menghalangi balok Player, dan juga dependensinya. Ketika sebuah balok dihalangi dua balok, jumlahkan dependensi dari kedua balok tersebut. Balok yang dapat digerak dianggap tidak diblokir.
- Not Always *Admissible*, karena terdapat kemungkinan bahwa satu balok mempunyai dependensi terhadap balok yang sama. Misalkan dua balok yang menghalangi Player dihalangi satu balok yang sama. Dengan perhitungan heuristik, diperlukan 4 gerakan untuk menghilangkan itu semua. Namun, terdapat kemungkinan bahwa satu gerakan cukup untuk membebaskan balok yang menghalangi kedua balok tersebut, sehingga hanya membutuhkan 3 gerakan untuk mencapai tujuan.



- $f(n) = g(n) + h(n)$
- Tidak selalu optimal, berdasarkan *admissibility* heuristic

#### 4. Iterative Deepening A\*

IDA\* bertujuan untuk mengatasi masalah memori yang tinggi pada A\*, dengan memanfaatkan prinsip pencarian kedalaman bertahap seperti yang diterapkan pada Iterative Deepening Search (IDS). Seperti A\*, IDA\* juga mengkombinasikan cost yang sudah dibayar ( $g(n)$ ) dan estimasi biaya ke tujuan ( $h(n)$ ) untuk memandu pencarian. Setiap state yang dikunjungi ini dibatasi threshold cost yang di setiap iterasi pelan-pelan ditambahkan berdasarkan States visited.

## IV. SOURCE CODE

### 1. Main.java

Kelas utama untuk menampilkan GUI dan proses program.

```
package tucil.rhsolver.app;

import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.control.ComboBox;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.ToggleButton;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.stage.Stage;
import javafx.stage.FileChooser;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.VBox;
import javafx.scene.layout.HBox;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.ToggleGroup;
import javafx.scene.control.RadioButton;
import javafx.scene.control.Tooltip;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.util.Duration;
import javafx.scene.input.ClipboardContent;
import javafx.scene.input.Dragboard;
import javafx.scene.input.TransferMode;

import java.io.File;
import java.time.Instant;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Main extends Application {
    private StackPane previewPane;
    private GridPane previewGrid;
    private Label fileLabel;
    private GridPane boardView;
    private Board boardToSolve;
```

```

private final Map<Character, Color> colorMap = new HashMap<>();
private char[][] manualBoard;
private int manualRows = 6, manualCols = 6;
private FlowPane piecesPalette;
private Map<Character, Boolean> pieceOnBoardMap = new HashMap<>();
private int[] finishPosition = null;
private static final char FINISH_MARKER = '!';
private final Map<Character, StackPane> piecePreviews = new HashMap<>();
private char draggedPiece = 'P';
private int draggedPieceSize = 2;
private boolean draggedPieceHorizontal = true;
private char dragSourcePiece = '.';
private int[] dragSourcePosition = null;
private boolean isDraggingFromBoard = false;

private static final int POOL_HORIZONTAL_2 = 0;
private static final int POOL_VERTICAL_2 = 1;
private static final int POOL_HORIZONTAL_3 = 2;
private static final int POOL_VERTICAL_3 = 3;
private static final int POOL_HORIZONTAL_4 = 4;
private static final int POOL_VERTICAL_4 = 5;
private static final int POOL_HORIZONTAL_5 = 6;
private static final int POOL_VERTICAL_5 = 7;
private static final int POOL_HORIZONTAL_6 = 8;
private static final int POOL_VERTICAL_6 = 9;
private int currentPool = POOL_HORIZONTAL_2;

public static void main(String[] args) {
    launch();
}

@Override
public void start(Stage primaryStage) {
    colorMap.put('P', Color.RED);
    colorMap.put(FINISH_MARKER, Color.GREEN);

    for (char c = 'A'; c <= 'Z'; c++) {
        pieceOnBoardMap.put(c, false);
    }

    primaryStage.setTitle("Rush Hour Solver");
    primaryStage.setResizable(true);

    ScrollPane scrollRoot = new ScrollPane();
    scrollRoot.setFitToWidth(true);
    scrollRoot.setFitToHeight(true);
    scrollRoot.setPannable(true);

    BorderPane root = new BorderPane();
    root.setPadding(new Insets(15));
    root.setStyle("-fx-background-color: #f5f5f5;");

    VBox topSection = new VBox(10);
    topSection.setPadding(new Insets(10));
    topSection.setStyle("-fx-background-color: white; -fx-border-color:
#ddd; -fx-border-radius: 5; -fx-padding: 10;");

```

```

        HBox modeSelector = new HBox(10);
        modeSelector.setStyle("-fx-background-color: #eaeaea; -fx-padding: 8;
-fx-border-radius: 5;");
        ToggleGroup inputModeGroup = new ToggleGroup();
        RadioButton fileMode = new RadioButton("Load From File");
        RadioButton manualMode = new RadioButton("Manual Board Editor");
        fileMode.setToggleGroup(inputModeGroup);
        manualMode.setToggleGroup(inputModeGroup);
        fileMode.setSelected(true);
        fileMode.setStyle("-fx-font-weight: bold;");
        manualMode.setStyle("-fx-font-weight: bold;");
        modeSelector.getChildren().addAll(fileMode, manualMode);

        Button loadButton = new Button("Load Board File");
        loadButton.setStyle("-fx-background-color: #4CAF50; -fx-text-fill:
white; -fx-font-weight: bold;");
        fileLabel = new Label("No file loaded.");
        fileLabel.setStyle("-fx-font-size: 12;");

        TextField rowField = new TextField("6");
        TextField colField = new TextField("6");
        rowField.setPrefWidth(40);
        colField.setPrefWidth(40);
        rowField.setStyle("-fx-font-size: 12;");
        colField.setStyle("-fx-font-size: 12;");
        Button applySizeButton = new Button("Apply Size");
        applySizeButton.setStyle("-fx-background-color: #2196F3; -fx-text-fill:
white; -fx-font-weight: bold;");

        HBox sizeEditor = new HBox(5, new Label("Rows:"), rowField, new
Label("Cols:"), colField, applySizeButton);
        sizeEditor.setVisible(false);
        sizeEditor.setStyle("-fx-background-color: #eaeaea; -fx-padding: 8;
-fx-border-radius: 5;");

        // Pool toggle buttons
        HBox poolControls = new HBox(10);
        poolControls.setAlignment(Pos.CENTER_LEFT);
        poolControls.setPadding(new Insets(5));
        poolControls.setVisible(false);
        poolControls.setStyle("-fx-background-color: #eaeaea; -fx-padding: 8;
-fx-border-radius: 5;");

        Label poolLabel = new Label("Piece Pool:");
        poolLabel.setStyle("-fx-font-weight: bold;");

        ToggleGroup poolToggleGroup = new ToggleGroup();

        ToggleButton pool2H = new ToggleButton("2-Size H");
        pool2H.setToggleGroup(poolToggleGroup);
        pool2H.setSelected(true);
        pool2H.setUserData(POOL_HORIZONTAL_2);

        ToggleButton pool2V = new ToggleButton("2-Size V");
        pool2V.setToggleGroup(poolToggleGroup);

```

```

pool2V.setUserData(POOL_VERTICAL_2);

ToggleButton pool3H = new ToggleButton("3-Size H");
pool3H.setToggleGroup(poolToggleGroup);
pool3H.setUserData(POOL_HORIZONTAL_3);

ToggleButton pool3V = new ToggleButton("3-Size V");
pool3V.setToggleGroup(poolToggleGroup);
pool3V.setUserData(POOL_VERTICAL_3);

ToggleButton pool4H = new ToggleButton("4-Size H");
pool4H.setToggleGroup(poolToggleGroup);
pool4H.setUserData(POOL_HORIZONTAL_4);

ToggleButton pool4V = new ToggleButton("4-Size V");
pool4V.setToggleGroup(poolToggleGroup);
pool4V.setUserData(POOL_VERTICAL_4);

ToggleButton pool5H = new ToggleButton("5-Size H");
pool5H.setToggleGroup(poolToggleGroup);
pool5H.setUserData(POOL_HORIZONTAL_5);

ToggleButton pool5V = new ToggleButton("5-Size V");
pool5V.setToggleGroup(poolToggleGroup);
pool5V.setUserData(POOL_VERTICAL_5);

ToggleButton pool6H = new ToggleButton("6-Size H");
pool6H.setToggleGroup(poolToggleGroup);
pool6H.setUserData(POOL_HORIZONTAL_6);

ToggleButton pool6V = new ToggleButton("6-Size V");
pool6V.setToggleGroup(poolToggleGroup);
pool6V.setUserData(POOL_VERTICAL_6);

poolControls.getChildren().addAll(poolLabel, pool2H, pool2V, pool3H,
pool3V, pool4H, pool4V, pool5H, pool5V, pool6H, pool6V);

pool2H.setOnAction(e -> {
    currentPool = POOL_HORIZONTAL_2;
    updatePiecesPalette();
});

pool2V.setOnAction(e -> {
    currentPool = POOL_VERTICAL_2;
    updatePiecesPalette();
});

pool3H.setOnAction(e -> {
    currentPool = POOL_HORIZONTAL_3;
    updatePiecesPalette();
});

pool3V.setOnAction(e -> {
    currentPool = POOL_VERTICAL_3;
    updatePiecesPalette();
});

```

```

pool4H.setOnAction(e -> {
    currentPool = POOL_HORIZONTAL_4;
    updatePiecesPalette();
});

pool4V.setOnAction(e -> {
    currentPool = POOL_VERTICAL_4;
    updatePiecesPalette();
});

pool5H.setOnAction(e -> {
    currentPool = POOL_HORIZONTAL_5;
    updatePiecesPalette();
});

pool5V.setOnAction(e -> {
    currentPool = POOL_VERTICAL_5;
    updatePiecesPalette();
});

pool6H.setOnAction(e -> {
    currentPool = POOL_HORIZONTAL_6;
    updatePiecesPalette();
});

pool6V.setOnAction(e -> {
    currentPool = POOL_VERTICAL_6;
    updatePiecesPalette();
});

Button finishMarkerBtn = new Button("Place Finish Marker");
finishMarkerBtn.setStyle("-fx-background-color: #FF9800; -fx-text-fill:
white; -fx-font-weight: bold;");
finishMarkerBtn.setTooltip(new Tooltip("Drag finish marker to the
board"));
finishMarkerBtn.setVisible(false);

Button resetButton = new Button("Reset");
resetButton.setStyle("-fx-background-color: #f44336; -fx-text-fill:
white; -fx-font-weight: bold;");
resetButton.setOnAction(e -> {
    manualBoard = new char[manualRows][manualCols];
    for (int i = 0; i < manualRows; i++) for (int j = 0; j <
manualCols; j++) manualBoard[i][j] = '.';
    finishPosition = null;
    for (char c : pieceOnBoardMap.keySet()) {
        pieceOnBoardMap.put(c, false);
    }
    renderManualEditor();
});
resetButton.setVisible(false);

piecesPalette = new FlowPane(10, 10);
piecesPalette.setPadding(new Insets(10));

```

```

piecesPalette.setAlignment(Pos.CENTER_LEFT);
piecesPalette.setPrefHeight(120);
piecesPalette.setVisible(false);
piecesPalette.setStyle("-fx-background-color: white; -fx-border-color:
#ddd; -fx-border-radius: 5;");

loadButton.setOnAction(e -> loadBoardFile(primaryStage));
applySizeButton.setOnAction(e -> {
    manualRows = Integer.parseInt(rowField.getText());
    manualCols = Integer.parseInt(colField.getText());
    manualBoard = new char[manualRows][manualCols];
    for (int i = 0; i < manualRows; i++) for (int j = 0; j <
manualCols; j++) manualBoard[i][j] = '.';
    finishPosition = null;
    for (char c : pieceOnBoardMap.keySet()) {
        pieceOnBoardMap.put(c, false);
    }
    renderManualEditor();
});

fileMode.setOnAction(e -> {
    loadButton.setVisible(true);
    fileLabel.setVisible(true);
    sizeEditor.setVisible(false);
    finishMarkerBtn.setVisible(false);
    resetButton.setVisible(false);
    poolControls.setVisible(false);
    piecesPalette.setVisible(false);
    renderBoard(boardToSolve);
});

manualMode.setOnAction(e -> {
    loadButton.setVisible(false);
    fileLabel.setVisible(false);
    sizeEditor.setVisible(true);
    finishMarkerBtn.setVisible(true);
    resetButton.setVisible(true);
    poolControls.setVisible(true);
    piecesPalette.setVisible(true);
    if (manualBoard == null) {
        manualBoard = new char[manualRows][manualCols];
        for (int i = 0; i < manualRows; i++) for (int j = 0; j <
manualCols; j++) manualBoard[i][j] = '.';
    }
    updatePiecesPalette();
    renderManualEditor();
});

HBox editorSection = new HBox(20);
editorSection.setAlignment(Pos.CENTER_LEFT);

VBox leftPanel = new VBox(10, finishMarkerBtn, resetButton,
piecesPalette);
leftPanel.setPadding(new Insets(10));

topSection.getChildren().addAll(modeSelector, loadButton, fileLabel,

```

```

sizeEditor, poolControls, leftPanel);

    boardView = new GridPane();
    boardView.setHgap(2);
    boardView.setVgap(2);
    boardView.setAlignment(Pos.CENTER);
    boardView.setStyle("-fx-background-color: white; -fx-border-color:
#ddd; -fx-border-radius: 5; -fx-padding: 10;");

    HBox bottomSection = new HBox(10);
    bottomSection.setPadding(new Insets(10));
    bottomSection.setAlignment(Pos.CENTER_LEFT);
    bottomSection.setStyle("-fx-background-color: white; -fx-border-color:
#ddd; -fx-border-radius: 5;");

    ComboBox<String> algoSelector = new ComboBox<>();
    algoSelector.getItems().addAll("GBFS", "UCS", "A*", "IDA*", "SA");
    algoSelector.setValue("GBFS");
    algoSelector.setStyle("-fx-font-size: 12;");

    ComboBox<String> heuristicSelector = new ComboBox<>();
    heuristicSelector.getItems().addAll("Recursive", "Max Depth",
"BlockCount");
    heuristicSelector.setValue("Recursive");
    heuristicSelector.setStyle("-fx-font-size: 12;");

    Button solveButton = new Button("Solve");
    solveButton.setStyle("-fx-background-color: #4CAF50; -fx-text-fill:
white; -fx-font-weight: bold;");
    Label algoStatus = new Label("Algorithm: GBFS");
    Label heuristicStatus = new Label("Heuristic: Recursive");
    algoStatus.setStyle("-fx-font-size: 12;");
    heuristicStatus.setStyle("-fx-font-size: 12;");

    algoSelector.setOnAction(e -> algoStatus.setText("Algorithm: " +
algoSelector.getValue()));
    heuristicSelector.setOnAction(e -> heuristicStatus.setText("Heuristic:
" + heuristicSelector.getValue()));

    solveButton.setOnAction(e -> {
        String selectedAlgo = algoSelector.getValue();
        algoStatus.setText("Solving with " + selectedAlgo + "...");
        heuristicStatus.setText("Heuristic: " +
heuristicSelector.getValue());

        String selectedHeuristic = heuristicSelector.getValue();
        if (selectedHeuristic.equals("BlockCountDistance")) {
            heuristicStatus.setText("Heuristic: BlockCountDistance");
        } else if (selectedHeuristic.equals("Recursive")) {
            heuristicStatus.setText("Heuristic: Recursive");
        } else {
            heuristicStatus.setText("Heuristic: Max Depth");
        }
    });

    Board toSolve;
    if (manualMode.isSelected()) {

```



```

        toSolve = new Board(manualRows, manualCols);
        toSolve.setMatrix(manualBoard);
        toSolve.parsePieces();
        if (finishPosition != null) {
            Coords finishCoord = new Coords(finishPosition[0],
finishPosition[1]);
            toSolve.setGoal(finishCoord);
        }
        toSolve.updateBoard();
    } else {
        if (this.boardToSolve == null) {
            algoStatus.setText("Please load a board first.");
            return;
        }
        toSolve = this.boardToSolve;
    }

    Instant startTime = Instant.now();

    Solver solver = new Solver();
    Board goalBoard = solver.GameSolver(toSolve, selectedAlgo,
selectedHeuristic);

    java.time.Duration solvingTime =
java.time.Duration.between(startTime, Instant.now());
    long millis = solvingTime.toMillis();
    String timeString;

    if (millis < 1000) {
        timeString = millis + " milliseconds";
    } else {
        double seconds = millis / 1000.0;
        timeString = String.format("%.2f seconds", seconds);
    }

    if (goalBoard == null) {
        algoStatus.setText("No solution found.");
        return;
    }

    List<Board> steps = solver.getResultInOrder(goalBoard);
    Timeline timeline = new Timeline();
    int delayMillis = 300;
    for (int i = 0; i < steps.size(); i++) {
        Board boardStep = steps.get(i);
        KeyFrame keyFrame = new KeyFrame(Duration.millis(i *
delayMillis), event -> renderBoard(boardStep));
        timeline.getKeyFrames().add(keyFrame);
    }
    timeline.setOnFinished(ev -> {
        String resultText = String.format("Solved in %d visited nodes
using %s (Time: %s)",
            solver.getVisited(), selectedAlgo, timeString);
        algoStatus.setText(resultText);
    });
    timeline.play();

```

```

});

    bottomSection.getChildren().addAll(algoSelector, heuristicSelector,
solveButton, algoStatus, heuristicStatus);

    root.setTop(topSection);
    root.setCenter(boardView);
    root.setBottom(bottomSection);

    scrollRoot.setContent(root);

    Scene scene = new Scene(scrollRoot, 900, 700);
    primaryStage.setScene(scene);
    primaryStage.show();

    createDraggableFinishMarker(finishMarkerBtn);
}

private void updatePiecesPalette() {
    piecesPalette.getChildren().clear();
    piecePreviews.clear();

    // int size = (currentPool == POOL_HORIZONTAL_2 || currentPool ==
POOL_VERTICAL_2) ? 2 : 3;
    // boolean isHorizontal = (currentPool == POOL_HORIZONTAL_2 ||
currentPool == POOL_HORIZONTAL_3);
    // currentpool can go from 0 to 9, 0 and 1 are 2-size, 2 and 3 are
3-size, 4 and 5 are 4-size, 6 and 7 are 5-size, and 8 and 9 are 6-size
    // orientation horizontal is even, vertical is odd
    int size = (currentPool / 2) + 2;
    boolean isHorizontal = (currentPool % 2 == 0);

    for (char c = 'A'; c <= 'Z'; c++) {
        createDraggablePiece(piecesPalette, c, size, isHorizontal);
    }

    updatePiecePaletteAvailability();
}

private void updatePiecePaletteAvailability() {
    for (char c = 'A'; c <= 'Z'; c++) {
        StackPane pieceContainer = piecePreviews.get(c);
        if (pieceContainer != null) {
            boolean isOnBoard = pieceOnBoardMap.getOrDefault(c, false);

            pieceContainer.setOpacity(isOnBoard ? 0.4 : 1.0);
            pieceContainer.setDisable(isOnBoard);

            if (isOnBoard) {
                for (int i = 0; i < pieceContainer.getChildren().size();
i++) {
                    if (pieceContainer.getChildren().get(i) instanceof
GridPane) {
                        GridPane grid = (GridPane)
pieceContainer.getChildren().get(i);
                        for (int j = 0; j < grid.getChildren().size(); j++)

```

```

{
    if (grid.getChildren().get(j) instanceof
Rectangle) {
        Rectangle rect = (Rectangle)
grid.getChildren().get(j);
        rect.setOpacity(1.0);
    }
}
}
}
}
}
}

private void loadBoardFile(Stage stage) {
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Open Rush Hour Board File");
    fileChooser.getExtensionFilters().add(new
FileChooser.ExtensionFilter("Text Files", "*.txt"));
    File selectedFile = fileChooser.showOpenDialog(stage);
    if (selectedFile != null) {
        fileLabel.setText("Loaded: " + selectedFile.getName());
        this.boardToSolve = IO.readInput(selectedFile.getAbsolutePath());
        assert boardToSolve != null;
        this.boardToSolve.updateBoard();
        renderBoard(this.boardToSolve);
    } else {
        fileLabel.setText("File loading cancelled.");
    }
}

private void renderBoard(Board board) {
    if (board == null) return;
    boardView.getChildren().clear();
    char[][] matrix = board.getMatrix();
    int rows = matrix.length;
    int cols = matrix[0].length;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            char c = matrix[i][j];
            Rectangle cell = new Rectangle(40, 40);
            cell.setStroke(Color.BLACK);
            if (c == '.') {
                cell.setFill(Color.LIGHTGRAY);
            } else {
                colorMap.putIfAbsent(c, Color.color(Math.random(),
Math.random(), Math.random()));
                cell.setFill(colorMap.get(c));
            }

            StackPane cellPane = new StackPane(cell);

            if (finishPosition != null && finishPosition[0] == i &&
finishPosition[1] == j) {
                Text finishText = new Text("F");

```

```

        finishText.setFont(Font.font("Arial", FontWeight.BOLD,
16));
        cellPane.getChildren().add(finishText);
    }

    boardView.add(cellPane, j, i);
}
}

private void renderManualEditor() {
    boardView.getChildren().clear();

    for (char c = 'A'; c <= 'Z'; c++) {
        pieceOnBoardMap.put(c, false);
    }

    for (int i = 0; i < manualRows; i++) {
        for (int j = 0; j < manualCols; j++) {
            char piece = manualBoard[i][j];
            if (piece != '.' && piece != FINISH_MARKER) {
                pieceOnBoardMap.put(piece, true);
            }
        }
    }

    updatePiecePaletteAvailability();

    for (int i = 0; i < manualRows; i++) {
        for (int j = 0; j < manualCols; j++) {
            Rectangle cell = new Rectangle(40, 40);
            cell.setStroke(Color.BLACK);
            char c = manualBoard[i][j];

            if (c == '.') {
                cell.setFill(Color.LIGHTGRAY);
            } else {
                colorMap.putIfAbsent(c, Color.color(Math.random(),
Math.random(), Math.random()));
                cell.setFill(colorMap.get(c));
            }

            StackPane cellPane = new StackPane(cell);

            if (finishPosition != null && finishPosition[0] == i &&
finishPosition[1] == j) {
                Text finishText = new Text("F");
                finishText.setFont(Font.font("Arial", FontWeight.BOLD,
16));
                cellPane.getChildren().add(finishText);
            }

            final int fi = i, fj = j;

            cell.setOnMouseClicked(event -> {
                if (event.getButton() ==

```

```

        javafx.scene.input.MouseButton.SECONDARY) {
            removePieceAtPosition(fi, fj);
        }
    });

    setupBoardCellDragHandlers(cellPane, fi, fj);

    if (c != '.' && c != FINISH_MARKER) {
        setupBoardPieceDragSource(cellPane, fi, fj, c);
    }

    boardView.add(cellPane, j, i);
}
}

private void createDraggablePiece(FlowPane palette, char pieceChar, int
size, boolean isHorizontal) {
    GridPane piecePreview = new GridPane();
    piecePreview.setHgap(1);
    piecePreview.setVgap(1);

    Color pieceColor = colorMap.computeIfAbsent(pieceChar,
        k -> Color.color(Math.random(), Math.random(), Math.random()));

    int rows = isHorizontal ? 1 : size;
    int cols = isHorizontal ? size : 1;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            Rectangle cell = new Rectangle(30, 30);
            cell.setFill(pieceColor);
            if (pieceChar == 'P') {
                cell.setStroke(Color.BLACK);
                cell.setStrokeWidth(2.0);
            } else {
                cell.setStroke(pieceColor.darker());
            }
            piecePreview.add(cell, j, i);
        }
    }

    StackPane container = new StackPane(piecePreview);
    container.setStyle("-fx-padding: 5; -fx-background-color: white;
-fx-border-color: #ccc;");

    container.setUserData(new PieceData(pieceChar, size, isHorizontal));

    container.setOnDragDetected(e -> {
        if (pieceOnBoardMap.getOrDefault(pieceChar, false)) {
            return;
        }

        PieceData data = (PieceData) container.getUserData();
        draggedPiece = data.piece;
    });
}

```

```

        draggedPieceSize = data.size;
        draggedPieceHorizontal = data.isHorizontal;

        Dragboard db = container.startDragAndDrop(TransferMode.COPY);
        ClipboardContent content = new ClipboardContent();
        content.putString(String.valueOf(draggedPiece));
        db.setContent(content);
        db.setDragView(container.snapshot(null, null));
        e.consume();
    });

    palette.getChildren().add(container);
    piecePreviews.put(pieceChar, container);
}

private void createDraggableFinishMarker(Button finishMarkerBtn) {
    finishMarkerBtn.setOnDragDetected(e -> {
        draggedPiece = FINISH_MARKER;
        draggedPieceSize = 1;
        draggedPieceHorizontal = true;

        Dragboard db = finishMarkerBtn.startDragAndDrop(TransferMode.COPY);
        ClipboardContent content = new ClipboardContent();
        content.putString(String.valueOf(FINISH_MARKER));
        db.setContent(content);

        StackPane finishMarker = new StackPane();
        Rectangle bg = new Rectangle(30, 30);
        bg.setFill(Color.GREEN);
        bg.setStroke(Color.BLACK);
        Text text = new Text("!");
        text.setFont(Font.font("Arial", FontWeight.BOLD, 14));
        finishMarker.getChildren().addAll(bg, text);

        db.setDragView(finishMarker.snapshot(null, null));
        e.consume();
    });
}

private void setupBoardCellDragHandlers(StackPane cellPane, int row, int col) {
    cellPane.setOnDragOver(event -> {
        if (event.getDragboard().hasString()) {
            event.acceptTransferModes(TransferMode.ANY); // Accept both
COPY and MOVE
        }
        event.consume();
    });

    cellPane.setOnDragDropped(event -> {
        Dragboard db = event.getDragboard();
        boolean success = false;

        if (db.hasString()) {
            if (isDraggingFromBoard) {

```

```

        success = movePieceOnBoard(row, col);
    } else {
        // Handle placing new pieces from palette
        success = placePieceAtPosition(row, col, draggedPiece,
draggedPieceSize, draggedPieceHorizontal);
    }
    event.setDropCompleted(success);
} else {
    event.setDropCompleted(false);
}

isDraggingFromBoard = false;
dragSourcePiece = '.';
dragSourcePosition = null;

event.consume();
});
}

private void setupBoardPieceDragSource(StackPane cellPane, int row, int
col, char pieceChar) {
    cellPane.setOnDragDetected(e -> {
        boolean isHorizontal = false;
        int pieceSize = 1;

        int horizontalSize = 1;
        for (int j = col + 1; j < manualCols; j++) {
            if (manualBoard[row][j] == pieceChar) {
                horizontalSize++;
            } else {
                break;
            }
        }
        for (int j = col - 1; j >= 0; j--) {
            if (manualBoard[row][j] == pieceChar) {
                horizontalSize++;
            } else {
                break;
            }
        }

        int verticalSize = 1;
        for (int i = row + 1; i < manualRows; i++) {
            if (manualBoard[i][col] == pieceChar) {
                verticalSize++;
            } else {
                break;
            }
        }
        for (int i = row - 1; i >= 0; i--) {
            if (manualBoard[i][col] == pieceChar) {
                verticalSize++;
            } else {
                break;
            }
        }
    }
}

```

```

        if (horizontalSize > verticalSize) {
            isHorizontal = true;
            pieceSize = horizontalSize;
        } else {
            isHorizontal = false;
            pieceSize = verticalSize;
        }

        int startRow = row;
        int startCol = col;
        if (isHorizontal) {
            while (startCol > 0 && manualBoard[row][startCol - 1] ==
pieceChar) {
                startCol--;
            }
        } else {
            while (startRow > 0 && manualBoard[startRow - 1][col] ==
pieceChar) {
                startRow--;
            }
        }

        draggedPiece = pieceChar;
        draggedPieceSize = pieceSize;
        draggedPieceHorizontal = isHorizontal;
        dragSourcePiece = pieceChar;
        dragSourcePosition = new int[]{startRow, startCol};
        isDraggingFromBoard = true;

        Dragboard db = cellPane.startDragAndDrop(TransferMode.MOVE);
        ClipboardContent content = new ClipboardContent();
        content.putString(String.valueOf(pieceChar));
        db.setContent(content);

        GridPane piecePreview = new GridPane();
        piecePreview.setHgap(1);
        piecePreview.setVgap(1);

        Color pieceColor = colorMap.get(pieceChar);
        int previewRows = isHorizontal ? 1 : pieceSize;
        int previewCols = isHorizontal ? pieceSize : 1;

        for (int i = 0; i < previewRows; i++) {
            for (int j = 0; j < previewCols; j++) {
                Rectangle cell = new Rectangle(30, 30);
                cell.setFill(pieceColor);
                cell.setStroke(pieceColor.darker());
                piecePreview.add(cell, j, i);
            }
        }

        StackPane previewContainer = new StackPane(piecePreview);
        db.setDragView(previewContainer.snapshot(null, null));

        e.consume();

```



```

    });
}

private boolean movePieceOnBoard(int targetRow, int targetCol) {
    if (dragSourcePosition == null || dragSourcePiece == '.') {
        return false;
    }

    int sourceRow = dragSourcePosition[0];
    int sourceCol = dragSourcePosition[1];
    char pieceToMove = dragSourcePiece;

    if (sourceRow == targetRow && sourceCol == targetCol) {
        return true;
    }

    for (int i = 0; i < manualRows; i++) {
        for (int j = 0; j < manualCols; j++) {
            if (manualBoard[i][j] == pieceToMove) {
                manualBoard[i][j] = '.';
            }
        }
    }

    boolean canPlace = true;
    int pieceRows = draggedPieceHorizontal ? 1 : draggedPieceSize;
    int pieceCols = draggedPieceHorizontal ? draggedPieceSize : 1;

    if (targetRow + pieceRows > manualRows || targetCol + pieceCols >
manualCols) {
        placePieceAtPosition(sourceRow, sourceCol, pieceToMove,
draggedPieceSize, draggedPieceHorizontal);
        return false;
    }

    for (int i = 0; i < pieceRows; i++) {
        for (int j = 0; j < pieceCols; j++) {
            if (targetRow + i >= manualRows || targetCol + j >= manualCols
||
                (manualBoard[targetRow + i][targetCol + j] != '.' &&
manualBoard[targetRow + i][targetCol + j] !=
pieceToMove)) {
                canPlace = false;
                break;
            }
        }
        if (!canPlace) break;
    }

    if (canPlace) {
        placePieceAtPosition(targetRow, targetCol, pieceToMove,
draggedPieceSize, draggedPieceHorizontal);
        return true;
    } else {
        placePieceAtPosition(sourceRow, sourceCol, pieceToMove,

```

```

draggedPieceSize, draggedPieceHorizontal);
    return false;
}

}

private boolean placePieceAtPosition(int row, int col, char piece, int
size, boolean horizontal) {
    if (piece == FINISH_MARKER) {
        finishPosition = new int[]{row, col};
        renderManualEditor();
        return true;
    }

    if (pieceOnBoardMap.getOrDefault(piece, false)) {
        for (int i = 0; i < manualRows; i++) {
            for (int j = 0; j < manualCols; j++) {
                if (manualBoard[i][j] == piece) {
                    manualBoard[i][j] = '.';
                }
            }
        }
        pieceOnBoardMap.put(piece, false);
    }

    int pieceRows = horizontal ? 1 : size;
    int pieceCols = horizontal ? size : 1;

    if (row + pieceRows > manualRows || col + pieceCols > manualCols) {
        System.out.println("Piece doesn't fit at this position");
        return false;
    }

    for (int i = 0; i < pieceRows; i++) {
        for (int j = 0; j < pieceCols; j++) {
            if (manualBoard[row + i][col + j] != '.' && manualBoard[row +
i][col + j] != piece) {
                System.out.println("Space already occupied by another
piece");
                return false;
            }
        }
    }

    for (int i = 0; i < pieceRows; i++) {
        for (int j = 0; j < pieceCols; j++) {
            manualBoard[row + i][col + j] = piece;
        }
    }

    pieceOnBoardMap.put(piece, true);
    renderManualEditor();
    return true;
}

private void removePieceAtPosition(int row, int col) {
    char pieceToRemove = manualBoard[row][col];

```

```

        if (pieceToRemove == FINISH_MARKER || (finishPosition != null &&
finishPosition[0] == row && finishPosition[1] == col)) {
            finishPosition = null;
            renderManualEditor();
            return;
        }

        if (pieceToRemove == '.') {
            return;
        }

        pieceOnBoardMap.put(pieceToRemove, false);

        boolean[][] visited = new boolean>manualRows>manualCols];
        removeConnectedPieces(row, col, pieceToRemove, visited);

        renderManualEditor();
    }

    private void removeConnectedPieces(int row, int col, char pieceChar,
boolean[][] visited) {
        if (row < 0 || row >= manualRows || col < 0 || col >= manualCols) {
            return;
        }

        if (visited[row][col] || manualBoard[row][col] != pieceChar) {
            return;
        }

        visited[row][col] = true;
        manualBoard[row][col] = '.';

        removeConnectedPieces(row + 1, col, pieceChar, visited);
        removeConnectedPieces(row - 1, col, pieceChar, visited);
        removeConnectedPieces(row, col + 1, pieceChar, visited);
        removeConnectedPieces(row, col - 1, pieceChar, visited);
    }

    private static class PieceData {
        public final char piece;
        public final int size;
        public final boolean isHorizontal;

        public PieceData(char piece, int size, boolean isHorizontal) {
            this.piece = piece;
            this.size = size;
            this.isHorizontal = isHorizontal;
        }
    }
}

//Compile
//javac --module-path "C:\javafx-sdk-24.0.1\lib" --add-modules
javafx.controls,javafx.fxml -d bin src\*.java
//Run

```

```
//java --module-path "C:\javafx-sdk-24.0.1\lib" --add-modules  
javafx.controls,javafx.fxml -cp bin src.Main
```

## 2. IO.java

Kelas yang digunakan untuk input konfigurasi file .txt

```
package tucil.rhsolver.app;  
import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.List;  
  
public class IO {  
    public static Board readInput(String filename) {  
        try (BufferedReader br = new BufferedReader(new FileReader(filename)))  
        {  
            String[] firstLine = br.readLine().split(" ");  
            int N = Integer.parseInt(firstLine[0]);  
            int M = Integer.parseInt(firstLine[1]);  
            int pieceCount = Integer.parseInt(br.readLine().trim());  
  
            Board board = new Board(N, M);  
  
            List<String> allLines = new ArrayList<>();  
            String line;  
            while ((line = br.readLine()) != null) {  
                allLines.add(line);  
            }  
  
            int matrixStartRow = 0;  
  
            if (allLines.size() > N) {  
                String firstDataLine = allLines.get(0).trim();  
                if (firstDataLine.length() == 1 && firstDataLine.charAt(0) ==  
'K') {  
                    matrixStartRow = 1;  
                    int kCol = allLines.get(0).indexOf('K');  
                    board.setGoal(new Coords(-1, kCol));  
                }  
            }  
  
            Coords kPosition = null;  
  
            for (int i = matrixStartRow; i < Math.min(N + matrixStartRow,  
allLines.size()); i++) {  
                int matrixRow = i - matrixStartRow;  
  
                String currentLine = allLines.get(i);  
  
                boolean hasLeadingK = false;
```

```

        int firstNonSpaceIndex = -1;

        for (int j = 0; j < currentLine.length(); j++) {
            if (currentLine.charAt(j) != ' ') {
                firstNonSpaceIndex = j;
                break;
            }
        }

        if (firstNonSpaceIndex != -1 &&
currentLine.charAt(firstNonSpaceIndex) == 'K') {
            kPosition = new Coords(matrixRow, -1);
            hasLeadingK = true;
        }

        int matrixCol = 0;
        for (int j = 0; j < currentLine.length() && matrixCol < M;
j++) {

            char c = currentLine.charAt(j);

            if (c == ' ') continue;

            if (c == 'K' && matrixCol >= M) {
                kPosition = new Coords(matrixRow, M);
                continue;
            }

            if (c == 'K' && hasLeadingK && j == firstNonSpaceIndex) {
                continue;
            }

            if (c == '.') {
                matrixCol++;
                continue;
            }

            if (c == 'K' && matrixCol < M) {
                kPosition = new Coords(matrixRow, matrixCol);
                matrixCol++;
                continue;
            }

            Coords coord = new Coords(matrixRow, matrixCol);
            if (!board.getPieces().containsKey(c)) {
                Piece piece = new Piece(c);
                piece.addCoord(coord);
                board.addPiece(piece);
            } else {
                board.getPieces().get(c).addCoord(coord);
            }

            matrixCol++;
        }

        if (currentLine.length() > matrixCol) {
            for (int j = matrixCol; j < currentLine.length(); j++) {

```

```

        if (currentLine.charAt(j) == 'K') {
            kPosition = new Coords(matrixRow, M);
            break;
        }
    }
}

if (kPosition == null && matrixStartRow + N < allLines.size()) {
    for (int i = matrixStartRow + N; i < allLines.size(); i++) {
        String belowLine = allLines.get(i);
        if (belowLine.contains("K")) {
            int kCol = belowLine.indexOf('K');
            kPosition = new Coords(N, kCol);
        }
    }
}

if (kPosition != null) {
    board.setGoal(kPosition);
}

// normalize goal position
if (board.getGoal().getX() == -1) {
    board.getGoal().setX(0);
}
if (board.getGoal().getY() == -1) {
    board.getGoal().setY(0);
}
if (board.getGoal().getY() > M-1) {
    board.getGoal().setY(M-1);
}
if (board.getGoal().getX() > N-1) {
    board.getGoal().setX(N);
}
return board;
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
}
return null;
}

public static void saveResult(List<Board> result){
}
}

```

### 3. Coords.java

Kelas pembantu sebagai koordinat dari piece dalam board.

```
package tucil.rhsolver.app;
```

```

public class Coords {
    private int x;
    private int y;

    public Coords(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Coords() {
        this.x = 0;
        this.y = 0;
    }

    public Coords(Coords other) {
        this.x = other.x;
        this.y = other.y;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void addX(int x) {
        this.x += x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void addY(int y) {
        this.y += y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public double distanceTo(Coords other) {
        return Math.sqrt(Math.pow(this.x - other.x, 2) + Math.pow(this.y
- other.y, 2));
    }

    public boolean isIntersecting(Coords other) {
        return this.x == other.x && this.y == other.y;
    }

    @Override
    public String toString() {
        return "Coords{" +

```

```

        "x=" + x +
        ", y=" + y +
        '>';
    }
}

```

#### 4. Board.java

Kelas ini adalah objek board puzzle yang akan diselesaikan

```

package tucil.rhsolver.app;

import java.util.*;

public class Board {
    private int row;
    private int col;
    private int iteration;
    private HashMap<Character, Piece> pieces;
    private Coords goal;
    private char[][] matrix;
    private String parentState;
    private String latestMove;
    private int heuristicCost;

    public Board(int row, int col){
        this.row = row;
        this.col = col;
        this.iteration = 0;
        this.pieces = new HashMap<>();
        this.matrix = new char[row][col];
        for (int i = 0; i < row; i++){
            for (int j = 0; j < col; j++){
                matrix[i][j] = '.';
            }
        }
        this.goal = new Coords(-1, -1);
        this.parentState = "";
        this.latestMove = "";
    }

    public Board(Board board){
        this.row = board.row;
        this.col = board.col;
        this.iteration = board.iteration;
        this.pieces = new HashMap<>();
        for (var entry : board.pieces.entrySet()) {
            this.pieces.put(entry.getKey(), new Piece(entry.getValue()));
        }
        // pastikan Piece punya copy-constructor
        this.goal = new Coords(board.goal.getX(), board.goal.getY());
        this.matrix = new char[row][col];
    }
}

```



```

        for (int i = 0; i < row; i++){
            System.arraycopy(board.matrix[i], 0, matrix[i], 0, col);
        }
    }

    public void setGoal(Coords goal){
        this.goal = goal;
    }

    public Coords getGoal(){
        return goal;
    }

    public HashMap<Character, Piece> getPieces(){
        return pieces;
    }

    public void addPiece(Piece piece){
        pieces.put(piece.getId(), piece);
    }

    public Piece getPlayer(){
        return pieces.get('P');
    }

    public void setParentState(String parentState){
        this.parentState = parentState;
    }

    public String getParentState(){
        return parentState;
    }

    public void setLatestMove(String latestMove){
        this.latestMove = latestMove;
    }

    public String getLatestMove(){
        return latestMove;
    }

    public void setHeuristicCost(int heuristicCost) {
        this.heuristicCost = heuristicCost;
    }

    public int getHeuristicCost() {
        return heuristicCost;
    }

    public int getIteration(){
        return iteration;
    }

    public String getStateKey(){
        StringBuilder state = new StringBuilder();
        for (Piece piece : pieces.values()){
            state.append(piece.getId()).append(":");
            for (Coords coord : piece.getPosition()){

```

```

state.append(coord.getX()).append(":").append(coord.getY()).append(";");
    }
    }
    return state.toString();
}

public void printBoard(){
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            System.out.print(matrix[i][j]);
        }
        System.out.println();
    }
}

public void updateBoard(){
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            matrix[i][j] = '.';
        }
    }
    for (Piece piece : pieces.values()){
        for (Coords coord : piece.getPosition()){
            matrix[coord.getX()][coord.getY()] = piece.getId();
        }
    }
}

public void removePiece(char id){
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            if (matrix[i][j] == id){
                matrix[i][j] = '.';
            }
        }
    }
}

public boolean isValidMove(Character id, boolean forward){
    Piece piece = pieces.get(id);
    if (piece == null) return false;
    int mult = forward ? 1 : -1;

    if (piece.isHorizontal()){
        for (Coords coord : piece.getPosition()){
            int newY = coord.getY() + mult;
            if (newY < 0 || newY >= col ||
(matrix[coord.getX()][newY] != '.' && matrix[coord.getX()][newY] !=
id)){
                return false;
            }
        }
    } else {
        for (Coords coord : piece.getPosition()){
            int newX = coord.getX() + mult;

```

```

        if (newX < 0 || newX >= row ||
(matrix[newX][coord.getY()] != '.' && matrix[newX][coord.getY()] !=
id)){
            return false;
        }
    }
}
return true;
}

public List<Board> generatePossibleBoards() {
    List<Board> possibleBoards = new ArrayList<>();
    HashMap<Character, Piece> tempPieces = new
HashMap<>(this.pieces);
    String stateKey = getStateKey();
    for (Piece piece : tempPieces.values()) {
        if (isValidMove(piece.getId(), true)) {
            Board newBoard = new Board(this);
            newBoard.iteration++;
            newBoard.updateBoard();
            Piece newPiece = newBoard.pieces.get(piece.getId());
            newPiece.move(true);
            newBoard.updateBoard();
            possibleBoards.add(newBoard);
            newBoard.setParentState(stateKey);
            if (newPiece.isHorizontal()) {
                newBoard.setLatestMove("Move " + newPiece.getId() + "
right");
            } else {
                newBoard.setLatestMove("Move " + newPiece.getId() + "
down");
            }
            Board newNewBoard = new Board(newBoard);
            Piece newNewPiece =
newNewBoard.pieces.get(piece.getId());
            while (newNewBoard.isValidMove(newNewPiece.getId(),
true)) {
                newNewBoard.updateBoard();
                newNewPiece.move(true);
                newNewBoard.updateBoard();
                possibleBoards.add(newNewBoard);
                newNewBoard.setParentState(stateKey);
                if (newNewPiece.isHorizontal()) {
                    newNewBoard.setLatestMove("Move " +
newNewPiece.getId() + " right");
                } else {
                    newNewBoard.setLatestMove("Move " +
newNewPiece.getId() + " down");
                }
                newNewBoard = new Board(newNewBoard);
                newNewPiece = newNewBoard.pieces.get(piece.getId());
            }
        }
        if (isValidMove(piece.getId(), false)) {
            Board newBoard = new Board(this);
            newBoard.iteration++;

```

```

        newBoard.updateBoard();
        Piece newPiece = newBoard.pieces.get(piece.getId());
        newPiece.move(false);
        newBoard.updateBoard();
        possibleBoards.add(newBoard);
        newBoard.setParentState(stateKey);
        if (newPiece.isHorizontal()) {
            newBoard.setLatestMove("Move " + newPiece.getId() + "
left");
        } else {
            newBoard.setLatestMove("Move " + newPiece.getId() + "
up");
        }
        Board newNewBoard = new Board(newBoard);
        Piece newNewPiece =
newNewBoard.pieces.get(piece.getId());
        while (newNewBoard.isValidMove(newNewPiece.getId(),
false)) {
            newNewBoard.updateBoard();
            newNewPiece.move(false);
            newNewBoard.updateBoard();
            possibleBoards.add(newNewBoard);
            newNewBoard.setParentState(stateKey);
            if (newNewPiece.isHorizontal()) {
                newNewBoard.setLatestMove("Move " +
newNewPiece.getId() + " left");
            } else {
                newNewBoard.setLatestMove("Move " +
newNewPiece.getId() + " up");
            }
            newNewBoard = new Board(newNewBoard);
            newNewPiece = newNewBoard.pieces.get(piece.getId());
        }
    }
    return possibleBoards;
}

public boolean isGoalState(){
    Piece player = getPlayer();
    return player.isIntersecting(goal);
}

public char[][] getMatrix(){
    return matrix;
}

public List<Coords> stepsToGoal(){
    List<Coords> steps = new ArrayList<>();
    Coords playerFirst = new
Coords(getPlayer().getPosition().getFirst());
    Coords goal = getGoal();
    // find all the steps to goal
    while (!playerFirst.isIntersecting(goal)){
        if (getPlayer().isHorizontal()){
            if (playerFirst.getY() < goal.getY()){

```

```

        steps.add(new Coords(playerFirst.getX(),
playerFirst.getY() + 1));
        playerFirst.addY(1);
    } else {
        steps.add(new Coords(playerFirst.getX(),
playerFirst.getY() - 1));
        playerFirst.addY(-1);
    }
    } else {
        if (playerFirst.getX() < goal.getX()){
            steps.add(new Coords(playerFirst.getX() + 1,
playerFirst.getY()));
            playerFirst.addX(1);
        } else {
            steps.add(new Coords(playerFirst.getX() - 1,
playerFirst.getY()));
            playerFirst.addX(-1);
        }
    }
}
return steps;
}

public List<Piece> getAllBlocking(){
    List<Coords> blocking = stepsToGoal();
    List<Piece> blockingPieces = new ArrayList<>();
    updateBoard();
    for (Coords coord : blocking){
        // ignore if the coord is out of bounds
        if (coord.getX() < 0 || coord.getX() >= row || coord.getY() <
0 || coord.getY() >= col){
            continue;
        }
        // ignore if the coord is the same as the player
        if (matrix[coord.getX()][coord.getY()] == 'P'){
            continue;
        }
        if(matrix[coord.getX()][coord.getY()] != '.'){
            char id = matrix[coord.getX()][coord.getY()];
            Piece blockingPiece = new Piece(pieces.get(id));
            if (!blockingPieces.contains(blockingPiece)){
                blockingPieces.add(blockingPiece);
            }
        }
    }
    return blockingPieces;
}

public boolean canMove(Piece piece){
    Character self = piece.getId();
    // check if the piece can move forward or backward
    boolean canMoveForward = isValidMove(self, true);
    boolean canMoveBackward = isValidMove(self, false);
    return canMoveForward || canMoveBackward;
}

```

```

public List<Character> getPiecesBlockingPiece(Piece piece){
    // get all the pieces that are blocking the given piece
    List<Character> blockingPieces = new ArrayList<>();
    if (canMove(piece)){
        return blockingPieces;
    }
    if (piece.isHorizontal()){
        for (Coords coord : piece.getPosition()){
            int newY = coord.getY() + 1;
            if (newY < col && matrix[coord.getX()][newY] != '.'){
                char id = matrix[coord.getX()][newY];
                if (id != piece.getId()){
                    blockingPieces.add(id);
                }
            }
            newY = coord.getY() - 1;
            if (newY >= 0 && matrix[coord.getX()][newY] != '.'){
                char id = matrix[coord.getX()][newY];
                if (id != piece.getId()){
                    blockingPieces.add(id);
                }
            }
        }
    } else {
        for (Coords coord : piece.getPosition()){
            int newX = coord.getX() + 1;
            if (newX < row && matrix[newX][coord.getY()] != '.'){
                char id = matrix[newX][coord.getY()];
                if (id != piece.getId()){
                    blockingPieces.add(id);
                }
            }
            newX = coord.getX() - 1;
            if (newX >= 0 && matrix[newX][coord.getY()] != '.'){
                char id = matrix[newX][coord.getY()];
                if (id != piece.getId()){
                    blockingPieces.add(id);
                }
            }
        }
    }
    return blockingPieces;
}

public List<Character> getPiecesBlockingPiece(Character id){
    Piece piece = pieces.get(id);
    if (piece == null) return new ArrayList<>();
    return getPiecesBlockingPiece(piece);
}

public int heuristicByBlockCount(){
    List<Piece> blockingPieces = getAllBlocking();
    int count = 0;
    for (Piece piece : blockingPieces){
        if (piece.getId() != 'P'){
            count++;
        }
    }
}

```

```

    }
}
int goal = 1;
if (getPlayer().isIntersecting(this.goal)){
    goal = 0;
}
return count + goal;
}

public int heuristicByRecursiveBlock(){
    List<Piece> initialBlocking = getAllBlocking();
    Set<Piece> visitedPieces = new HashSet<>();
    Stack<Piece> blockingPieces = new Stack<>();
    for (Piece piece : initialBlocking){
        if (piece.getId() != 'P'){
            blockingPieces.push(piece);
        }
    }
    int count = 0;
    while (!blockingPieces.isEmpty()){
        Piece piece = blockingPieces.pop();
        if (visitedPieces.contains(piece)){
            continue;
        }
        count++;
        visitedPieces.add(piece);
        List<Character> blocking = getPiecesBlockingPiece(piece);
        for (Character id : blocking){
            Piece blockingPiece = pieces.get(id);
            if (blockingPiece != null &&
!visitedPieces.contains(blockingPiece)){
                blockingPieces.push(blockingPiece);
            }
        }
    }
    int goal = 1;
    if (getPlayer().isIntersecting(this.goal)){
        goal = 0;
    }
    return count + goal;
}

public int maxDepth(Piece piece, Set<Piece> visited){
    if (visited.contains(piece)){
        return 0;
    }
    visited.add(piece);
    int depth = 0;
    for (Character id : getPiecesBlockingPiece(piece)) {
        Piece child = pieces.get(id);
        if (child != null) {
            depth = Math.max(depth, maxDepth(child, visited));
        }
    }
    return depth + 1;
}

```

```

public int heuristicByMaxDepth(){
    List<Piece> initialBlocking = getAllBlocking();
    Set<Piece> visitedPieces = new HashSet<>();
    int maxDepth = 0;
    for (Piece piece : initialBlocking){
        if (piece.getId() != 'P'){
            maxDepth = Math.max(maxDepth, maxDepth(piece,
visitedPieces));
        }
    }
    int goal = 1;
    if (getPlayer().isIntersecting(this.goal)){
        goal = 0;
    }
    return maxDepth + goal;
}

public int getHeuristicByType(String type){
    if (type.equals("BlockCount")){
        return heuristicByBlockCount();
    } else if (type.equals("Recursive")){
        return heuristicByRecursiveBlock();
    } else if (type.equals("Max Depth")){
        return heuristicByMaxDepth();
    }
    return 0;
}

public void setMatrix(char[][] matrix){
    this.matrix = matrix;
}

public void parsePieces(){
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            if(this.matrix[i][j] == '.'){
                continue;
            }

            char c = this.matrix[i][j];
            Coords coord = new Coords(i, j);
            if (!pieces.containsKey(c)) {
                Piece piece = new Piece(c);
                piece.addCoord(coord);
                addPiece(piece);
            } else {
                getPieces().get(c).addCoord(coord);
            }
        }
    }
}

public boolean isBoardValid(){
    Coords goal = getGoal();
    Piece player = getPlayer();

```



```

        if (goal.getX() < 0 || goal.getX() >= row || goal.getY() < 0 ||
goal.getY() >= col){
            return false;
        }
        if (player.getPosition().isEmpty()){
            return false;
        }
        // check if the player can reach the goal by orientation
        if (player.isHorizontal()){
            return goal.getX() == player.getPosition().getFirst().getX();
        } else {
            return goal.getY() == player.getPosition().getFirst().getY();
        }
    }
}

```

## 5. Piece.java

Kelas ini tempat metode-metode dari suatu piece diimplementasikan.

```

package tucil.rhsolver.app;

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

public class Piece {
    private Character id;
    private List<Coords> position;

    public Piece(char id) {
        this.id = id;
        this.position = new ArrayList<>();
    }

    public Piece(Piece piece) {
        this.id = piece.id;
        this.position = new ArrayList<>();
        for (Coords coord : piece.position) {
            this.position.add(new Coords(coord.getX(), coord.getY()));
        }
    }

    public Character getId() {
        return id;
    }

    public List<Coords> getPosition() {
        return position;
    }

    public void addCoord(Coords coord) {
        position.add(coord);
    }
}

```

```

public Boolean isHorizontal() {
    if (position.size() < 2) return true;
    return position.get(0).getX() == position.get(1).getX();
}

public void move(boolean forward) {
    int mult = forward ? 1 : -1;
    if (isHorizontal()) {
        for (Coords coord : position) {
            coord.addY(mult);
        }
    } else {
        for (Coords coord : position) {
            coord.addX(mult);
        }
    }
}

public boolean isIntersecting(Coords other) {
    for (Coords coord : position) {
        if (coord.isIntersecting(other)) {
            return true;
        }
    }
    return false;
}

public void debugPrint() {
    System.out.print("Piece " + id + ": ");
    for (Coords coord : position) {
        System.out.print(coord.toString() + " ");
    }
    System.out.println();
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(id).append(":");
    for (Coords coord : position) {
        sb.append(coord.getX()).append(":").append(coord.getY()).append(";");
    }
    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 id == piece.id;
}
}

```

## 6. Solver.java

Kelas ini mengimplementasikan penyelesaian game dengan algoritma-algoritma yang ada.

```
package tucil.rhsolver.app;

import java.util.*;

public class Solver {
    private HashMap<String, Board> visitedStates;
    private PriorityQueue<Board> queue;
    private int visited;

    public Solver(){
        this.visitedStates = new HashMap<>();
        this.queue = new
PriorityQueue<>(Comparator.comparingInt(Board::getHeuristicCost));
        this.visited = 0;
    }

    public void addVisited(Board visitedBoard){
        this.visitedStates.put(visitedBoard.getStateKey(), visitedBoard);
    }

    public int getVisited(){
        return this.visited;
    }

    public void addQueue(Board queuedBoard){
        this.queue.add(queuedBoard);
    }

    public Board GameSolver(Board parentBoard, String algorithm, String
heuristicType){
        if (!parentBoard.isBoardValid()){
            return null;
        }
        int heuristic;
        if(algorithm.equals("GBFS")){
            heuristic = parentBoard.getHeuristicByType(heuristicType);
        } else if(algorithm.equals("UCS")){
            heuristic = parentBoard.getIteration();
        } else if (algorithm.equals("A*")){
            heuristic = parentBoard.getHeuristicByType(heuristicType) +
parentBoard.getIteration();
        } else {
            return IDAStar(parentBoard, heuristicType);
        }
        parentBoard.setHeuristicCost(heuristic);
        addQueue(parentBoard);

        while (!queue.isEmpty()){
            Board currentBoard = this.queue.poll();
        }
    }
}
```

```

currentBoard.printBoard();
if(currentBoard.isGoalState()){
    addVisited(currentBoard);
    System.out.println("Visited: " + visited);
    System.out.println("Heuristic: " + heuristicType);
    return currentBoard;
}
String currentKey = currentBoard.getStateKey();
if(this.visitedStates.containsKey(currentKey)){
    if (this.visitedStates.get(currentKey).getHeuristicCost()
<= currentBoard.getHeuristicCost() || algorithm.equals("GBFS")) {
        continue;
    }
}

addVisited(currentBoard);
visited++;

for(Board next : currentBoard.generatePossibleBoards()){
    String key = next.getStateKey();
    if(!this.visitedStates.containsKey(key) ||
this.visitedStates.get(key).getHeuristicCost() >
next.getHeuristicCost()){
        int childHeuristic;
        if(algorithm.equals("GBFS")){
            childHeuristic =
next.getHeuristicByType(heuristicType);
        } else if(algorithm.equals("UCS")){
            childHeuristic = next.getIteration();
            System.out.println(childHeuristic);
        } else {
            childHeuristic =
next.getHeuristicByType(heuristicType) + next.getIteration();
        }
        next.setHeuristicCost(childHeuristic);
        addQueue(next);
    }
}

return null;
}

public List<Board> getResultInOrder(Board goalBoard) {
    List<Board> path = new ArrayList<>();

    Board currentBoard = goalBoard;

    Stack<Board> reversePath = new Stack<>();

    while (currentBoard != null) {
        reversePath.push(currentBoard);

        if (currentBoard.getParentState().equals("")) {
            break;

```

```

    }

    String parentKey = currentBoard.getParentState();
    currentBoard = visitedStates.get(parentKey);

}

while (!reversePath.isEmpty()) {
    path.add(reversePath.pop());
}

return path;
}

public Board IDAStar(Board parentBoard, String heuristicType) {
    int threshold = parentBoard.getHeuristicByType(heuristicType);

    while (true) {
        visitedStates.clear();

        Stack<Board> stack = new Stack<>();
        stack.push(parentBoard);

        int nextThreshold = Integer.MAX_VALUE;

        while (!stack.isEmpty()) {
            Board currentBoard = stack.pop();

            if (currentBoard.isGoalState()) {
                System.out.println("Visited: " + visited);
                System.out.println("Heuristic: " + heuristicType);
                return currentBoard;
            }

            String currentKey = currentBoard.getStateKey();
            if (visitedStates.containsKey(currentKey)) {
                Board storedIteration =
visitedStates.get(currentKey);
                if (storedIteration.getIteration() <=
currentBoard.getIteration()) {
                    continue;
                }
            }

            visited++;
            addVisited(currentBoard);

            List<Board> successors = new
ArrayList<>(currentBoard.generatePossibleBoards());

            Collections.reverse(successors);

            for (Board next : successors) {
                int g = next.getIteration();
                int h = next.getHeuristicByType(heuristicType);
                int f = g + h;

```

```
        next.setHeuristicCost(f);

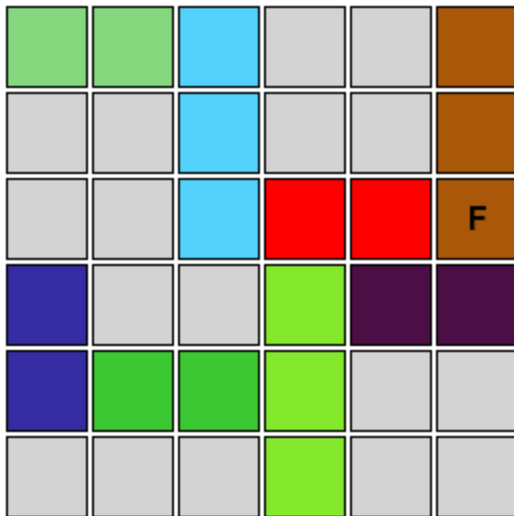
        if (f <= threshold) {
            stack.push(next);
        } else {
            nextThreshold = Math.min(nextThreshold, f);
        }
    }
}

if (nextThreshold == Integer.MAX_VALUE) {
    return null;
}

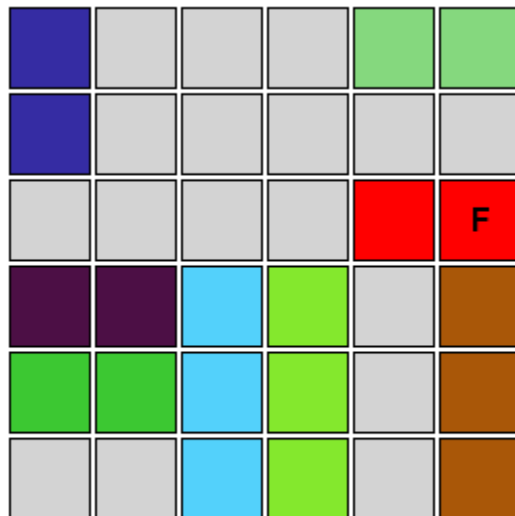
threshold = nextThreshold;
}
}
```

## V. TEST CASE

### 1. Konfigurasi:

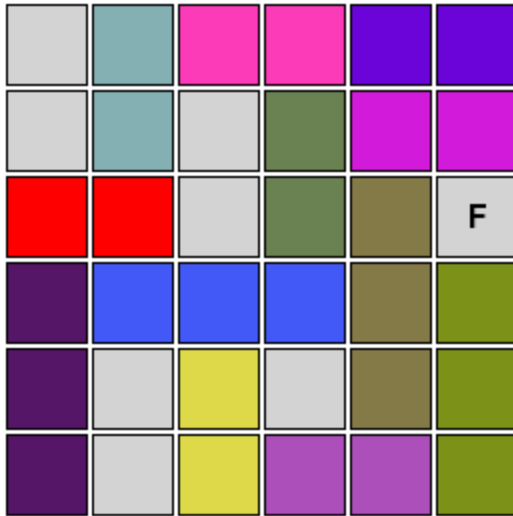


Hasil (GBFS):

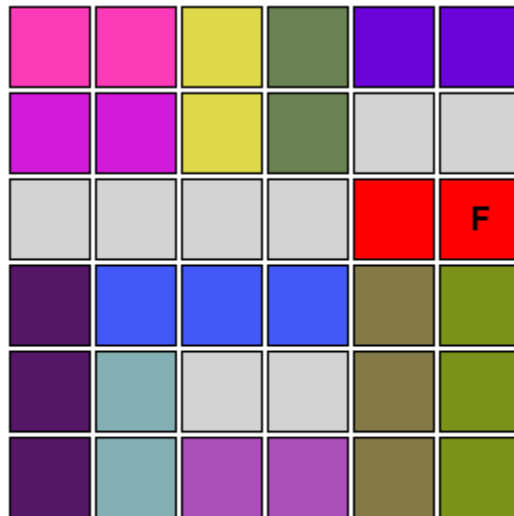


Solved in 600 visited nodes using GBFS (Time: 2.08 seconds) Heuristic: Recursive

2. Konfigurasi:



Hasil (A\*):

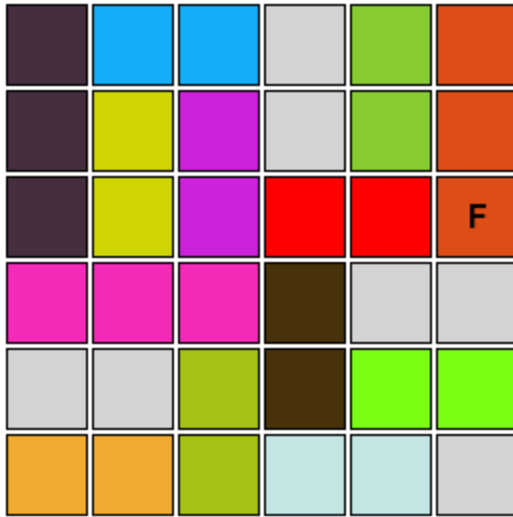



---

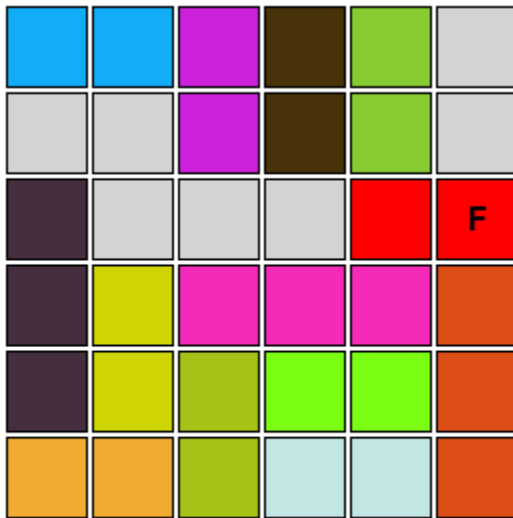
Solved in 3355 visited nodes using A\* (Time: 109 milliseconds) Heuristic: Recursive

Konfigurasi 3:





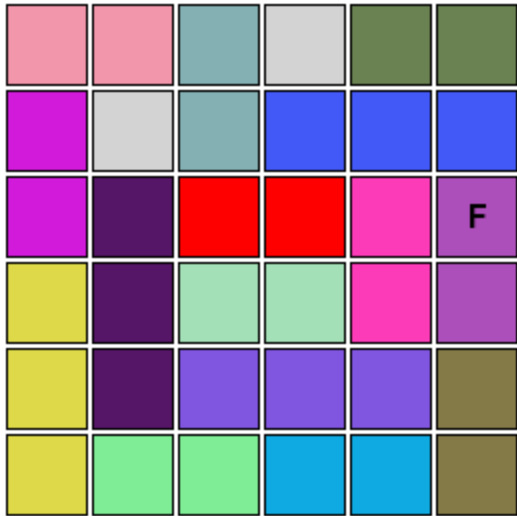
Hasil (IDA\*):




---

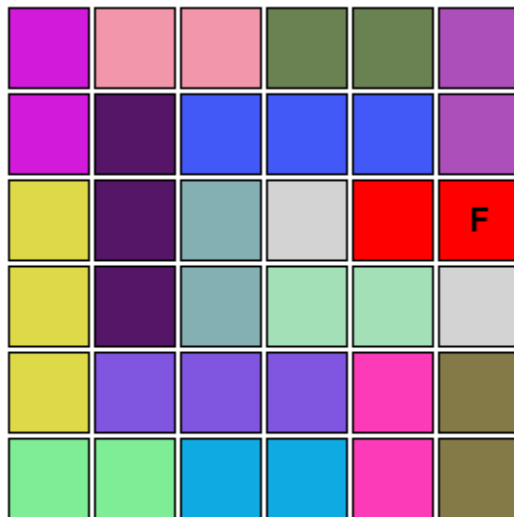
 Solved in 29718 visited nodes using IDA\* (Time: 8,28 seconds) Heuristic: Recursive

Konfigurasi 4:



Hasil (UCS):

---



Solved in 68 visited nodes using UCS (Time: 11 milliseconds) Heuristic: Recursive

## VI. ANALISIS PERCOBAAN

### 1. Hasil Pencarian Algoritma dan Heuristik

Dari penggunaan aplikasi Rush Hour Puzzle Solver, terdapat beberapa hal yang dihasilkan mengenai algoritma-algoritma *pathfinding* dan heuristic yang dimiliki. Dalam pencarian, GBFS secara tidak konsisten mengembalikan solusi yang valid dalam waktu tercepat, walaupun untuk kasus yang kompleks biasanya cukup tidak optimal. Jika diarahkan dengan heuristik yang baik, akan menemukan tujuan dalam waktu yang cepat, dan berlaku juga sebaliknya. Lalu, ini diikuti oleh IDA\*, yang mendapatkan solusi optimal dalam waktu yang cukup cepat, tetapi melakukan visit node yang berjumlah banyak karena pengulangan pencarian di setiap iterasi. Kemudian, A\* mendapatkan solusi optimal juga. Akhirnya, UCS yang memiliki waktu pencarian paling lambat, karena memeriksa setiap node satu persatu dalam bentuk BFS tanpa pengarahan. Hasil ini konsisten di pencarian tingkat tinggi yang menggunakan move di atas 20.

Dalam pemeriksaan heuristik, tidak ada pemenang yang jelas dalam melakukan pencarian. Ketiga heuristik memiliki momen ketika yang tercepat, tetapi untuk kompleksitas tinggi, umumnya heuristik Recursive memiliki keunggulan, karena menghitung keseluruhan balok yang saling memblokir. Walaupun tidak selalu admissible, Recursive belum mengembalikan hasil yang tidak optimal dalam pencarian.

### 2. Kompleksitas Algoritma

Dalam algoritma pencarian Uniform Cost Search. karena pada problem ini setara dengan BFS, kita akan menganalisis kompleksitas pencarian solusi dengan pendekatan BFS. Dalam pencarian yang menyebar (UCS, A\*, IDA\*), worst-case merupakan ketika pencarian akan melewati semua node (V) dan sisi (E) dari sebuah graf. Keseluruhan V dapat direpresentasikan oleh faktor percabangan b dan kedalaman solusi d dalam bentuk  $b^d$ , dan kita mengetahui di setiap node memiliki sisi sebanyak b. Jika dihitung, kompleksitas waktu dan ruang dari algoritma-algoritma tersebut adalah:

$$O(V + E) = O(b^d + b(d + 1)) \approx O(b^d)$$

Khusus IDA\*, kompleksitas ruang adalah  $O(bd)$  karena hanya menyimpan satu path dan setiap kedalaman, memproses setiap anak, berbeda dengan algoritma lainnya.

Greedy Best-First Search (GBFS) memiliki kompleksitas waktu dan ruang  $O(b^m)$ , dengan m adalah kedalaman maksimum dalam ruang pencarian. Meski dapat menemukan solusi lebih cepat daripada algoritma lain dalam banyak kasus karena tidak memperhatikan path lainnya, kompleksitasnya tetap eksponensial terhadap kedalaman

karena masih terdapat kemungkinan menjelajahi keseluruhan state yang mungkin dalam worst-case.

## VII. BONUS

### 1. Algoritma Pathfinding Tambahan (IDA\*)

Iterative Deepening A\* (IDA\*) adalah sebuah algoritma yang menggabungkan pendekatan Iterative Deepening dengan algoritma A\*. IDA\* bertujuan untuk mengatasi kelemahan A\* dalam hal penggunaan memori yang besar, dengan menggunakan strategi pencarian yang secara bertahap meningkatkan batas  $f(n)$ . IDA\* tidak menyimpan seluruh ruang pencarian dalam memori, tetapi hanya menyimpan path saat ini yang sedang dieksplorasi. Penjelasan lebih lanjutnya dapat dilihat di bab-bab sebelumnya.

### 2. Metode Heuristik Tambahan

Heuristik ini dijadikan pengarah dalam algoritma pathfinding A\*, IDA\* dan juga GBFS. Untuk setiap heuristik ditambahkan cost 1 jika Piece tidak terdapat di Goal. Ini untuk mensimulasikan gerakan terakhir menuju akhir ketika tidak ada balok yang menutupi jalan.

#### a. BlockCount

Heuristik ini menghitung jumlah balok yang menghalangi balok Player dari mencapai Goal. Ini hanya menghitung balok yang berpapasan di garis lurus Player menuju Goal. Heuristik ini admissible karena tidak mungkin melebihi jumlah gerakan balok yang dibutuhkan untuk mencapai Goal. Minimal satu gerakan per balok yang menghalangi Goal.

#### b. MaxDepth

Heuristik ini menghitung dependensi dorongan terdalam yang menghalangi balok Player dari mencapai Goal. Ini hanya menghitung balok yang terjebak dan tidak dapat gerak di hadapan Player menuju Goal. Ketika balok yang menghalangi Goal tidak dapat bergerak dan dihalangi oleh balok lain, itu menandakan kedalaman 2. Heuristik ini admissible karena minimal satu balok dengan dependensinya harus dihadapi untuk mencapai Goal.

#### c. Recursive








Heuristik ini menghitung jumlah keseluruhan dependensi dari balok-balok yang menghalangi balok Player dari mencapai Goal. Ketika terdapat balok A yang memerlukan balok B untuk digerakkan terlebih dahulu, jumlahkan jumlah dependensi balok B terhadap jumlah dependensi balok A. Heuristik ini tidak selalu admissible karena terdapat kemungkinan dua balok atau lebih yang menghalangi Goal untuk mempunyai dependensi ke balok yang sama.

### 3. GUI

Aplikasi ini menggunakan library JavaFX dan Maven untuk implementasi GUI. GUI ini memiliki 2 page yaitu page input file .txt dan juga graphical input. Pada page input file, user dapat memilih file .txt yang dimiliki sebagai konfigurasi. Untuk graphical input, player dapat memilih dimensi board dari puzzle, piece dari pool yang ada dan bisa

ditarik menuju board. Lalu ada bagian selektor algoritma maupun heuristic untuk user pilih. Lalu user dapat mengklik tombol “Solve” untuk menyelesaikan puzzle.

## VI. LAMPIRAN

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 dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt		
5. [Bonus] Implementasi algoritma pathfinding alternatif		
6. [Bonus] Implementasi 2 atau lebih heuristik alternatif		
7. [Bonus] Program memiliki GUI		
8. Program dan laporan dibuat (kelompok) sendiri		

Link repo: [https://github.com/pixelatedbus/Tucil3\\_13523064\\_13523084.git](https://github.com/pixelatedbus/Tucil3_13523064_13523084.git)

Spesifikasi:  Spesifikasi Tugas Kecil 3 Stima 2024/2025