

LAPORAN TUGAS KECIL 3 PENYELESAIAN PUZZLE RUSH HOUR MENGGUNAKAN ALGORITMA PATHFINDING

IF2211– Strategi Algoritma



Dosen:

Dr. Ir. Rinaldi, M.T.

Dr. Nur Ulfa Maulidevi, S.T, M.Sc

Anggota Kelompok:

Farrell Jabaar Altafataza - 10122057

Muhammad Adli Arindra - 18222089

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG**

2025

DAFTAR ISI

DAFTAR ISI.....	1
BAB I: DESKRIPSI TUGAS.....	1
BAB II: LANDASAN TEORI.....	2
Uniform Cost Search (UCS).....	2
Greedy Best First Search (GBFS).....	2
A* Search.....	2
BAB III: ANALISIS ALGORITMA.....	4
BAB IV: SOURCE PROGRAM.....	5
Main.java.....	5
Board.java.....	6
Piece.java.....	10
Benchmark.java.....	11
Data.java.....	12
Direction.java.....	14
Listener.java.....	14
ReadInput.java.....	14
BoardPanel.java.....	17
RootWindow.java.....	20
Heuristic.java.....	24
Pathfinder.java.....	26
BAB V: PENGUJIAN PROGRAM.....	28
Test Case 1.....	28
Test Case 2.....	29
Test Case 3.....	29
Test Case 4.....	30
Anailisis Hasil Pengujian.....	31
LAMPIRAN.....	32
DAFTAR PUSTAKA.....	32

BAB I: DESKRIPSI TUGAS

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.

Tugas ini bertujuan untuk mengimplementasikan algoritma *pathfinding* dalam konteks permainan Rush Hour. Akan dibangun sebuah program yang dapat menentukan urutan langkah-langkah kombinasi gerakan untuk menyelesaikan puzzle tersebut. Algoritma yang akan digunakan pada tugas kali ini adalah algoritma Greedy Best First Search, Uniform Cost Search, dan A*.

BAB II: LANDASAN TEORI

Uniform Cost Search (UCS)

Uniform Cost Search (UCS) merupakan algoritma pencarian yang memprioritaskan jalur dengan biaya kumulatif terendah dari simpul awal ke simpul saat ini. UCS menggunakan struktur data *priority queue* yang selalu memproses simpul dengan total biaya paling kecil terlebih dahulu. Proses terus berlanjut hingga simpul tujuan ditemukan, dan karena jalur dipilih berdasarkan biaya terkecil, UCS menjamin hasil yang optimal selama semua biaya lintasan bernilai non-negatif.

Greedy Best First Search (GBFS)

Greedy Best First Search adalah algoritma yang mengandalkan heuristik untuk memperkirakan kedekatan suatu simpul ke simpul tujuan. Fungsi evaluasi yang digunakan adalah $f(n) = h(n)$, di mana $h(n)$ adalah estimasi biaya dari simpul saat ini ke tujuan. Algoritma ini memilih simpul yang tampak paling dekat ke tujuan, tanpa mempertimbangkan biaya yang sudah dikeluarkan. GBFS biasanya lebih cepat dibanding UCS, namun tidak menjamin solusi optimal.

A* Search

A* merupakan algoritma pencarian yang mengkombinasikan keunggulan UCS dan GBFS melalui fungsi evaluasi total $f(n) = h(n) + g(n)$, dengan:

- $g(n)$: biaya aktual dari simpul awal ke simpul n ,
- $h(n)$: estimasi biaya dari simpul n ke tujuan.

Dengan pendekatan ini, A* mampu menemukan solusi yang optimal asalkan fungsi heuristiknya bersifat admissible (tidak melebihi biaya sebenarnya ke tujuan) dan consistent (tidak melanggar ketaksamaan segitiga). A* sangat efektif dalam ruang pencarian yang kompleks, namun membutuhkan sumber daya memori yang lebih besar.

BAB III: ANALISIS ALGORITMA

Fungsi evaluasi yang digunakan pada masing-masing algoritma menunjukkan perbedaan pendekatan yang dilakukan. Fungsi evaluasi dilambangkan dengan $f(n) = g(n) + h(n)$ dengan $g(n)$ merupakan *cost* atau biaya yang diperlukan untuk mencapai simpul n dan $h(n)$ merupakan estimasi *cost* dari simpul n ke tujuan. Pada UCS, fungsi evaluasi yang digunakan adalah $f(n) = g(n)$. Algoritma ini tidak menggunakan heuristik dan hanya fokus pada jalur dengan biaya terkecil. Pada GBFS, fungsi evaluasi yang digunakan adalah $f(n) = h(n)$. GBFS tidak mempertimbangkan biaya yang sudah ditempuh, melainkan hanya melihat seberapa dekat simpul saat ini dengan tujuan secara heuristik. Sedangkan pada A*, fungsi evaluasi yang digunakan merupakan kombinasi dari keduanya, yaitu $f(n) = g(n) + h(n)$, sehingga algoritma ini mempertimbangkan baik biaya aktual maupun estimasi ke depan.

Heuristik yang digunakan pada program ini adalah *blockingTiles*, yang menghitung jumlah blok yang menghalangi jalan keluar, dan *distanceToExit*, yang menggunakan jarak Manhattan dari posisi *piece* utama ke tujuan.

Dalam algoritma A*, kualitas solusi yang dihasilkan oleh algoritma ini sangat bergantung kepada heuristik. Kedua heuristik ini bersifat admissible karena tidak pernah melebihi biaya sebenarnya untuk mencapai tujuan. Sebagai contoh, `blockingTiles` hanya menghitung jumlah blok yang menghalangi jalur *piece* utama, yang secara realistis menunjukkan paling tidak dibutuhkan satu langkah untuk memindahkan tiap blok tersebut. Sementara itu, `distanceToExit` menggunakan jarak terpendek tanpa mempertimbangkan hambatan, sehingga juga tidak akan melebihi cost sebenarnya.

Dalam algoritma UCS, solusi yang dihasilkan dapat dikatakan serupa dengan algoritma BFS karena semua biaya langkah identik sehingga $g(n)$ di UCS akan sama dengan kedalaman *node* dalam pohon pencarian. Artinya, pada konteks ini, UCS sama dengan BFS.

Secara teoritis, A* lebih efisien dibandingkan UCS. A* tidak hanya menghitung *cost* ($g(n)$), tetapi juga $h(n)$ untuk bertindak. Hal ini menyebabkan keputusan yang diambil oleh algoritma A* lebih fokus untuk menuju arah tujuan dan lebih sedikit mengeksplorasi jalan yang tidak relevan. Meskipun UCS optimal, sering kali algoritma ini mengeksplorasi banyak *nodes* yang tidak terlalu relevan karena tidak memiliki panduan ke arah tujuan.

Di sisi lain, Greedy Best First Search hanya memperhitungkan heuristik saja tanpa mempertimbangkan biaya perjalanan ($g(n)$). Sering kali GBFS mengeksplorasi jalur yang terlihat ‘dekat’ dengan arah tujuan, tanpa mengetahui apakah jalur tersebut efektif secara keseluruhan. Meskipun GBFS dapat mencapai solusi dengan lebih cepat di berbagai kasus, GBFS tidak memberikan jaminan bahwa solusi yang ditemukan merupakan solusi yang optimal.

BAB IV: SOURCE PROGRAM

Main.java

```
import element.Board;
import element.Piece;
import java.util.List;
import model.Data;
import pathfinder.Pathfinder;
import utils.Listener;
import utils.ReadInput;
import view.RootWindow;
```

```

public class Main {
    public static RootWindow window;
    public static Data data;
    public static Pathfinder path;

    public static void main(String[] args) {
        data = new Data();
        path = new Pathfinder(data);
        Listener listener = new Listener() {
            @Override
            public void onSearch(String algorithm, String heuristic) {
                Main.searchSolution(algorithm, heuristic);
            }
            @Override
            public void onFileSelected(String filePath) {
                Main.readInput(filePath);
            }
        };

        readInput("test/1.txt");
        searchSolution("", "");

        javax.swing.SwingUtilities.invokeLater(() -> {
            window = new RootWindow(data, listener);
            window.showWindow();
        });
    }

    private static void readInput(String path) {
        ReadInput reader = new ReadInput();
        reader.read(path);

        Board board = new Board(reader.getBoardSize()[0],
reader.getBoardSize()[1], reader.getExit());
        List<Piece> pieces = reader.getPieces();

        for (Piece piece : pieces) {
            board.addPiece(piece);
        }

        data.setInitialBoard(board);
    }

    public static void searchSolution(String algorithm, String
heuristic) {

```

```

        data.benchmark.startTimer();
        List<Board> solutionSteps = path.search(data.getInitialBoard());
        data.setSolutionSteps(solutionSteps);
        data.benchmark.stopTimer();
        if (window != null) window.updateBoardAndLabel();
    }
}

```

Board.java

```

package element;

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

public class Board {
    public List<Piece> pieces;
    public int a, b;
    private final char exit;

    public Board(int a, int b, char exit) {
        this.pieces = new ArrayList<>();
        this.a = a;
        this.b = b;
        this.exit = exit;
    }

    public void addPiece(Piece piece) {
        if (piece != null) {
            pieces.add(piece);
        }
    }

    public void resetPieces() {
        pieces.clear();
    }

    public char getExit() {
        return this.exit;
    }

    public char[][] getBoard() {
        char[][] mat = new char[b][a];
        for (int i = 0; i < b; i++) {

```



```

        for (int j = 0; j < a; j++) {
            mat[i][j] = '.';
        }
    }

    for (Piece piece : this.pieces) {
        char label = piece.label;
        char orientation = piece.orientation;
        int x = piece.x;
        int y = piece.y;
        int length = piece.length;

        if (orientation == 'H') {
            for (int i = 0; i < length; i++) {
                mat[y][x + i] = label;
            }
        } else if (orientation == 'V') {
            for (int i = 0; i < length; i++) {
                mat[y + i][x] = label;
            }
        }
    }

    return mat;
}

public Piece getPrimaryPiece() {
    for (Piece piece : this.pieces) {
        if (piece.label == 'P') {
            return piece;
        }
    }

    throw new IllegalStateException("Main piece not found!");
}

public void print() {
    char[][] mat = getBoard();
    for (char[] row : mat) {
        for (char c : row) {
            System.out.print(c + " ");
        }
        System.out.println();
    }
}
}

```

```

public long getHash() {
    long ret = 17;
    for (Piece piece : this.pieces) {
        ret = ret * 31 + Integer.toUnsignedLong(piece.getHash());
    }
    return ret;
}

public int getWidth() {
    return this.a;
}

public int getHeight() {
    return this.b;
}

public boolean isGoal(){
    Piece primaryPiece = getPrimaryPiece();
    switch(exit) {
        case 'U' -> {
            return primaryPiece.orientation == 'V' && primaryPiece.y
== 0;
        }
        case 'D' -> {
            return primaryPiece.orientation == 'V' &&
(primaryPiece.y + primaryPiece.length) >= getHeight();
        }
        case 'L' -> {
            return primaryPiece.orientation == 'H' && primaryPiece.x
== 0;
        }
        case 'R' -> {
            return primaryPiece.orientation == 'H' &&
(primaryPiece.x + primaryPiece.length) >= getWidth();
        }
        default -> throw new IllegalArgumentException("Invalid exit
direction: " + exit);
    }
}

public Board copy() {
    Board copy = new Board(this.a, this.b, this.exit);

    for (Piece piece : this.pieces) {
        copy.pieces.add(piece.copy());
    }
    return copy;
}

```

```

    }

    public List<Board> generateNeighbors() {
        List<Board> ret = new ArrayList<>();
        int height = this.b;
        int width = this.a;
        char[][] mat = this.getBoard();

        for (int i = 0; i < this.pieces.size(); ++i) {
            Piece piece = this.pieces.get(i);

            int[] posPlus = piece.positionPlus();
            int xPlus = posPlus[0];
            int yPlus = posPlus[1];

            Board boardPlus = this.copy();
            if (xPlus >= 0 && xPlus < width && yPlus >= 0 && yPlus <
height && mat[yPlus][xPlus] == '.') {
                boardPlus.pieces.get(i).movePlus();
                ret.add(boardPlus);
            }

            int[] posMinus = piece.positionMinus();
            int xMinus = posMinus[0];
            int yMinus = posMinus[1];

            Board boardMinus = this.copy();
            if (xMinus >= 0 && xMinus < width && yMinus >= 0 && yMinus <
height && mat[yMinus][xMinus] == '.') {
                boardMinus.pieces.get(i).moveMinus();
                ret.add(boardMinus);
            }
        }

        return ret;
    }
}

```

Piece.java

```

package element;

import java.util.Objects;

```

```

public class Piece {
    public char label, orientation; // 'H' or 'V'
    public int x, y, length;
    public Piece(char label, char orientation, int x, int y, int length) {
        this.label = label;
        this.orientation = orientation;
        this.x = x;
        this.y = y;
        this.length = length;
    }

    public int getHash() {
        return Objects.hash(label, orientation, x, y, length);
    }

    public int[] positionPlus() {
        return orientation == 'H'
            ? new int[] { x + length, y }
            : new int[] { x, y + length };
    }

    public int[] positionMinus() {
        return orientation == 'H'
            ? new int[] { x - 1, y }
            : new int[] { x, y - 1 };
    }

    public void movePlus() {
        if (orientation == 'H') {
            x += 1;
        } else if (orientation == 'V') {
            y += 1;
        }
    }

    public void moveMinus() {
        if (orientation == 'H') {
            x -= 1;
        } else if (orientation == 'V') {
            y -= 1;
        }
    }

    public Piece copy() {
        return new Piece(label, orientation, x, y, length);
    }
}

```

Benchmark.java

```
package model;

public class Benchmark {
    private int visitedNodes = 0;
    private long startTime = 0;
    private long elapsedTime = 0;

    public synchronized int getVisitedNodes() {
        return this.visitedNodes;
    }

    public synchronized void setVisitedNodes(int nodes) {
        this.visitedNodes = nodes;
    }

    public synchronized void reset() {
        this.visitedNodes = 0;
        this.startTime = 0;
        this.elapsedTime = 0;
    }

    public synchronized void startTimer() {
        this.startTime = System.nanoTime();
    }

    public synchronized void stopTimer() {
        if (this.startTime != 0) {
            this.elapsedTime = System.nanoTime() - this.startTime;
        }
    }

    public synchronized long getElapsedTimeMillis() {
        return this.elapsedTime / 1_000_000;
    }

    public synchronized long getElapsedTimeMicros() {
        return this.elapsedTime / 1_000;
    }

    public synchronized long getElapsedTimeNanos() {
        return this.elapsedTime;
    }
}
```

```
}
```

Data.java

```
package model;

import element.Board;
import java.util.ArrayList;
import java.util.List;

public class Data {
    private Board initialBoard;
    private List<Board> solutionSteps = new ArrayList<>();
    private int currentStepIndex = 0;
    private String heuristicMethod = "Distance"; // Distance, Tiles
    private String searchMethod = "UCS"; // UCS, GBFS, A*
    public Benchmark benchmark;

    public Data() {
        this.benchmark = new Benchmark();
    }

    public synchronized Board getInitialBoard() {
        return initialBoard;
    }

    public synchronized void setInitialBoard(Board initialBoard) {
        this.initialBoard = initialBoard;
    }

    public synchronized List<Board> getSolutionSteps() {
        return solutionSteps;
    }

    public synchronized void setSolutionSteps(List<Board> steps) {
        this.solutionSteps = steps;
        this.currentStepIndex = 0;
    }

    public synchronized int getCurrentStepIndex() {
        return currentStepIndex;
    }
}
```

```

    }

    public synchronized void setCurrentStepIndex(int index) {
        this.currentStepIndex = index;
    }

    public synchronized String getHeuristicMethod() {
        return this.heuristicMethod;
    }

    public synchronized void setHeuristicMethod(String heuristicMethod)
    {
        this.heuristicMethod = heuristicMethod;
    }

    public synchronized String getSearchMethod() {
        return this.searchMethod;
    }

    public synchronized void setSearchMethod(String searchMethod) {
        this.searchMethod = searchMethod;
    }
}

```

Direction.java

```

package utils;

public enum Direction {
    UP(-1, 0),
    DOWN(1, 0),
    LEFT(0, -1),
    RIGHT(0, 1);

    public final int dx;
    public final int dy;

    Direction(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
    }
}

```

Listener.java

```
package utils;

public interface Listener {
    void onSearch(String algorithm, String heuristic);
    void onFileSelected(String filePath);
}
```

ReadInput.java

```
package utils;

import element.Piece;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Scanner;
import java.util.Set;

public class ReadInput {
    private int rows;
    private int cols;
    private int pieceCount;
    private List<Piece> pieces;
    private char[][] board;
    private char exit;

    public ReadInput() {
        pieces = new ArrayList<>();
        exit = 'N';
    }

    public void read(String filePath) {
        try (Scanner scanner = new Scanner(new File(filePath))) {
```



```

        rows = scanner.nextInt();
        cols = scanner.nextInt();
        scanner.nextLine();

        pieceCount = scanner.nextInt();

        int i = 0, j = 0;
        board = new char[rows][cols];

        while (scanner.hasNext()) {
            String token = scanner.nextLine();
            Boolean first = true;
            for (char c : token.toCharArray()) {
                if (c == ' ') continue;

                if (c == 'K') {
                    if (j == 0)
                        exit = 'U';
                    else if (j == rows - 1)
                        exit = 'D';
                    else if (first)
                        exit = 'L';
                    else if (i == cols)
                        exit = 'R';
                }
                else {
                    if (i >= cols) {
                        i = 0;
                        j++;
                    }
                    board[j][i] = c;
                    i++;
                }
                first = false;
            }
        }

        pieces = parsePieces();

    } catch (FileNotFoundException e) {
        System.err.println("File not found: " + filePath);
    } catch (Exception e) {
    }
}

public char getExit() {

```

```

        return exit;
    }

    private List<Piece> parsePieces() {
        List<Piece> result = new ArrayList<>();
        Set<Character> seen = new HashSet<>();

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                char label = board[i][j];
                if (label == '.' || seen.contains(label))
                    continue;

                int length = 1;
                char orientation;

                if (j + 1 < cols && board[i][j + 1] == label) {
                    orientation = 'H';
                    int col = j + 1;
                    while (col < cols && board[i][col] == label) {
                        length++;
                        col++;
                    }
                }
                else if (i + 1 < rows && board[i + 1][j] == label) {
                    orientation = 'V';
                    int row = i + 1;
                    while (row < rows && board[row][j] == label) {
                        length++;
                        row++;
                    }
                }
                else {
                    orientation = 'H';
                }

                result.add(new Piece(label, orientation, j, i, length));
                seen.add(label);
            }
        }
        return result;
    }

    public List<Piece> getPieces() {

```

```

        return pieces;
    }

    public int getPieceCount() {
        return pieceCount;
    }

    public int[] getBoardSize() {
        return new int[] { rows, cols };
    }
}

```

BoardPanel.java

```

package view;

import element.Board;
import java.awt.*;
import java.util.*;
import javax.swing.*;

public class BoardPanel extends JPanel {
    private Board board;
    private final Map<Character, Color> colorMap = new HashMap<>();

    public BoardPanel(Board board) {
        this.board = board;
        generateColors();
    }

    private void generateColors() {
        Random rand = new Random();

        for (char[] row : board.getBoard()) {
            for (char c : row) {
                if (c == 'P') continue;
                if (!colorMap.containsKey(c)) {
                    Color color = new Color(rand.nextInt(200),
rand.nextInt(200), rand.nextInt(200));
                    colorMap.put(c, color);
                }
            }
        }
    }
}

```

```

    }
}

public void setBoard(Board newBoard) {
    this.board = newBoard;
    generateColors();
    repaint();
}

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (board == null) return;

    int rows = board.a;
    int cols = board.b;
    int cellWidth = 50;
    int cellHeight = 50;

    g.setFont(new Font("Monospaced", Font.BOLD, Math.min(cellWidth,
cellHeight) / 2));

    char[][] mat = board.getBoard();
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            char cellChar = mat[i][j];
            int x = j * cellWidth;
            int y = i * cellHeight;

            switch (cellChar) {
                case 'P' -> g.setColor(Color.RED);
                case '.' -> g.setColor(Color.darkGray);
                default ->
g.setColor(colorMap.getOrDefault(cellChar, Color.LIGHT_GRAY));
            }

            g.fillRect(x, y, cellWidth, cellHeight);
            g.setColor(Color.BLACK);
            g.drawRect(x, y, cellWidth, cellHeight);
            g.drawString(String.valueOf(cellChar), x + cellWidth / 2
- 4, y + cellHeight / 2 + 5);
        }
    }

    char exit = board.getExit();
    g.setColor(Color.RED);

```

```

        int width = cols * cellWidth;
        int height = rows * cellHeight;
        int borderThickness = 10;

        switch (exit) {
            case 'L' -> g.fillRect(0, 0, borderThickness, height);
            case 'R' -> g.fillRect(width - borderThickness, 0,
borderThickness, height);
            case 'U' -> g.fillRect(0, 0, width, borderThickness);
            case 'D' -> g.fillRect(0, height - borderThickness,
width, borderThickness);
        }
    }

    @Override
    public Dimension getPreferredSize() {
        if (board == null) return new Dimension(100, 100);
        int cellSize = 50;
        int width = board.b * cellSize;
        int height = board.a * cellSize;
        return new Dimension(width, height);
    }
}

```

RootWindow.java

```

package view;

import element.Board;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.util.List;
import javax.swing.*;
import model.Data;
import utils.Listener;

public final class RootWindow extends JFrame {
    private JComboBox<String> algorithmDropdown;
    private JComboBox<String> heuristicDropdown;
    private final BoardPanel boardPanel;

    private final Data data;
}

```

```

private int currentStep = 0;
private Timer playbackTimer;

private JButton prevButton;
private JButton nextButton;
private JLabel indexLabel;
private JButton resetButton;

private JLabel nodesLabel;
private JLabel timeLabel;

public RootWindow(Data data, Listener listener) {
    this.data = data;

    setTitle("Game Board");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(600, 700);
    setLocationRelativeTo(null);

    boardPanel = new BoardPanel(data.getInitialBoard());

    JPanel centerPanel = new JPanel(new GridBagLayout());
    centerPanel.add(boardPanel);
    add(centerPanel, BorderLayout.CENTER);

    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new BoxLayout(controlPanel,
BoxLayout.Y_AXIS));

    JPanel infoPanel = new JPanel(new
FlowLayout(FlowLayout.CENTER));
    nodesLabel = new JLabel("Visited Nodes: 0");
    timeLabel = new JLabel("Time: 0 ms");
    infoPanel.add(nodesLabel);
    infoPanel.add(timeLabel);

    controlPanel.add(infoPanel);

    JPanel row1 = new JPanel(new FlowLayout(FlowLayout.CENTER));
    JButton loadFileButton = new JButton("Load New File");
    row1.add(loadFileButton);

    algorithmDropdown = new JComboBox<>(new String[] { "GBFS",
"UCS", "A*" });
    heuristicDropdown = new JComboBox<>(new String[] { "Distance",

```

```

"Tiles" });
    JButton searchButton = new JButton("Search");

    row1.add(new JLabel("Algorithm:"));
    row1.add(algorithmDropdown);

    row1.add(new JLabel("Heuristic:"));
    row1.add(heuristicDropdown);

    row1.add(searchButton);

    JPanel row2 = new JPanel(new FlowLayout(FlowLayout.CENTER));

    JButton playButton = new JButton("Play");
    JButton pauseButton = new JButton("Pause");
    JButton stepButton = new JButton("Step");

    row2.add(playButton);
    row2.add(pauseButton);
    row2.add(stepButton);

    prevButton = new JButton("-");
    nextButton = new JButton("+");
    indexLabel = new JLabel("0");
    resetButton = new JButton("Reset");

    row2.add(prevButton);
    row2.add(indexLabel);
    row2.add(nextButton);
    row2.add(resetButton);

    controlPanel.add(row1);
    controlPanel.add(row2);

    loadFileButton.addActionListener(e -> {
        JFileChooser fileChooser = new JFileChooser(new
java.io.File("test"));
        fileChooser.setFileFilter(new
javax.swing.filechooser.FileNameExtensionFilter("Text Files", "txt"));
        int result = fileChooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            String selectedFilePath =
fileChooser.getSelectedFile().getAbsolutePath();
            if (listener != null) {
                listener.onFileSelected(selectedFilePath);
            }
        }
    });

```

```

        }
    });

    algorithmDropdown.addActionListener(e -> {
        String selected = (String)
algorithmDropdown.getSelectedItem();
        if (selected != null) {
            data.setSearchMethod(selected);
        }
    });

    heuristicDropdown.addActionListener(e -> {
        String selected = (String)
heuristicDropdown.getSelectedItem();
        if (selected != null) {
            data.setHeuristicMethod(selected);
        }
    });

    searchButton.addActionListener((ActionEvent e) -> {
        String algorithm = (String)
algorithmDropdown.getSelectedItem();
        String heuristic = (String)
heuristicDropdown.getSelectedItem();
        if (listener != null) {
            listener.onSearch(algorithm, heuristic);
        }
    });

    playButton.addActionListener(e -> {
        List<Board> steps = data.getSolutionSteps();
        if (steps == null || steps.isEmpty()) return;
        if (playbackTimer != null && playbackTimer.isRunning())
return;

        playbackTimer = new Timer(500, evt -> {
            if (currentStep < steps.size() - 1) {
                currentStep++;
                updateBoardAndLabel();
            } else {
                playbackTimer.stop();
            }
        });
        playbackTimer.start();
    });
});

```



```

        pauseButton.addActionListener(e -> {
            if (playbackTimer != null) {
                playbackTimer.stop();
            }
        });

        stepButton.addActionListener(e -> {
            List<Board> steps = data.getSolutionSteps();
            if (steps == null || currentStep >= steps.size() - 1)
return;

            currentStep++;
            updateBoardAndLabel();
        });

        prevButton.addActionListener(e -> {
            if (currentStep > 0) {
                currentStep--;
                updateBoardAndLabel();
            }
        });

        nextButton.addActionListener(e -> {
            List<Board> steps = data.getSolutionSteps();
            if (steps != null && currentStep < steps.size() - 1) {
                currentStep++;
                updateBoardAndLabel();
            }
        });

        resetButton.addActionListener(e -> {
            currentStep = 0;
            updateBoardAndLabel();
        });

        setLayout(new BorderLayout());
        add(centerPanel, BorderLayout.CENTER);
        add(controlPanel, BorderLayout.SOUTH);

        updateBoardAndLabel();
    }

    public void updateBoardAndLabel() {
        List<Board> steps = data.getSolutionSteps();
        if (steps != null && !steps.isEmpty() && currentStep >= 0 &&
currentStep < steps.size()) {
            setBoard(steps.get(currentStep));

```

```

        indexLabel.setText(String.valueOf(currentStep));
        nodesLabel.setText("Visited Nodes: " +
data.benchmark.getVisitedNodes());
        timeLabel.setText("Time: " +
data.benchmark.getElapsedTimeMillis() + " ms");
    }
}

public void showWindow() {
    setVisible(true);
}

public void setBoard(Board board) {
    boardPanel.setBoard(board);
}
}

```

Heuristic.java

```

package pathfinder;

import element.*;

public class Heuristic {
    public static int distanceToExit(Board board) {
        Piece primary = board.getPrimaryPiece();
        int dist = 0;

        switch (board.getExit()) {
            case 'L' -> dist = primary.x;
            case 'R' -> dist = board.getWidth() - (primary.x +
primary.length);
            case 'U' -> dist = primary.y;
            case 'D' -> dist = board.getHeight() - (primary.y +
primary.length);
        }
        return Math.max(dist, 0);
    }

    public static int blockingTiles(Board board) {
        Piece primary = board.getPrimaryPiece();
        char[][] mat = board.getBoard();
        int count = 0;
    }
}

```

```

switch (board.getExit()) {
    case 'L' -> {
        int row = primary.y;
        for (int col = primary.x - 1; col >= 0; col--) {
            if (mat[row][col] != '.') count++;
        }
    }
    case 'R' -> {
        int row = primary.y;
        int startCol = primary.x + primary.length;
        for (int col = startCol; col < board.getWidth(); col++)
        {
            if (mat[row][col] != '.') count++;
        }
    }
    case 'U' -> {
        int col = primary.x;
        for (int row = primary.y - 1; row >= 0; row--) {
            if (mat[row][col] != '.') count++;
        }
    }
    case 'D' -> {
        int col = primary.x;
        int startRow = primary.y + primary.length;
        for (int row = startRow; row < board.getHeight(); row++)
        {
            if (mat[row][col] != '.') count++;
        }
    }
}
return count;
}
}

```

Pathfinder.java

```

package pathfinder;

import element.Board;
import java.util.*;
import model.Data;

```

```

public class Pathfinder {
    private final PriorityQueue<State> queue;
    private final Data data;

    public Pathfinder(Data data) {
        this.queue = new
PriorityQueue<>(Comparator.comparingDouble(state -> state.cost));
        this.data = data;
    }

    private static class State {
        public Board board;
        public double cost;
        public State parent;

        State(Board board, double cost, State parent) {
            this.board = board;
            this.cost = cost;
            this.parent = parent;
        }
    }

    public List<Board> search(Board board) {
        queue.clear();
        double startcost = 0;
        queue.add(new State(board, startcost, null));
        HashMap<Long, Boolean> visited = new HashMap<>();
        visited.put(board.getHash(), true);
        int visitedNodes = 0;

        while (!queue.isEmpty()) {
            State current = queue.poll();
            Board currentBoard = current.board;
            visitedNodes ++;

            if (currentBoard.isGoal()) {
                List<Board> path = new ArrayList<>();
                State node = current;
                while (node != null) {
                    path.add(node.board);
                    node = node.parent;
                }
                Collections.reverse(path);
                data.benchmark.setVisitedNodes(visitedNodes);
                return path;
            }
        }
    }
}

```

```

    }

    List<Board> neighbors = currentBoard.generateNeighbors();

    for (Board neighbor : neighbors) {
        long neighborHash = neighbor.getHash();
        if (!visited.containsKey(neighborHash)) {
            double cost = getCost(neighbor, current);
            State nextState = new State(neighbor, cost,
current);

            queue.add(nextState);
            visited.put(neighborHash, true);
        }
    }

    return new ArrayList<>();
}

// UCS, GBFS, A*
private double getCost(Board board, State state) {
    String method = data.getSearchMethod();
    if (null != method) switch (method) {
        case "UCS" -> {
            return state.cost + 1;
        }
        case "GBFS" -> {
            return getHeuristic(board);
        }
        case "A*" -> {
            return getHeuristic(board) + state.cost + 1;
        }
        default -> {
        }
    }

    return 0.0;
}

// Distance, Tiles
private double getHeuristic(Board board) {
    if (data.getHeuristicMethod().equals("Distance")) {
        return Heuristic.blockingTiles(board);
    }
    else {
        return Heuristic.distanceToExit(board);
    }
}

```

```

    }
  }
}

```






BAB V: PENGUJIAN PROGRAM

Test Case 1

```

6 6
12
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.

```

Algoritma	Heuristik	Hasil
GBFS	Distance	 GBFS - Distance.mp4
GBFS	Tiles	 GBFS - Tiles
UCS	-	 1 - UCS - Distance.mp4
A*	Distance	 1 - A - Distance.mp4
A*	Tiles	 1 - A - Tiles.mp4






Test Case 2

```

6 6
12
AAB..F
.BBCDF
KGPPC.F
GH.III
.HJ...






```

.LJMM.

Algoritma	Heuristik	Hasil
GBFS	Distance	 2 - GBFS - Distance.mp4
GBFS	Tiles	 2 - GBFS - Tiles.mp4
UCS	-	 2 - UCS - Distance.mp4
A*	Distance	 2 - A - Distance.mp4
A*	Tiles	 2 - A - Tiles.mp4






Test Case 3

6 6
9
K
AA..BB
CP.D.B
CP.DEE
CP.DFF
GGG..H
.....

Algoritma	Heuristik	Hasil
GBFS	Distance	 3 - GBFS - Distance.mp4
GBFS	Tiles	 3 - GBFS - Tiles.mp4
UCS	-	 3 - UCS - Distance.mp4
A*	Distance	 3 - A - Distance.mp4
A*	Tiles	 3 - A - Tiles.mp4

Test Case 4

```
6 6
9
AA..BB
CP.D.B
CP.DEE
CP.DFF
GGG..H
.....
K
```

Algoritma	Heuristik	Hasil
GBFS	Distance	 4 - GBFS - Distance.mp4
GBFS	Tiles	 4 - GBFS - Tiles.mp4
UCS	-	 4 - UCS - Distance.mp4
A*	Distance	 4 - A - Distance.mp4
A*	Tiles	 4 - A - Tiles.mp4

Anailisis Hasil Pengujian

Pengujian dilakukan dengan menggunakan empat test case yang merepresentasikan berbagai konfigurasi puzzle Rush Hour. Masing-masing test case diuji dengan tiga algoritma pencarian jalur yaitu Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A*, dengan dua pilihan heuristik: Tiles (*blocking tiles*) dan Distance (*distance to exit*). Pengamatan terhadap video hasil eksekusi menunjukkan variasi yang signifikan dalam hal waktu pencarian dan efisiensi jalur tergantung pada algoritma dan heuristik yang digunakan.

Program berhasil dijalankan dengan baik pada keempat test case yang disediakan. Secara umum, waktu eksekusi cukup cepat, terutama pada konfigurasi puzzle yang sederhana atau memiliki banyak ruang gerak. Algoritma UCS dan A* menunjukkan performa stabil dan akurat dalam menemukan solusi, sedangkan GBFS memiliki keunggulan dalam kecepatan pada beberapa skenario tertentu, meskipun terkadang mengorbankan keoptimalan jalur.

Namun, terdapat beberapa kasus di mana program tidak berhasil menemukan jalur yang optimal. Misalnya, pada konfigurasi puzzle yang kompleks, algoritma cenderung "berputar-putar" atau mengeksplorasi simpul yang tidak mendekatkan ke tujuan, terutama saat menggunakan heuristik yang kurang akurat. Hal ini terjadi meskipun posisi mobil merah sudah dekat dengan pintu keluar, menunjukkan bahwa proses pencarian belum cukup efisien dalam mengenali peluang solusi cepat.

Untuk meningkatkan performa program, beberapa pendekatan dapat diterapkan. Salah satunya adalah merancang heuristik yang lebih informatif dan sesuai dengan karakteristik puzzle Rush Hour, misalnya dengan mempertimbangkan tidak hanya jumlah penghalang tetapi juga posisi relatif dan potensi pergerakan mobil penghalang. Selain itu, optimalisasi struktur data dan pemangkasan simpul yang tidak produktif (pruning) dapat mengurangi eksplorasi berlebih dan mempercepat proses pencarian. Penerapan metode hybrid atau adaptif juga dapat dieksplorasi untuk mengombinasikan kelebihan masing-masing algoritma.

LAMPIRAN

[Tautan GitHub](#)

DAFTAR PUSTAKA

1. Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
2. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*.