LAPORAN TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Disusun oleh:

Nama	**NIM**
Michael Alexander Angkawijaya	13523102
Aria Judhistira	13523112

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

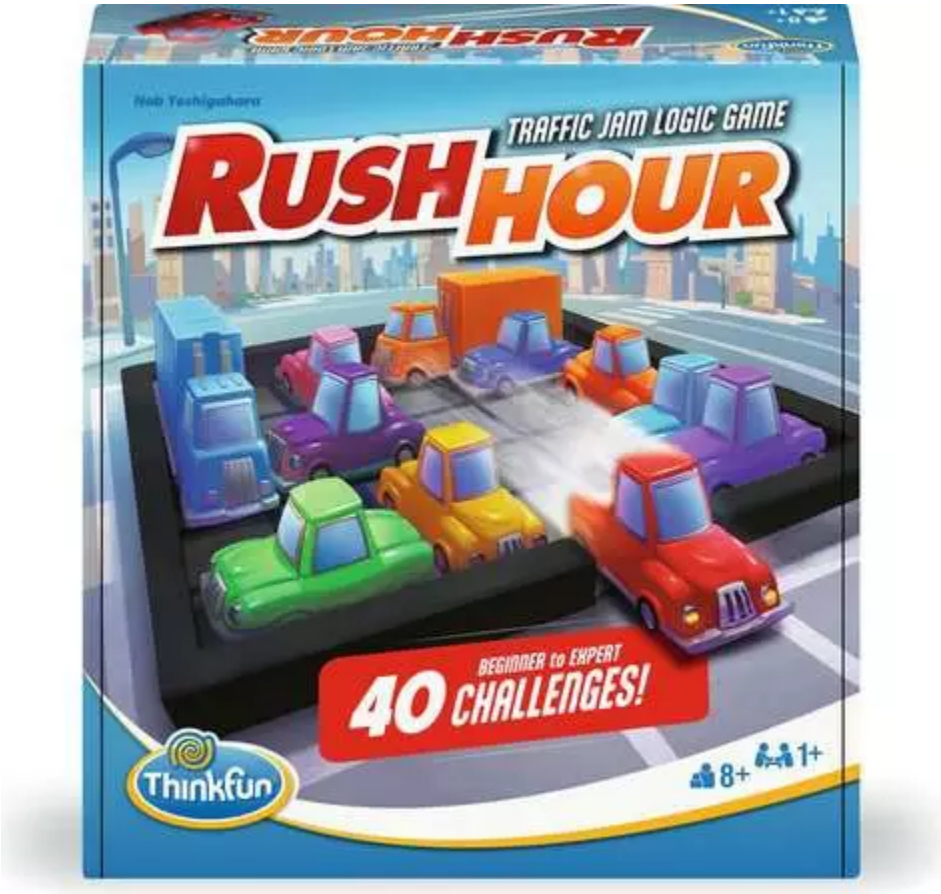
INSTITUT TEKNOLOGI BANDUNG

JL. GANESA 10, BANDUNG 40132

2025

DAFTAR ISI

BAB 1: Deskripsi Tugas



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.

BAB 2: Landasan Teori

2.1 Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search (UCS) merupakan algoritma pencarian rute yang mengunjungi *node-node* pada graf berbobot dengan prinsip *best-first*. Pada algoritma UCS, berdasarkan prinsip tersebut, pencarian rute terpendek dilakukan dengan mencari jalur dengan biaya jalur dari *node* awal hingga *node* tujuan yang paling minimum. Berikut adalah langkah-langkah penerapan algoritma UCS untuk penyelesaian puzzle ini:

1. Masukkan *state* awal ke dalam Priority Queue.
2. Siapkan set berisi *state* yang sudah pernah dikunjungi dan map untuk mencatat jalur antara *state*.
3. Ambil *state* dengan biaya terkecil pada Priority Queue. Biaya terkecil dalam hal ini adalah kedalaman *node*.
4. Periksa apakah *state* sudah memenuhi kondisi tujuan. Jika sudah, maka pencarian selesai.
5. Jika belum, jabarkan setiap kemungkinan *state* berikutnya berdasarkan *state* saat ini. Jika kemungkinan *state* tersebut belum pernah dikunjungi, masukkan ke dalam Priority Queue, set, dan map.
6. Ulangi setiap langkah dari langkah ketiga hingga ditemukan *state* yang mencapai kondisi tujuan atau semua kemungkinan sudah dikunjungi.

2.2 Algoritma Greedy Best-First Search (GBFS)

Algoritma Greedy Best-First Search merupakan algoritma pencarian rute *informed* yang menggunakan suatu fungsi heuristik untuk mengestimasikan jarak suatu *node* terhadap *node* tujuan. Dengan prinsip algoritma Greedy, algoritma ini akan memeriksa dan menjelajahi *node* dengan nilai fungsi heuristik yang tampak paling dekat atau paling menjanjikan terhadap *node* tujuan. Kelemahan dari algoritma ini, sama seperti algoritma Greedy, adalah tidak terjaminnya ditemukan solusi optimal, yakni jumlah gerakan yang paling sedikit. Berikut adalah langkah-langkah penerapan algoritma GBFS dalam menyelesaikan puzzle ini.

1. Masukkan *state* awal ke dalam Priority Queue.
2. Siapkan set berisi *state* yang sudah pernah dikunjungi dan map untuk mencatat jalur antara *state*.
3. Ambil *state* dengan nilai heuristik terkecil pada Priority Queue. Nilai heuristik ditentukan oleh pengguna dari dua pilihan, yaitu berdasarkan jarak kendaraan utama dari pintu keluar atau berdasarkan jumlah kendaraan yang memblokir kendaraan utama dari pintu keluar.
4. Periksa apakah *state* sudah memenuhi kondisi tujuan. Jika sudah, maka pencarian selesai.
5. Jika belum, jabarkan setiap kemungkinan *state* berikutnya berdasarkan *state* saat ini. Jika kemungkinan *state* tersebut belum pernah dikunjungi, masukkan ke dalam Priority Queue, set, dan map.
6. Ulangi setiap langkah dari langkah ketiga hingga ditemukan *state* yang mencapai kondisi tujuan atau semua kemungkinan sudah dikunjungi.

2.3 Algoritma A*

Algoritma A* merupakan algoritma pencarian rute *informed* yang, dapat dikatakan, menggabungkan prinsip kerja algoritma UCS dan algoritma GBFS. Cara kerja algoritma ini menggunakan fungsi evaluasi f(n) yang merupakan penjumlahan dari fungsi g(n) dan fungsi h(n). Fungsi g(n) menyatakan biaya aktual dari *node* awal, sedangkan fungsi h(n) menyatakan estimasi biaya menuju *node* tujuan. Berikut adalah penjabaran langkah-langkah penerapan algoritma A* dalam menyelesaikan puzzle ini.

1. Masukkan *state* awal ke dalam Priority Queue.
2. Siapkan set berisi *state* yang sudah pernah dikunjungi dan map untuk mencatat jalur antara *state*.
3. Ambil *state* dengan nilai fungsi evaluasi terkecil pada Priority Queue. Fungsi evaluasi disini merupakan penjumlahan dari kedalaman *node* (jarak dari *node* tujuan) dengan nilai heuristik (estimasi biaya menuju *node* tujuan).
4. Periksa apakah *state* sudah memenuhi kondisi tujuan. Jika sudah, maka pencarian selesai.

5. Jika belum, jabarkan setiap kemungkinan *state* berikutnya berdasarkan *state* saat ini. Jika kemungkinan *state* tersebut belum pernah dikunjungi, masukkan ke dalam Priority Queue, set, dan map.
6. Ulangi setiap langkah dari langkah ketiga hingga ditemukan *state* yang mencapai kondisi tujuan atau semua kemungkinan sudah dikunjungi.

2.4 Algoritma Iterative Deepening A* (IDA*)

Algoritma Iterative Deepening A* merupakan algoritma pencarian rute *informed* yang menggabungkan prinsip kerja algoritma A* dengan prinsip Iterative Deepening. Cara kerja algoritma ini adalah melakukan pencarian secara mendalam dengan kedalaman yang dibatasi oleh fungsi evaluasi f(n). Berikut adalah penjabaran langkah-langkah penerapan algoritma Iterative Deepening A* dalam menyelesaikan puzzle ini.

1. Hitung nilai heuristik awal dari state awal dan tetapkan sebagai nilai *threshold* awal.
2. Lakukan pencarian dengan DFS mulai dari state awal, tetapi hanya telusuri node yang memiliki nilai evaluasi f(n) tidak melebihi *threshold* saat ini.
3. Setiap kali mencapai node dengan f(n) melebihi *threshold*, catat nilai f(n) tersebut sebagai kandidat *threshold* berikutnya.
4. Jika ditemukan state yang memenuhi kondisi tujuan, maka pencarian dihentikan dan solusi dikembalikan.
5. Jika tidak ditemukan solusi dan tidak ada lagi node yang memenuhi syarat $f(n) \leq$ *threshold*, maka pencarian diulang dengan *threshold* baru yaitu nilai f(n) terkecil yang melebihi ambang sebelumnya.
6. Proses ini terus diulang hingga ditemukan solusi atau semua kemungkinan eksplorasi telah habis.

BAB 3: Analisis Algoritma *Pathfinding*

Dalam penyelesaian puzzle Rush Hour, algoritma Uniform Cost Search (UCS) digunakan sebagai baseline dari pendekatan uninformed search. Uninformed search berarti strategi pencarian tidak memiliki pengetahuan tambahan mengenai posisi tujuan selain struktur dasar graf atau ruang keadaan. UCS bekerja dengan memilih node berdasarkan biaya g(n), yaitu total biaya yang telah dikeluarkan dari titik awal hingga titik tersebut. Karena dalam puzzle Rush Hour semua gerakan memiliki biaya yang seragam (yaitu satu langkah per geser), maka UCS menjadi identik dengan Breadth-First Search (BFS). Akibatnya, urutan eksplorasi node dan jalur solusi yang dihasilkan oleh UCS dan BFS akan sama saja.

Berbeda dengan UCS, algoritma pencarian informed seperti Greedy Best First Search dan A* menggunakan fungsi heuristik untuk membantu memperkirakan kedekatan sebuah node terhadap tujuan. Dalam konteks ini, kita memperkenalkan beberapa fungsi penting: g(n) menyatakan biaya riil dari node awal ke node n (jumlah langkah sejauh ini), h(n) adalah estimasi biaya dari node n ke node tujuan (heuristik), dan f(n) adalah total estimasi biaya dari awal hingga tujuan melalui n. Berdasarkan kombinasi ini, terdapat tiga pendekatan: UCS menggunakan f(n) = g(n), Greedy Best First Search menggunakan f(n) = h(n), dan A* menggunakan f(n) = g(n) + h(n).

Dalam penerapan A*, heuristik h(n) yang digunakan sangat berperan penting dalam menentukan efisiensi dan optimalitas. Sebuah heuristik disebut *admissible* jika untuk setiap node n, nilai h(n) tidak pernah melebihi nilai sebenarnya h*(n)—yakni biaya minimum sebenarnya untuk mencapai tujuan dari node n. Artinya, heuristik yang admissible bersifat optimistik, tidak pernah melebih-lebihkan estimasi biaya ke tujuan. Dalam tugas ini, heuristik yang digunakan adalah jarak antara ujung primary piece dengan pintu keluar (*DistanceToExit*) dan jumlah kendaraan yang secara langsung menghalangi jalur tersebut (*BlockingPieces*). Karena nilai ini tidak pernah melebihi jumlah langkah riil yang dibutuhkan untuk mencapai pintu keluar, maka heuristik tersebut dikategorikan *admissible*, dan menjamin bahwa A* akan menemukan solusi yang optimal.

Secara teoritis, A* lebih efisien dibandingkan UCS dalam menyelesaikan Rush Hour karena A* diarahkan oleh heuristik untuk memprioritaskan eksplorasi ke arah goal. UCS, sebagai

algoritma *uninformed*, cenderung mengeksplorasi node secara menyeluruh tanpa arah, sehingga membutuhkan lebih banyak eksplorasi dan waktu dalam kasus kompleks. Dengan adanya informasi heuristik, A* dapat memangkas banyak cabang yang tidak relevan dan fokus pada jalur yang lebih menjanjikan.

Sementara itu, algoritma Greedy Best First Search hanya mempertimbangkan nilai h(n) tanpa memperhatikan biaya g(n). Meskipun Greedy BFS sering kali sangat cepat karena langsung menuju ke arah goal, algoritma ini tidak menjamin solusi optimal. Hal ini terjadi karena ia bisa saja memilih jalur yang tampak paling dekat ke tujuan tetapi justru memiliki total langkah lebih panjang. Maka, meskipun efisien secara waktu dalam banyak kasus, Greedy BFS tidak menjamin solusi yang optimal.

BAB 4: Implementasi

4.1 Main.java

```
package com.tuciltiga;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        FXMLLoader loader = new
FXMLLoader(getClass().getResource("/view/mainView.fxml"));
        Parent root = loader.load();

        Scene scene = new Scene(root);
        stage.setTitle("Tugas Kecil 3 Strategi Algoritma");
        stage.setScene(scene);
        stage.setMinWidth(600);
        stage.setMinHeight(500);
        stage.show();
    }

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

4.2 Piece.java

```
package game;

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

/* Kelas Piece yang merepresentasikan sebuah piece */
public class Piece {

    // Atribut
    private char name;
```

```
    private List<Coordinate> coordinates; // list koordinat dari piece
    private boolean isPrimary; // flag untuk primary piece
    private boolean isHorizontal; // flag untuk horizontal piece

    // Methods
    public Piece(char name, List<Coordinate> coordinates, boolean isPrimary,
boolean isHorizontal) {
        this.name = name;
        this.coordinates = coordinates;
        this.isPrimary = isPrimary;
        this.isHorizontal = isHorizontal;
    }

    public Piece(Piece other) {
        this.name = other.name;
        this.coordinates = new ArrayList<>();
        for (Coordinate coord : other.coordinates) {
            this.coordinates.add(new Coordinate(coord));
        }
        this.isPrimary = other.isPrimary;
        this.isHorizontal = other.isHorizontal;
    }

    // Getter
    public char getName() {
        return name;
    }
    public List<Coordinate> getCoordinates() {
        return coordinates;
    }
    public boolean isPrimary() {
        return isPrimary;
    }
    public boolean isHorizontal() {
        return isHorizontal;
    }

    // Setter
    public void setName(char name) {
        this.name = name;
    }
    public void setCoordinates(List<Coordinate> coordinates) {
        this.coordinates = coordinates;
    }
    public void setPrimary(boolean isPrimary) {
        this.isPrimary = isPrimary;
    }
```

```
    public void setHorizontal(boolean isHorizontal) {
        this.isHorizontal = isHorizontal;
    }

    public int getWidth() {
        int minCol = Integer.MAX_VALUE;
        int maxCol = Integer.MIN_VALUE;
        for (Coordinate coordinate : coordinates) {
            if (coordinate.getCol() < minCol) {
                minCol = coordinate.getCol();
            }
            if (coordinate.getCol() > maxCol) {
                maxCol = coordinate.getCol();
            }
        }
        return maxCol - minCol + 1;
    }

    public int getHeight() {
        int minRow = Integer.MAX_VALUE;
        int maxRow = Integer.MIN_VALUE;
        for (Coordinate coordinate : coordinates) {
            if (coordinate.getRow() < minRow) {
                minRow = coordinate.getRow();
            }
            if (coordinate.getRow() > maxRow) {
                maxRow = coordinate.getRow();
            }
        }
        return maxRow - minRow + 1;
    }

    // Method untuk mengubah posisi piece
    public void shift(int rowShift, int colShift) {
        for (Coordinate coordinate : coordinates) {
            coordinate.shift(rowShift, colShift);
        }
    }
    public void move(Coordinate shift) {
        for (Coordinate coordinate : coordinates) {
            coordinate.moveTo(shift);
        }
    }
    public void moveDirection(Move.Direction direction) {
        for (Coordinate coordinate : coordinates) {
            switch (direction) {
                case UP -> coordinate.shiftRow(-1);
```

```
                case DOWN -> coordinate.shiftRow(1);
                case LEFT -> coordinate.shiftCol(-1);
                case RIGHT -> coordinate.shiftCol(1);
            }
        }
    }

    public void removeCoordinateAtCoordinate(Coordinate coordinate) {
        coordinates.removeIf(coord -> coord.equals(coordinate));
    }

    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Piece ").append(name).append(": ");
        for (Coordinate coordinate : coordinates) {
            sb.append("(").append(coordinate.getRow()).append(",
").append(coordinate.getCol()).append(") ");
        }
        return sb.toString();
    }
}
```

4.3 Coordinate.java

```
package game;

/* Kelas Coordinate yang merepresentasikan suatu koordinat pada papan */
public class Coordinate {
    private int row; // posisi baris
    private int col; // posisi kolom

    // Constructor
    public Coordinate(int row, int col) {
        this.row = row;
        this.col = col;
    }
    public Coordinate(Coordinate other) {
        this.row = other.row;
        this.col = other.col;
    }

    // Getter
    public int getRow() {
```

```
        return row;
    }
    public int getCol() {
        return col;
    }

    // Setter
    public void setRow(int row) {
        this.row = row;
    }
    public void setCol(int col) {
        this.col = col;
    }

    // Method untuk mengubah posisi koordinat
    public void shift(int rowShift, int colShift) {
        this.row += rowShift;
        this.col += colShift;
    }
    public void moveTo(Coordinate shift) {
        this.row = shift.getRow();
        this.col = shift.getCol();
    }
    public void shiftRow(int rowShift) {
        this.row += rowShift;
    }
    public void shiftCol(int colShift) {
        this.col += colShift;
    }

    // override method equals
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Coordinate that = (Coordinate) obj;
        return row == that.row && col == that.col;
    }

    @Override
    public int hashCode() {
        int result = Integer.hashCode(row);
        result = 31 * result + Integer.hashCode(col);
        return result;
    }
}
```

4.4 Move.java

```
package game;

public class Move {
    private final char pieceName;
    private final Direction direction;
    BoardState board;

    public Move(char pieceName, Direction direction, BoardState board) {
        this.pieceName = pieceName;
        this.direction = direction;
        this.board = board;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Move move = (Move) obj;
        return pieceName == move.pieceName && direction == move.direction &&
board.equals(move.board);
    }

    @Override
    public int hashCode() {
        int result = Character.hashCode(pieceName);
        result = 31 * result + (direction != null ? direction.hashCode() :
0);
        result = 31 * result + (board != null ? board.hashCode() : 0);
        return result;
    }

    public char getPieceName() {
        return pieceName;
    }

    public Direction getDirection() {
        return direction;
    }

    public BoardState getBoard() {
        return board;
    }

    public enum Direction {
        UP, DOWN, LEFT, RIGHT
```

```
    }
}
```

4.5 Algorithm.java

```
package game;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;

public abstract class Algorithm {
    public static List<BoardState> reconstructPath(Map<BoardState,
BoardState> parentMap, BoardState currentState) {
        List<BoardState> path = new ArrayList<>();
        while (currentState != null) {
            path.add(currentState);
            currentState = parentMap.get(currentState);
        }
        Collections.reverse(path);
        return path;
    }
}
```

4.6 AStar.java

```
package game;

import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

/* Kelas untuk solver menggunakan algoritma A* */
public class AStar extends Algorithm {
    // Fungsi solver utama, mengembalikan array of Object berisi jalur dan
jumlah gerakan
    public static Object[] solve(BoardState initialState) {
```

```
        BoardState currentState = initialState; // state awal

        // PriorityQueue untuk menyimpan state berdasarkan nilai heuristik
        PriorityQueue<BoardState> queue = new
PriorityQueue<>(Comparator.comparingInt(state -> state.getValue() +
state.getDepth()));
        queue.add(currentState);

        // Set berisi state yang sudah dikunjungi
        Set<BoardState> visited = new HashSet<>();
        visited.add(currentState); // Setiap state yang dijelajah dimasukkan

        // Map untuk menyimpan parent dari setiap state
        Map<BoardState, BoardState> parentMap = new HashMap<>();

        int countNode = 0; // menghitung jumlah gerakan yang dieksplorasi

        // eksplorasi semua kemungkinan
        while (!queue.isEmpty()) {
            currentState = queue.poll(); // cek state terdepan
            countNode++; // increment jumlah node

            if (currentState.isGoal()) {
                return new Object[]{reconstructPath(parentMap,
currentState), countNode}; // jika sudah mencapai tujuan, kembalikan jalur
            }

            // daftar semua langkah
            List<BoardState> possibleMoves =
currentState.getPossibleMoves();

            // iterasi setiap langkah dan tambahkan jika belum dikunjungi
            for (BoardState nextState : possibleMoves) {
                if (!visited.contains(nextState)) {
                    visited.add(nextState);
                    parentMap.put(nextState, currentState);
                    queue.add(nextState);
                }
            }
        }

        return null; // tidak ada solusi
    }
}
```

4.7 GBFS.java

```
package game;

import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

/* Kelas untuk solver menggunakan algoritma Greedy Best First Search */
public class GBFS extends Algorithm {
    // Fungsi solver utama, mengembalikan array of Object berisi jalur dan
jumlah gerakan
    public static Object[] solve(BoardState initialState) {
        BoardState currentState = initialState; // state awal

        // PriorityQueue untuk menyimpan state berdasarkan nilai heuristik
        PriorityQueue<BoardState> queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::getValue));
        queue.add(currentState);

        // Set berisi state yang sudah dikunjungi
        Set<BoardState> visited = new HashSet<>();
        visited.add(currentState); // Setiap state yang dijelajah dimasukkan

        // Map untuk menyimpan parent dari setiap state
        Map<BoardState, BoardState> parentMap = new HashMap<>();

        int countNode = 0; // menghitung jumlah gerakan yang dieksplorasi

        // eksplorasi semua kemungkinan
        while (!queue.isEmpty()) {
            currentState = queue.poll(); // cek state terdepan
            countNode++; // increment jumlah node

            if (currentState.isGoal()) {
                return new Object[]{reconstructPath(parentMap,
currentState), countNode}; // jika sudah mencapai tujuan, kembalikan jalur
            }

            // daftar semua langkah
            List<BoardState> possibleMoves =
currentState.getPossibleMoves();
```

```
            // iterasi setiap langkah dan tambahkan jika belum dikunjungi
            for (BoardState nextState : possibleMoves) {
                if (!visited.contains(nextState)) {
                    queue.add(nextState);
                    visited.add(nextState);
                    parentMap.put(nextState, currentState);
                }
            }
        }

        return null; // tidak ada solusi
    }
}
```

4.8 UCS.java

```
package game;

import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

/* Kelas untuk solver menggunakan algoritma UCS */
public class UCS extends Algorithm {
    // Fungsi solver utama, mengembalikan array of Object berisi jalur dan
jumlah gerakan
    public static Object[] solve(BoardState initialState) {
        BoardState currentState = initialState; // state awal

        // PriorityQueue untuk menyimpan state berdasarkan kedalaman
        PriorityQueue<BoardState> queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::getDepth));
        queue.add(currentState);

        // Set berisi state yang sudah dikunjungi
        Set<BoardState> visited = new HashSet<>();
        visited.add(currentState); // Setiap state yang dijelajah dimasukkan

        // Map untuk menyimpan parent dari setiap state
        Map<BoardState, BoardState> parentMap = new HashMap<>();
```

```
        int countNode = 0; // menghitung jumlah gerakan yang dieksplorasi

        // eksplorasi semua kemungkinan
        while (!queue.isEmpty()) {
            currentState = queue.poll(); // cek state terdepan
            countNode++; // increment jumlah node

            System.out.println("Node ke-" + countNode); // tampilkan node
yang sedang dieksplorasi
            System.out.println(currentState.getDepth()); // tampilkan state
yang sedang dieksplorasi

            if (currentState.isGoal()) {
                return new Object[]{reconstructPath(parentMap,
currentState), countNode}; // jika sudah mencapai tujuan, kembalikan jalur
            }

            // daftar semua langkah
            List<BoardState> possibleMoves =
currentState.getPossibleMoves();

            // iterasi setiap langkah dan tambahkan jika belum dikunjungi
            for (BoardState nextState : possibleMoves) {
                if (!visited.contains(nextState)) {
                    visited.add(nextState);
                    parentMap.put(nextState, currentState);
                    queue.add(nextState);
                }
            }
        }

        return null; // tidak ada solusi
    }
}
```

4.9 Heuristic.java

```
package game;

public interface Heuristic {
    /**
     * Menghitung nilai heuristik untuk state tertentu.
```

```
     *
     * @param state State yang akan dihitung nilai heuristiknya.
     * @return Nilai heuristik untuk state tersebut.
     */
    int calcValue(BoardState state);
}
```

4.10 BoardState.java

```
package game;

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

import utils.Utils;

/* Kelas yang merepresentasikan state papan pada permainan */
public class BoardState {
    private int row;
    private int col;
    private char[][] board;
    private Coordinate exitCoordinate; // koordinat keluar
    private List<Piece> pieces; // list piece
    private Piece primaryPiece; // piece utama
    private Heuristic heuristic; // heuristik yang digunakan
    private int value; // nilai heuristik
    private Move lastMove; // langkah terakhir yang diambil
    private int depth; // depth node

    // Konstruktor
    public BoardState() {
        this.row = 0;
        this.col = 0;
        this.board = null;
        this.exitCoordinate = null;
        this.pieces = null;
        this.primaryPiece = null;
        this.heuristic = null;
        this.value = 0;
        this.lastMove = null;
        this.depth = 0;
    }
    public BoardState(int row, int col, char[][] board, List<Piece> pieces,
Coordinate exitCoordinate, Piece primaryPiece, Heuristic heuristic, Move
```

```
lastMove, int depth) {
        this.row = row;
        this.col = col;
        this.board = board;
        this.exitCoordinate = exitCoordinate;
        this.pieces = pieces;
        this.primaryPiece = primaryPiece;
        this.heuristic = heuristic;
        BoardState self = this;
        this.value = 0;
        if(this.heuristic != null) this.value = heuristic.calcValue(self);
        this.lastMove = lastMove;
        this.depth = depth;
    }

    // cek apakah state sudah mencapai tujuan
    public boolean isGoal() {
        return Utils.isPieceOnCoordinate(primaryPiece, exitCoordinate);
    }

    // membuat state baru dari state yang ada dengan menggerakkan sebuah
piece
    public BoardState movePiece(Piece pieceToMove, Move.Direction direction)
{ // Renamed parameter for clarity
        if (!isValidMove(pieceToMove, direction)) {
            return null; // jika tidak valid, kembalikan null
        }

        // Create a deep copy of the pieces list
        List<Piece> newPiecesList = new ArrayList<>();
        Piece movedPieceInNewList = null;
        for (Piece p : this.pieces) {
            Piece copiedPiece = new Piece(p);
            newPiecesList.add(copiedPiece);
            if (p.getName() == pieceToMove.getName()) {
                movedPieceInNewList = copiedPiece;
            }
        }

        if (movedPieceInNewList == null) {
            return null;
        }
        movedPieceInNewList.moveDirection(direction);

        Piece newPrimaryPiece = (this.primaryPiece.getName() ==
movedPieceInNewList.getName()) ? movedPieceInNewList : null;
        if (newPrimaryPiece == null) {
```

```
            for (Piece p : newPiecesList) {
                if (p.getName() == this.primaryPiece.getName()) {
                    newPrimaryPiece = p;
                    break;
                }
            }
        }

        char[][] newBoardArray = buildBoard(newPiecesList);

        return new BoardState(this.row, this.col, newBoardArray,
newPiecesList, this.exitCoordinate, newPrimaryPiece, this.heuristic, new
Move(movedPieceInNewList.getName(), direction, this), this.depth + 1);
    }

    // membuat board dari list of pieces
    public char[][] buildBoard(List<Piece> pieces) {
        char[][] newBoard = new char[board.length][board[0].length];
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (board[i][j] == 'K') {
                    newBoard[i][j] = 'K';
                } else if (!Character.isLetter(board[i][j]) && board[i][j]
!= '.') {
                    newBoard[i][j] = ' ';
                }
            }
        }

        // exit coordinate di atas
        if (exitCoordinate.getRow() == 0 && exitCoordinate.getCol() <
board[0].length-1) {
            for (int i = 1; i < row+1; i++) {
                for (int j = 0; j < col; j++) {
                    newBoard[i][j] = '.';
                }
            }
        } else if (exitCoordinate.getCol() == 0 && exitCoordinate.getRow() <
board.length-1) { // exit coord di kiri
            for (int i = 0; i < row; i++) {
                for (int j = 1; j < col+1; j++) {
                    newBoard[i][j] = '.';
                }
            }
        } else {
            for (int i = 0; i < row; i++) {
                for (int j = 0; j < col; j++) {
```

```
                    newBoard[i][j] = '.';
                }
            }
        }

        for (Piece piece : pieces) {
            for (Coordinate coordinate : piece.getCoordinates()) {
                newBoard[coordinate.getRow()][coordinate.getCol()] =
piece.getName();
            }
        }
        return newBoard;
    }

    public static void printBoard(char[][] board) {
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                System.out.print(board[i][j]);
            }
            System.out.println();
        }
    }

    // cek apakah piece bisa bergerak ke arah yang diinginkan
    public boolean isValidMove(Piece piece, Move.Direction direction) {
        int trow = board.length;
        int tcol = board[0].length;

        for (Coordinate coordinate : piece.getCoordinates()) {
            int nextRow = coordinate.getRow();
            int nextCol = coordinate.getCol();

            switch (direction) {
                case UP:    nextRow--; break;
                case DOWN:  nextRow++; break;
                case LEFT:  nextCol--; break;
                case RIGHT: nextCol++; break;
                default: return false;
            }

            if (nextRow < 0 || nextRow >= trow || nextCol < 0 || nextCol >=
tcol) {
                return false;
            }

            char targetCell = board[nextRow][nextCol];
```

```
            if (targetCell == piece.getName()) {
                continue;
            }

            if (piece.isPrimary()) {
                if (targetCell != '.' && targetCell != 'K') {
                    return false;
                }
            } else {
                if (targetCell != '.') {
                    return false;
                }
            }
        }
        return true;
    }

    private boolean isBetween(int value, int bound1, int bound2) {
        return (bound1 < value && value < bound2) || (bound2 < value &&
value < bound1);
    }

    public boolean isPieceBlocking(Piece piece) {
        for (Coordinate coordinate : piece.getCoordinates()) {
            if(this.primaryPiece.isHorizontal()){
                // berarti blockingnya kan vertikal
                if(coordinate.getRow() == exitCoordinate.getRow()){
                    for(Coordinate coordinate2 :
this.primaryPiece.getCoordinates()){
                        if(isBetween(coordinate.getCol(),
coordinate2.getCol(), exitCoordinate.getCol())){
                            return true;
                        }
                    }
                }
            }
            else{
                // berarti blockingnya kan horizontal
                if(coordinate.getCol() == exitCoordinate.getCol()){
                    for(Coordinate coordinate2 :
this.primaryPiece.getCoordinates()){
                        if(isBetween(coordinate.getRow(),
coordinate2.getRow(), exitCoordinate.getRow())){
                            return true;
                        }
                    }
                }
```

```
            }
        }
        return false;
    }

    public List<BoardState> getPossibleMoves() {
        List<BoardState> possibleMoves = new ArrayList<>();
        for (Piece piece : this.pieces) { // Iterate through a stable list
of pieces
            Move.Direction[] directionsToTry;
            if (piece.isHorizontal()) {
                directionsToTry = new Move.Direction[]{Move.Direction.LEFT,
Move.Direction.RIGHT};
            } else {
                directionsToTry = new Move.Direction[]{Move.Direction.UP,
Move.Direction.DOWN};
            }

            for (Move.Direction direction : directionsToTry) {
                BoardState nextState = movePiece(piece, direction);
                if (nextState != null) {
                    possibleMoves.add(nextState);
                }
            }
        }

        return possibleMoves;
    }

    public Move.Direction getDirectionToExit() {
        if (this.primaryPiece.isHorizontal()) {
            // antara kiri atau kanan
            if (this.exitCoordinate.getCol() == 0) { // di kiri
                return Move.Direction.LEFT;
            } else {
                return Move.Direction.RIGHT;
            }
        } else { // atas atau bawah
            if (this.exitCoordinate.getRow() == 0) { // di atas
                return Move.Direction.UP;
            } else {
                return Move.Direction.DOWN;
            }
        }
    }

    @Override
```

```java
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof BoardState)) return false;
        BoardState other = (BoardState) obj;
        if (this.row != other.row || this.col != other.col) return false;
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (this.board[i][j] != other.board[i][j]) return false;
            }
        }
        return true;
    }

    @Override
    public int hashCode() {
        int result = 1;
        result = 31 * result + row;
        result = 31 * result + col;
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                result = 31 * result + board[i][j];
            }
        }
        return result;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                sb.append(board[i][j]);
            }
            sb.append("\n");
        }
        return sb.toString();
    }

    // Getter
    public int getRow() {
        return row;
    }
    public int getCol() {
        return col;
    }
    public char[][] getBoard() {
        return board;
```

```
    }
    public Coordinate getExitCoordinate() {
        return exitCoordinate;
    }
    public List<Piece> getPieces() {
        return pieces;
    }
    public Piece getPrimaryPiece() {
        return primaryPiece;
    }
    public Heuristic getHeuristic() {
        return heuristic;
    }
    public int getValue() {
        return value;
    }
    public Move getLastMove() {
        return lastMove;
    }
    public int getDepth() {
        return depth;
    }
    // Setter
    public void setRow(int row) {
        this.row = row;
    }
    public void setCol(int col) {
        this.col = col;
    }
    public void setBoard(char[][] board) {
        this.board = board;
    }
    public void setExitCoordinate(Coordinate exitCoordinate) {
        this.exitCoordinate = exitCoordinate;
    }
    public void setPieces(List<Piece> pieces) {
        this.pieces = pieces;
    }
    public void setPrimaryPiece(Piece primaryPiece) {
        this.primaryPiece = primaryPiece;
    }
    public void setValue(int value) {
        this.value = value;
    }
    public void setLastMove(Move lastMove) {
        this.lastMove = lastMove;
    }
```

```
    public void setDepth(int depth) {
        this.depth = depth;
    }
    public void setHeuristic(Heuristic heuristic) {
        this.heuristic = heuristic;
        this.value = heuristic.calcValue(this);
    }

}
```

4.11 IOHandler.java

```
package utils;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Scanner;
import java.util.Set;

import game.BoardState;
import game.Coordinate;
import game.Move;
import game.Piece;

/* Kelas statik untuk handle input dan output */
public class IOHandler {
    private static final Scanner scanner = new Scanner(System.in);

    // TODO: Validasi input

    // baca file path
    public static File readInputFile() {
        System.out.println("Masukkan nama file input (pastikan berada di
dalam folder test/problem): ");
        String fileName = scanner.nextLine();
        File file = new File("test/problem/" + fileName);
```

```
        while (!file.exists()) {
            System.out.println("File tidak ditemukan. Silakan coba lagi: ");
            fileName = scanner.nextLine();
            file = new File("test/problem/" + fileName);
        }

        return file;
    }

    // konversi input file ke dalam papan
    public static BoardState convertInput(File file) {
        try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
            String line;

            // membaca baris pertama untuk mendapatkan ukuran papan
            line = reader.readLine();
            String[] boardConfig = line.split(" ");
            int row = Integer.parseInt(boardConfig[0]);
            int col = Integer.parseInt(boardConfig[1]);

            // membaca jumlah piece (kecuali piece keluar)
            line = reader.readLine();
            String pieceNum = line.trim();
            int numPieces = Integer.parseInt(pieceNum);

            line = reader.readLine();

            // membaca konfigurasi papan
            char[][] tempBoard = new char[row+1][col+1];
            int tempRow = 0;
            while (line != null) {
                String[] linePieces = line.split("");
                if (tempRow >= tempBoard.length) {
                    throw new IOException("baris kelebihan");
                }
                for (int i = 0; i < linePieces.length; i++) {
                    if (i >= linePieces.length) {
                        throw new IOException("kolom kelebihan");
                    }
                    tempBoard[tempRow][i] = linePieces[i].charAt(0);
                }
                tempRow++;
                line = reader.readLine();
            }

            // System.out.println("Getting exit coordinates");
```

```
            // get exit coordinate
            Coordinate exitCoordinate = getExitCoordinate(tempBoard);
            if (exitCoordinate == null) {
                throw new IOException("Koordinat keluar tidak ditemukan");
            }

            // System.out.println("Getting exit direction");
            // get exit side
            Move.Direction exitSide = getExitSide(tempBoard);

            // System.out.println("Trimming board");
            // trim board
            char[][] newBoard = trimBoard(tempBoard, exitSide);
            if (newBoard == null) {
                throw new IOException("Gagal memotong papan");
            }

            // System.out.println("Converting pieces");
            // convert pieces
            List<Piece> pieces = convertPieces(newBoard, numPieces);

            // System.out.println("Finished converting pieces");

            Piece primaryPiece = getPrimaryPiece(pieces);

            // for (Piece piece : pieces) {
            //     System.out.println(piece);
            // }

            // for (int i = 0; i < newBoard.length; i++) {
            //     for (int j = 0; j < newBoard[i].length; j++) {
            //         System.out.print(newBoard[i][j]);
            //     }
            //     System.out.println();
            // }
            // buat BoardState dengan papan lengkap dari awal
            return new BoardState(row, col, newBoard, pieces,
exitCoordinate, primaryPiece, null, null, 0);
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
            return null;
        }
    }

    // dapatkan list pieces
    public static List<Piece> convertPieces(char[][] board, int numPieces) {
        List<Piece> pieces = new ArrayList<>();
```

```
        Set<Character> processedPieces = new HashSet<>();
        int targetCount = numPieces + 1;  // +1 for primary piece
        int currentCount = 0;

        for (int i = 0; i < board.length && currentCount < targetCount; i++)
{
            for (int j = 0; j < board[i].length && currentCount <
targetCount; j++) {
                char c = board[i][j];

                if (!Character.isLetter(c) || c == 'K' ||
processedPieces.contains(c)) {
                    continue;
                }

                List<Coordinate> coordinates = new ArrayList<>();
                for (int k = i; k < board.length; k++) {
                    for (int l = (k == i ? j : 0); l < board[k].length; l++)
{
                        if (board[k][l] == c) {
                            coordinates.add(new Coordinate(k, l));
                        }
                    }
                }

                if (coordinates.isEmpty() || coordinates.size() < 2 ||
coordinates.size() > 3) {
                    throw new IllegalArgumentException("Invalid piece
coordinates for piece: " + c);
                }

                boolean isPrimary = c == 'P';
                boolean isHorizontal = Utils.isPieceHorizontal(coordinates);
                pieces.add(new Piece(c, coordinates, isPrimary,
isHorizontal));

                processedPieces.add(c);
                currentCount++;
            }
        }

        if (currentCount != targetCount) {
            throw new IllegalArgumentException("Invalid number of pieces
found. Expected: " + targetCount + ", Found: " + currentCount);
        }

        return pieces;
```

```
    }

    public static Coordinate getExitCoordinate(char[][] fileBoard) {
        for (int i = 0; i < fileBoard.length; i++) {
            for (int j = 0; j < fileBoard[i].length; j++) {
                if (fileBoard[i][j] == 'K') {
                    return new Coordinate(i, j);
                }
            }
        }
        return null;
    }

    // dapatkan sisi keluar dari papan
    public static Move.Direction getExitSide(char[][] fileBoard) {
        for (int i = 0; i < fileBoard.length; i++) {
            for (int j = 0; j < fileBoard[i].length; j++) {
                if (fileBoard[i][j] == 'K') {
                    if (i == 0 && j < fileBoard[i].length - 1) {
                        System.out.println("UP");
                        return Move.Direction.UP;
                    } else if (i == fileBoard.length - 1 && j <
fileBoard[i].length - 1) {
                        System.out.println("DOWN");
                        return Move.Direction.DOWN;
                    } else if (j == 0 && i < fileBoard.length - 1) {
                        System.out.println("LEFT");
                        return Move.Direction.LEFT;
                    } else if (j == fileBoard[i].length - 1 && i <
fileBoard.length - 1) {
                        System.out.println("RIGHT");
                        return Move.Direction.RIGHT;
                    }
                }
            }
        }
        return null;
    }

    // trim papan sesuai dengan sisi keluar
    public static char[][] trimBoard(char[][] fileBoard, Move.Direction
exitSide) {
        switch (exitSide) {
            case UP -> {
                char[][] newBoard = new
char[fileBoard.length][fileBoard[0].length-1];
                // trim kolom paling kanan
```

```
                for (int i = 0; i < fileBoard.length; i++) {
                    System.arraycopy(fileBoard[i], 0, newBoard[i], 0,
fileBoard[i].length - 1);
                }
                return newBoard;
            }
            case DOWN -> {
                char[][] newBoard = new
char[fileBoard.length][fileBoard[0].length-1];
                // trim kolom paling kanan
                for (int i = 0; i < fileBoard.length; i++) {
                    System.arraycopy(fileBoard[i], 0, newBoard[i], 0,
fileBoard[i].length - 1);
                }
                return newBoard;
            }
            case LEFT -> {
                char[][] newBoard = new
char[fileBoard.length-1][fileBoard[0].length];
                // trim baris paling bawah
                for (int i = 0; i < fileBoard.length - 1; i++) {
                    System.arraycopy(fileBoard[i], 0, newBoard[i], 0,
fileBoard[i].length);
                }
                return newBoard;
            }
            case RIGHT -> {
                char[][] newBoard = new
char[fileBoard.length-1][fileBoard[0].length];
                // trim baris paling bawah
                for (int i = 0; i < fileBoard.length - 1; i++) {
                    System.arraycopy(fileBoard[i], 0, newBoard[i], 0,
fileBoard[i].length);
                }
                return newBoard;
            }
        }
        return null;
    }

    // input untuk algoritma
    public static int inputAlgorithm() {
        System.out.println("Pilih algoritma yang ingin digunakan: ");
        System.out.println("1. Greedy Best First Search");
        System.out.println("2. Uniformed Cost Search");
        System.out.println("3. A*");
        System.out.print("Pilihan Anda: ");
```

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

        while (choice < 1 || choice > 3) {
            System.out.print("Pilihan tidak valid. Silakan coba lagi: ");
            choice = scanner.nextInt();
        }

        return choice;
    }

    public static Piece getPieceByName(char name, List<Piece> pieces) {
        for (Piece piece : pieces) {
            if (piece.getName() == name) {
                return piece;
            }
        }
        return null;
    }

    public static Piece getPrimaryPiece(List<Piece> pieces) {
        for (Piece piece : pieces) {
            if (piece.isPrimary()) {
                return piece;
            }
        }
        return null;
    }

    public static void findPrimaryPiece(List<Piece> pieces) {
        for (Piece piece : pieces) {
            if (piece.isPrimary()) {
                System.out.println("Primary Piece: " + piece.getName());
                return;
            }
        }
        System.out.println("Primary Piece tidak ditemukan.");
    }

    public static void outputToFile(String fileName, List<BoardState> path)
{
        // Construct the full path to the directory
        File outputDir = new File("test/result"); // Relative to project
root is often more reliable

        if (!outputDir.exists()) {
            boolean created = outputDir.mkdirs();
            if (created) {
```

```
                System.out.println("Created output directory: " +
outputDir.getAbsolutePath());
            } else {
                System.out.println("Failed to create output directory: " +
outputDir.getAbsolutePath());            }
        }

        File filePath = new File(outputDir, fileName); // Use the directory
and filename

        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(filePath))) {
            writer.write("Papan Awal");
            writer.newLine();
            BoardState initialState = path.get(0);
            for (int i = 0; i < initialState.getBoard().length; i++) {
                for (int j = 0; j < initialState.getBoard()[i].length; j++)
{
                    if (!Character.isLetter(initialState.getBoard()[i][j])
&& initialState.getBoard()[i][j] != '.') {
                        writer.write("");
                    } else {
                        writer.write(initialState.getBoard()[i][j]);
                    }
                }
                writer.newLine();
            }

            writer.newLine();
            for (int idx = 1; idx < path.size(); idx++) {
                BoardState state = path.get(idx);
                writer.write("Gerakan " + idx + ": " +
state.getLastMove().getPieceName() + "-");
                switch (state.getLastMove().getDirection()) {
                    case UP -> writer.write("atas");
                    case DOWN -> writer.write("bawah");
                    case LEFT -> writer.write("kiri");
                    case RIGHT -> writer.write("kanan");
                }
                writer.newLine();
                for (int i = 0; i < state.getBoard().length; i++) {
                    for (int j = 0; j < state.getBoard()[i].length; j++) {
                        if (!Character.isLetter(state.getBoard()[i][j]) &&
state.getBoard()[i][j] != '.') {
                            writer.write("");
                        } else {
                            writer.write(state.getBoard()[i][j]);
```

```
                        }
                    }
                    writer.newLine();
                }
                writer.newLine();
            }

        } catch (IOException e) {
            System.out.println("Error writing to file: " + e.getMessage());
        }
    }

    public static String getOutputFileName() {
        int count = 1;
        File outputDir = new File("test/result"); // Define base directory
consistently

        // Ensure the directory exists when generating the filename too
        if (!outputDir.exists()) {
            outputDir.mkdirs();
        }

        File filePath = new File(outputDir, "output" + count + ".txt");
        while (filePath.exists()) {
            count++;
            filePath = new File(outputDir, "output" + count + ".txt");
        }
        return "output" + count + ".txt";
    }
}
```

4.12 Utils.java

```
package utils;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import game.Coordinate;
import game.Piece;
import javafx.scene.paint.Color;
```

```java
/* Kelas untuk fungsi-fungsi utilitas */
public class Utils {
    // mengecek apakah nama piece sudah ada di dalam list
    public static boolean isPieceInList(char name, List<Piece> pieces) {
        for (Piece piece : pieces) {
            if (piece.getName() == name) {
                return true;
            }
        }
        return false;
    }

    // mengecek apakah piece horizontal berdasarkan list koordinatnya
    public static boolean isPieceHorizontal(List<Coordinate> coordinates) {
        // jika semua koordinat memiliki baris yang sama, maka piece
horizontal
        for (int i = 1; i < coordinates.size(); i++) {
            if (coordinates.get(i).getRow() != coordinates.get(0).getRow())
{
                return false;
            }
        }
        return true;
    }

    // membentuk board baru dengan menambahkan 2 baris dan 2 kolom kosong
    public static char[][] buildNewBoard(char[][] board) {
        char[][] newBoard = new char[board.length+2][board[0].length+2];

        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                newBoard[i+1][j+1] = board[i][j];
            }
        }

        return newBoard;
    }

    // mengecek apakah piece menempati koordinat
    public static boolean isPieceOnCoordinate(Piece piece, Coordinate
coordinate) {
        for (Coordinate coord : piece.getCoordinates()) {
            if (coord.equals(coordinate)) {
                return true;
            }
        }
        return false;
```

```
    }

    // Map berisi pasangan huruf dengan warna untuk image
    public static final Map<Character, Color> imageColorMap = new
HashMap<>();
    static {
        imageColorMap.put('K', Color.rgb(0, 255, 102));
        imageColorMap.put('P', Color.rgb(204, 0, 0));
        imageColorMap.put('.', Color.rgb(96, 96, 96));
        imageColorMap.put(' ', Color.WHITE); // warna putih untuk spasi
    }
}
```

4.13 Controller.java

```
package com.tuciltiga.controller;

import java.io.File;
import java.util.List;

import game.AStar;
import game.BlockingPiecesHeuristic;
import game.BoardState;
import game.DistanceToExitHeuristic;
import game.GBFS;
import game.IDAStar;
import game.Move;
import game.Piece;
import game.UCS;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.animation.TranslateTransition;
import javafx.fxml.FXML;
import javafx.scene.Node;
import javafx.scene.control.Alert;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.FileChooser;
import javafx.util.Duration;
```

```
import utils.IOHandler;

public class Controller {
    @FXML
    TextField configPathField = new TextField();

    @FXML
    ComboBox<Algorithm> algorithmCombo = new ComboBox<>();

    @FXML
    ComboBox<Heuristic> heuristicCombo = new ComboBox<>();

    @FXML
    Label moveCountLabel = new Label();

    @FXML
    Pane gameBoard = new Pane();

    @FXML
    Label runtimeLabel = new Label();

    @FXML
    Label nodeCountLabel = new Label();

    @FXML
    Timeline timeline;

    BoardState initialState;

    private int stateIdx = 0;

    private List<BoardState> solutionPath;
    private int nodeCount;

    private final int cellSize = 50;

    @FXML
    public void initialize() {
        // Setup algorithm choices
        algorithmCombo.getItems().setAll(Algorithm.values());
        algorithmCombo.getSelectionModel().selectFirst();
        heuristicCombo.getItems().setAll(Heuristic.values());
        heuristicCombo.getSelectionModel().selectFirst();
    }

    @FXML
    public void handleFileBrowse() {
```

```
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Select config file");
        fileChooser.getExtensionFilters().add(new
FileChooser.ExtensionFilter("Text Files", "*.txt"));

        File file =
fileChooser.showOpenDialog(configPathField.getScene().getWindow());
        if (file != null) {
            configPathField.setText(file.getAbsolutePath());
            loadBoard(file);
        }
    }

    private void loadBoard(File configFile) {
        try {
            initialState = IOHandler.convertInput(configFile);
            gameBoard.getChildren().clear();
            displayBoard(initialState);
        } catch (Exception e) {
            showAlert("Error", "Failed to load board configuration: " +
e.getMessage());
        }
    }

    private void displayBoard(BoardState boardState) {
        gameBoard.setPrefSize(boardState.getBoard()[0].length * cellSize,
boardState.getBoard().length * cellSize);
        gameBoard.setStyle("-fx-background-color: white; -fx-border-color:
black; -fx-border-width: 2px;");
        gameBoard.getChildren().clear();

        int row = boardState.getBoard().length;
        int col = boardState.getBoard()[0].length;

        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (boardState.getBoard()[i][j] == '.') {
                    Rectangle rect = new Rectangle(cellSize, cellSize);
                    rect.setFill(Color.GRAY);
                    rect.setStroke(Color.BLACK);
                    rect.setTranslateX(j * cellSize);
                    rect.setTranslateY(i * cellSize);
                    gameBoard.getChildren().add(rect);
                } else if (boardState.getBoard()[i][j] == 'K') {
                    Rectangle rect = new Rectangle(cellSize, cellSize);
                    rect.setFill(Color.GREEN);
                    rect.setTranslateX(j * cellSize);
```

```
                    rect.setTranslateY(i * cellSize);
                    gameBoard.getChildren().add(rect);
                }
            }
        }

        for (Piece piece : boardState.getPieces()) {
            Rectangle rect = new Rectangle(cellSize * piece.getWidth(),
cellSize * piece.getHeight());
            rect.setFill(piece.isPrimary() ? Color.RED : Color.BLUE);
            rect.setStroke(Color.BLACK);
            rect.setTranslateX(piece.getCoordinates().get(0).getCol() *
cellSize);
            rect.setTranslateY(piece.getCoordinates().get(0).getRow() *
cellSize);
            rect.setUserData(piece.getName());
            gameBoard.getChildren().add(rect);
        }

    }

    @FXML
    private void handleSolve() {
        Algorithm algorithm = algorithmCombo.getValue();
        Heuristic heuristic = heuristicCombo.getValue();
        String configPath = configPathField.getText();

        if (configPath.isEmpty() || !isPathValid(configPath)) {
            showAlert("Error", "Please select a valid config file first");
            return;
        }

        if (initialState == null) {
            showAlert("Error", "No board configuration loaded");
            return;
        }

        if (algorithm == null) {
            showAlert("Error", "Please select an algorithm");
            return;
        }

        if (heuristic == null) {
            showAlert("Error", "Please select a heuristic");
            return;
        }
```

```
        switch (heuristic) {
            case DistanceToExit -> {
                initialState.setHeuristic(new DistanceToExitHeuristic());
                break;
            }
            case BlockingVehicles -> {
                initialState.setHeuristic(new BlockingPiecesHeuristic());
                break;
            }
        }

        long startTime = System.currentTimeMillis();

        solver(algorithm, initialState);
        if (solutionPath == null) {
            showAlert("Error", "No solution found");
            return;
        }

        updateSolutionPath();
        long endTime = System.currentTimeMillis();
        long runTime = endTime - startTime;
        runtimeLabel.setText(String.valueOf(runTime));
        moveCountLabel.setText(String.valueOf(solutionPath.size()-1));
        nodeCountLabel.setText(String.valueOf(nodeCount));

        if (!solutionPath.isEmpty()) {
            displayBoard(solutionPath.get(0));
        }

        animate();
        if (timeline != null) {
           timeline.play();
        }
    }

    private void animate() {
        if (solutionPath == null || solutionPath.size() <= 1) {
            return;
        }

        if (timeline != null) {
            timeline.stop();
        }

        timeline = new Timeline();
        stateIdx = 0;
```

```
        for (int i = 1; i < solutionPath.size(); i++) {
            int toIdx = i;

            KeyFrame frame = new KeyFrame(Duration.seconds(0.5 * i), e -> {
                animateTransition(toIdx);
            });
            timeline.getKeyFrames().add(frame);
        }

        timeline.setOnFinished(event -> {
            if (solutionPath != null && !solutionPath.isEmpty()) {
                stateIdx = solutionPath.size() - 1;
            }
            System.out.println("Animation finished.");
        });
    }

    private void animateTransition(int toStateIndex) {
        if (solutionPath == null || toStateIndex < 1 || toStateIndex >=
solutionPath.size()) {
            return;
        }

        BoardState targetState = solutionPath.get(toStateIndex);

        Move move = targetState.getLastMove();

        if (move == null) {
            System.err.println("Error: Move is null for state at index " +
toStateIndex);
            displayBoard(targetState);
            return;
        }

        Rectangle pieceRect = findPieceBlock(move.getPieceName());

        if (pieceRect == null) {
            System.err.println("Error: Could not find piece " +
move.getPieceName() + " on the game board for animation.");
            displayBoard(targetState);
            return;
        }

        TranslateTransition moveAnimation = new
TranslateTransition(Duration.seconds(0.2), pieceRect);
        switch (move.getDirection()) {
```

```
            case LEFT:
                moveAnimation.setByX(-cellSize);
                break;
            case RIGHT:
                moveAnimation.setByX(cellSize);
                break;
            case UP:
                moveAnimation.setByY(-cellSize);
                break;
            case DOWN:
                moveAnimation.setByY(cellSize);
                break;
        }

        moveAnimation.setOnFinished(event -> {
            displayBoard(targetState);
        });

        moveAnimation.play();
    }

    private void updateSolutionPath() {
        if (solutionPath == null) {
            showAlert("Error", "No solution path available.");
        }

        BoardState lastState = solutionPath.get(solutionPath.size() - 1);
        Move.Direction exitDir = lastState.getDirectionToExit();
        switch (exitDir) {
            case UP -> {
                while (lastState.getPrimaryPiece().getCoordinates().size() >
1) {

lastState.getPrimaryPiece().removeCoordinateAtCoordinate(lastState.getExitCo
ordinate());
                    BoardState newState =
lastState.movePiece(lastState.getPrimaryPiece(), Move.Direction.UP);
                    solutionPath.add(newState);
                    lastState = solutionPath.get(solutionPath.size() - 1);
                }
            }
            case DOWN -> {
                while (lastState.getPrimaryPiece().getCoordinates().size() >
1) {

lastState.getPrimaryPiece().removeCoordinateAtCoordinate(lastState.getExitCo
ordinate());
```

```
                    BoardState newState =
lastState.movePiece(lastState.getPrimaryPiece(), Move.Direction.DOWN);
                    solutionPath.add(newState);
                    lastState = solutionPath.get(solutionPath.size() - 1);
                }
            }
            case LEFT -> {
                while (lastState.getPrimaryPiece().getCoordinates().size() >
1) {

lastState.getPrimaryPiece().removeCoordinateAtCoordinate(lastState.getExitCo
ordinate());
                    BoardState newState =
lastState.movePiece(lastState.getPrimaryPiece(), Move.Direction.LEFT);
                    solutionPath.add(newState);
                    lastState = solutionPath.get(solutionPath.size() - 1);
                }
            }
            case RIGHT -> {
                while (lastState.getPrimaryPiece().getCoordinates().size() >
1) {

lastState.getPrimaryPiece().removeCoordinateAtCoordinate(lastState.getExitCo
ordinate());
                    BoardState newState =
lastState.movePiece(lastState.getPrimaryPiece(), Move.Direction.RIGHT);
                    solutionPath.add(newState);
                    lastState = solutionPath.get(solutionPath.size() - 1);
                }
            }
        }
    }

    private Rectangle findPieceBlock(char pieceName) {
        for (Node node : gameBoard.getChildren()) {
            if (node instanceof Rectangle && node.getUserData() != null &&
node.getUserData().equals(pieceName)) {
                return (Rectangle) node;
            }
        }
        return null;
    }

    private void solver(Algorithm algorithm, BoardState initialState) {
        switch (algorithm) {
            case GBFS -> {
                Object[] result = GBFS.solve(initialState);
```

```
                if (result == null) {
                    showAlert("Error", "No solution found");
                    return;
                }
                solutionPath = (List<BoardState>) result[0];
                nodeCount = (int) result[1];
            }
            case ASTAR -> {
                Object[] result = AStar.solve(initialState);
                if (result == null) {
                    showAlert("Error", "No solution found");
                    return;
                }
                solutionPath = (List<BoardState>) result[0];
                nodeCount = (int) result[1];
            }
            case UCS -> {
                Object[] result = UCS.solve(initialState);
                if (result == null) {
                    showAlert("Error", "No solution found");
                    return;
                }
                solutionPath = (List<BoardState>) result[0];
                nodeCount = (int) result[1];
            }
            case IDA -> {
                Object[] result = IDAStar.solve(initialState);
                if (result == null) {
                    showAlert("Error", "No solution found");
                    return;
                }
                solutionPath = (List<BoardState>) result[0];
                nodeCount = (int) result[1];
            }
        }
    }

    private boolean isPathValid(String path) {
        File file = new File(path);
        return file.exists() && file.isFile() && file.canRead();
    }

    @FXML
    private void handleReset() {
        configPathField.clear();
        solutionPath = null;
        stateIdx = 0;
```

```
        nodeCount = 0;
        initialState = null;
        moveCountLabel.setText("0");
        runtimeLabel.setText("0.0");
        nodeCountLabel.setText("0");
        gameBoard.getChildren().clear();
    }

    private void showAlert(String title, String message) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }

    @FXML
    private void handlePause() {
        if (timeline != null) {
            timeline.pause();
        }
    }

    @FXML
    private void handleResume() {
        if (timeline != null) {
            timeline.play();
        }
    }

    @FXML
    private void handleReverse() {
        if (solutionPath != null && stateIdx > 0) {
            stateIdx--;
            BoardState previousState = solutionPath.get(stateIdx);
            displayBoard(previousState);
        }
    }

    @FXML
    private void handleForward() {
        if (solutionPath != null && stateIdx < solutionPath.size() - 1) {
            stateIdx++;
            BoardState nextState = solutionPath.get(stateIdx);
            displayBoard(nextState);
        }
    }
```

```
    @FXML
    private void handleSave() {
        if (solutionPath == null) {
            showAlert("Error", "No solution path available to save.");
            return;
        }

        String fileName = IOHandler.getOutputFileName();
        if (fileName == null || fileName.isEmpty()) {
            showAlert("Error", "Invalid file name.");
            return;
        }

        IOHandler.outputToFile(fileName, solutionPath);

        showAlert("Success", "Solution path saved to " + fileName);
    }
}
```

BAB 5: Pengujian

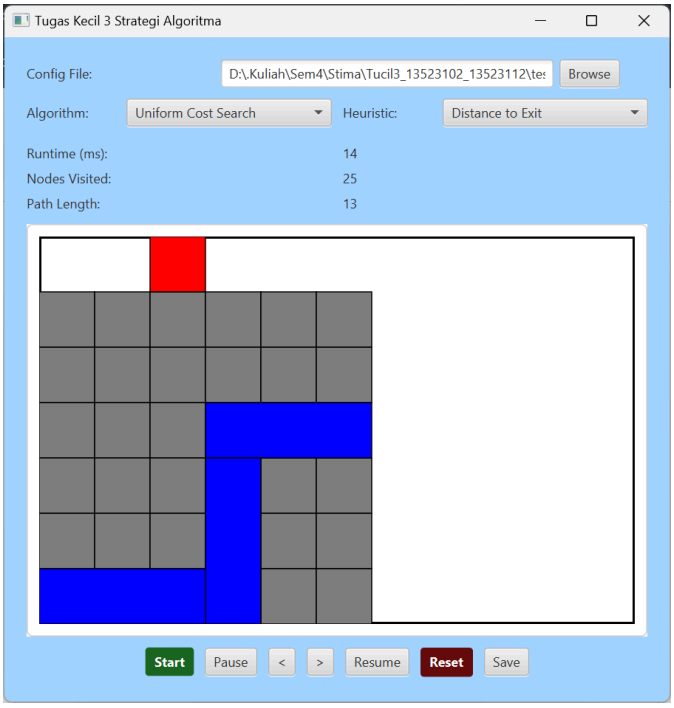
Pengujian dilakukan dengan test case yang ada pada permainan Rush Hour, diambil dari 4 level, yaitu level 1, level 20, level 30, dan level 40.

Gambar 2. Test Case Rush Hour

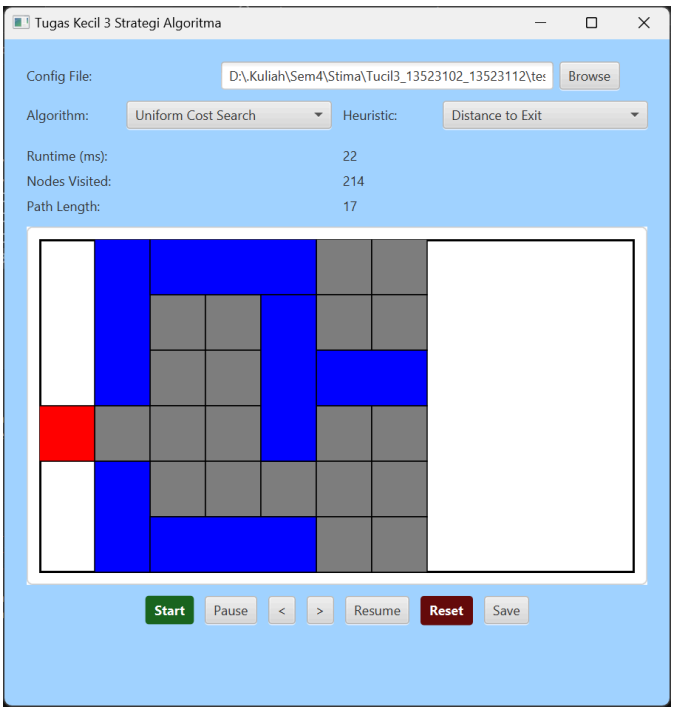
(Sumber: https://theplayfulotter.blogspot.com/2015/03/rush-hour-jr.html)

5.1 Algoritma Uniform Cost Search (UCS)

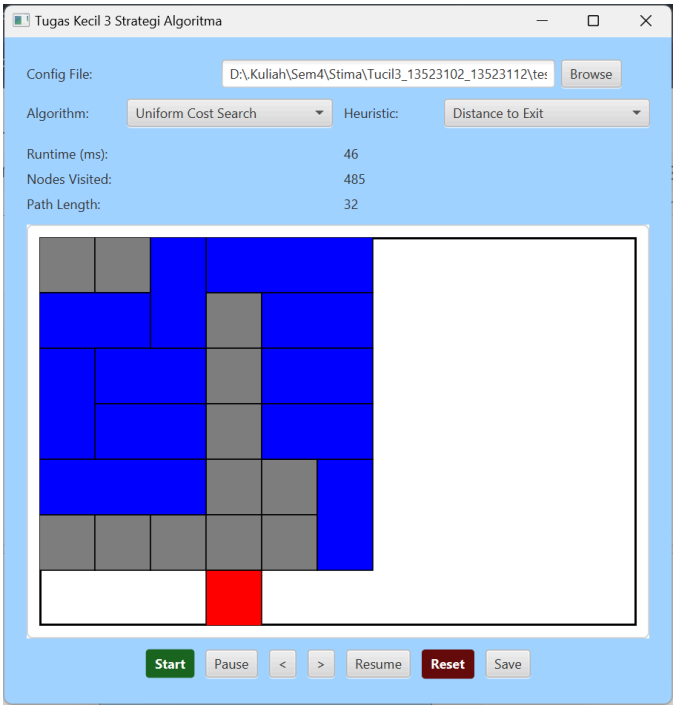
5.1.1 Rush Hour Level 1



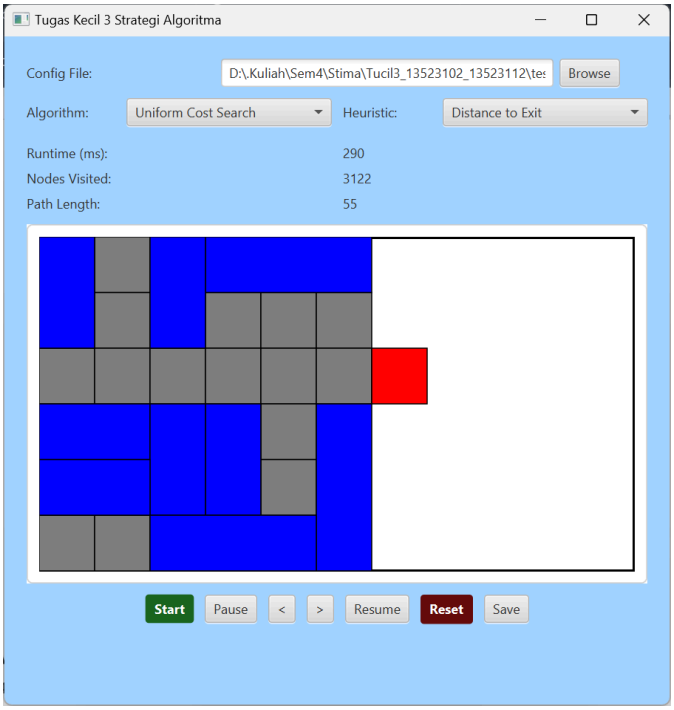
5.1.2 Rush Hour Level 20



5.1.3 Rush Hour Level 30

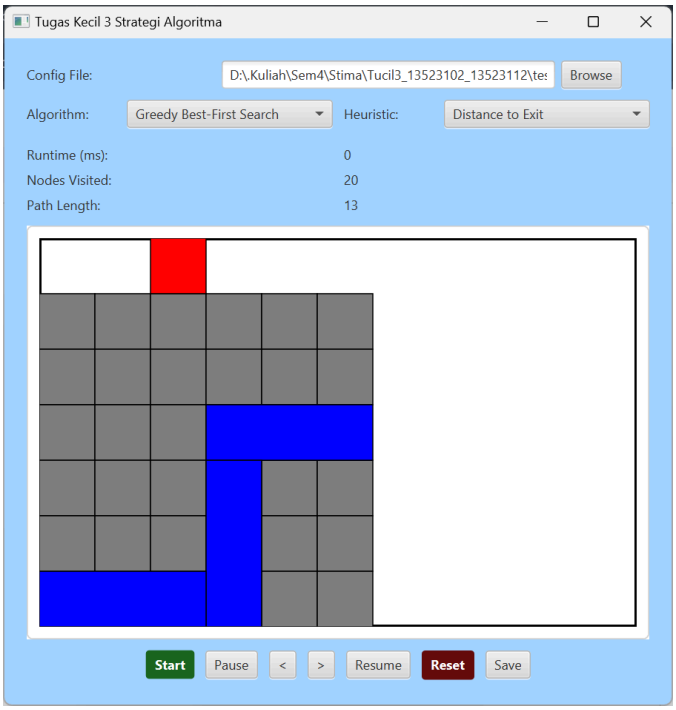


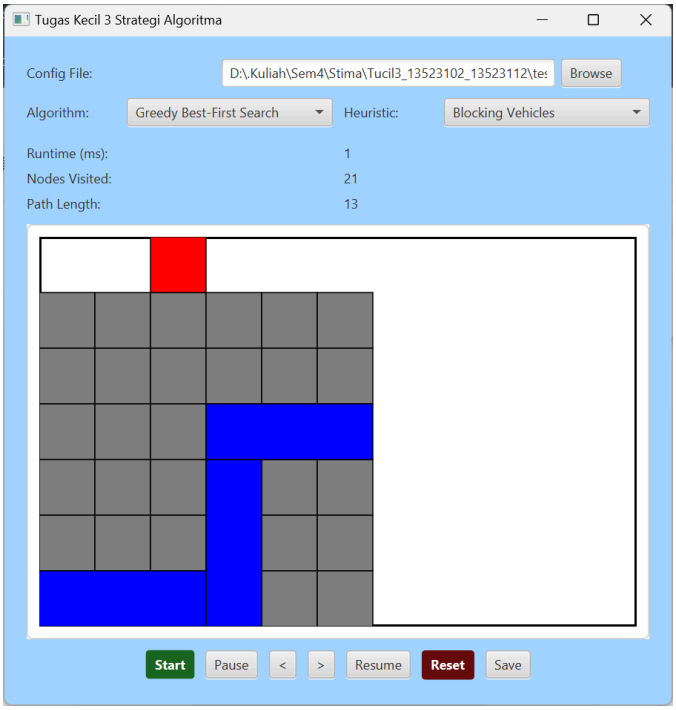
5.1.4 Rush Hour Level 40



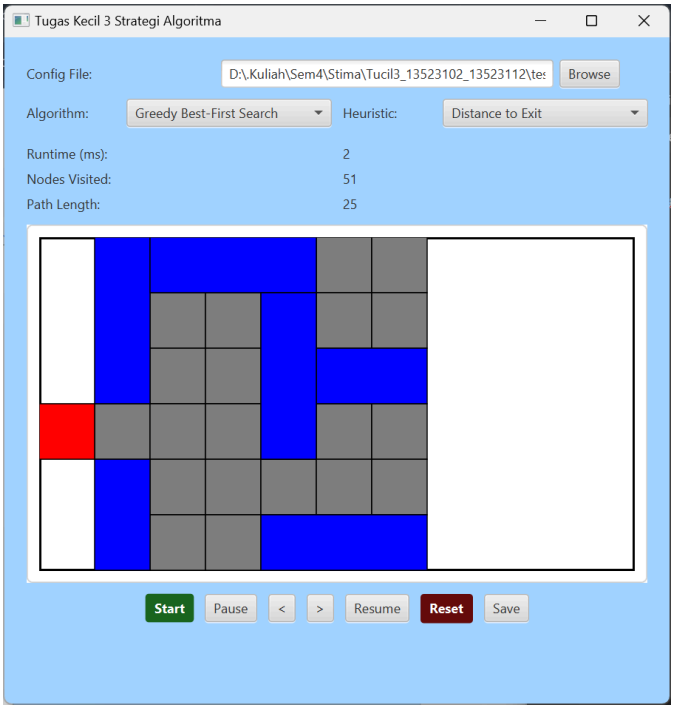
5.2 Algoritma Greedy Best First Search (GBFS)

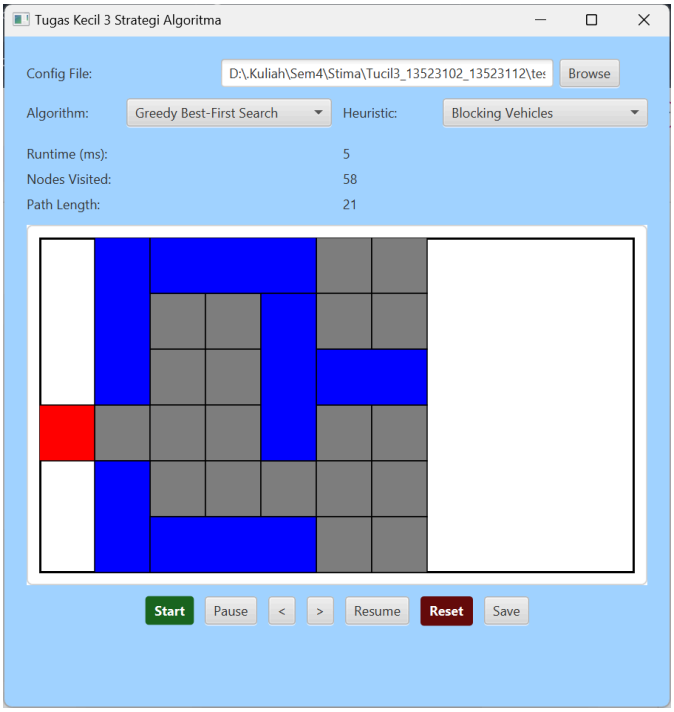
5.2.1 Rush Hour Expert Level 1



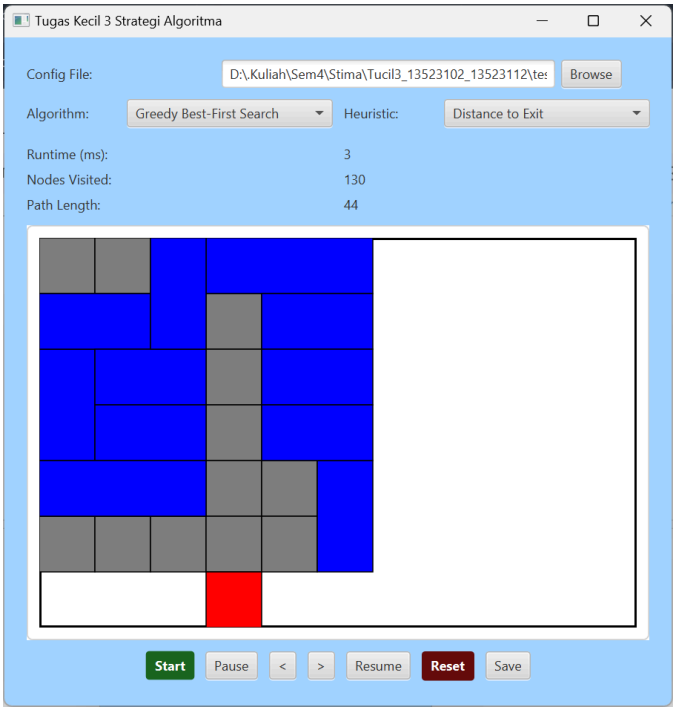


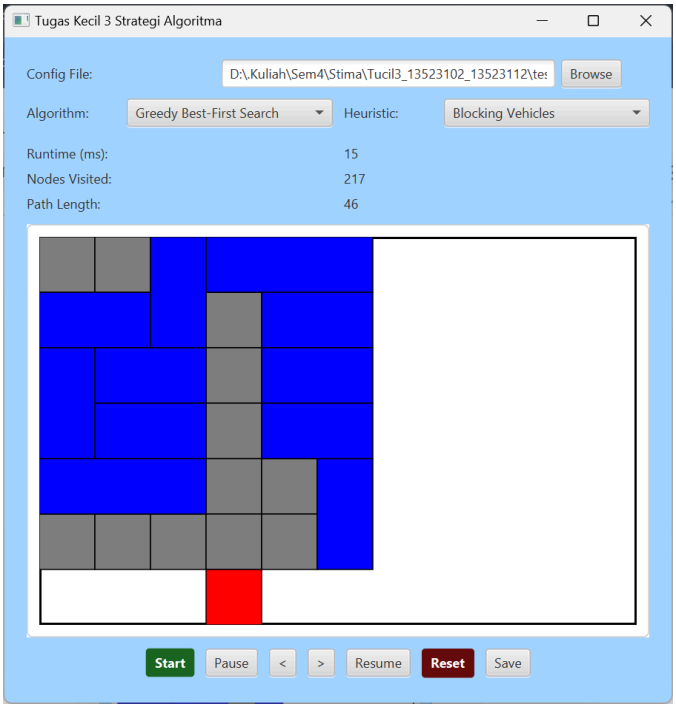
5.2.2 Rush Hour Expert Level 20



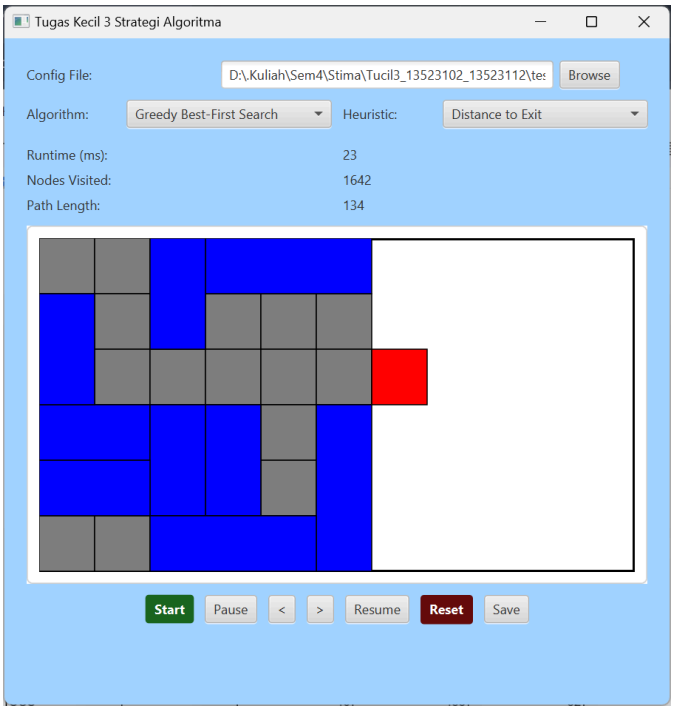


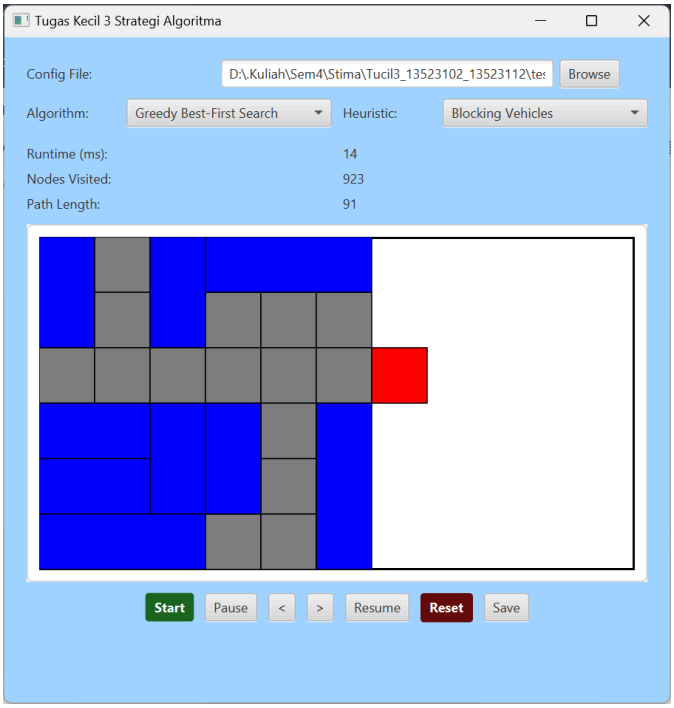
5.2.3 Rush Hour Expert Level 30





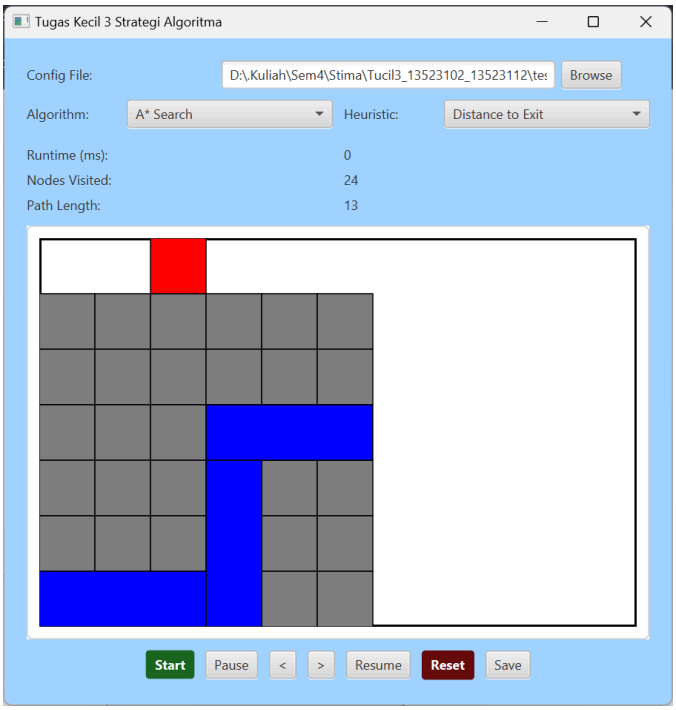
5.2.4 Rush Hour Expert Level 40

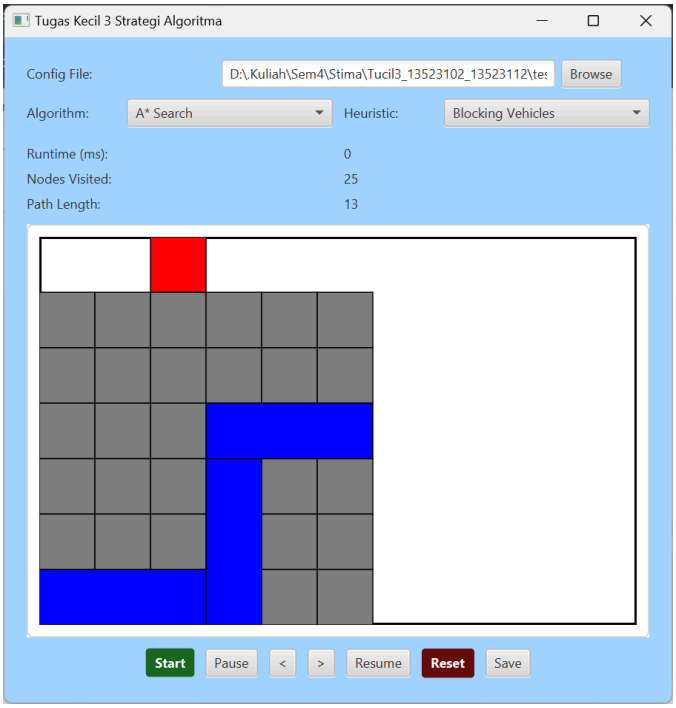




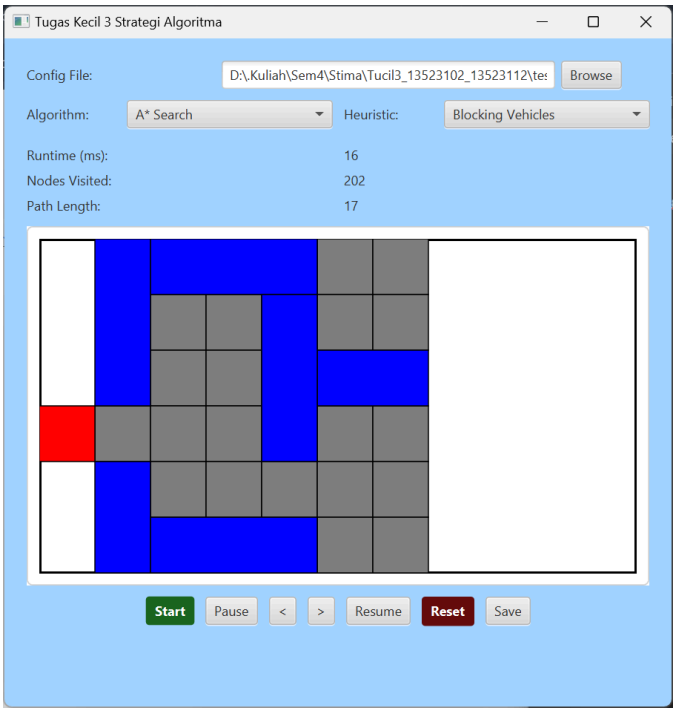
5.3 Algoritma A*

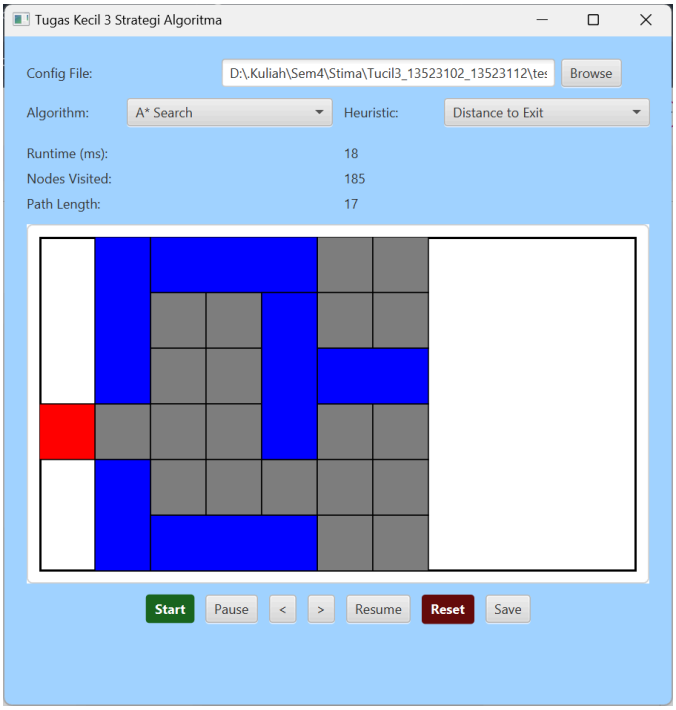
5.3.1 Rush Hour Expert Level 1



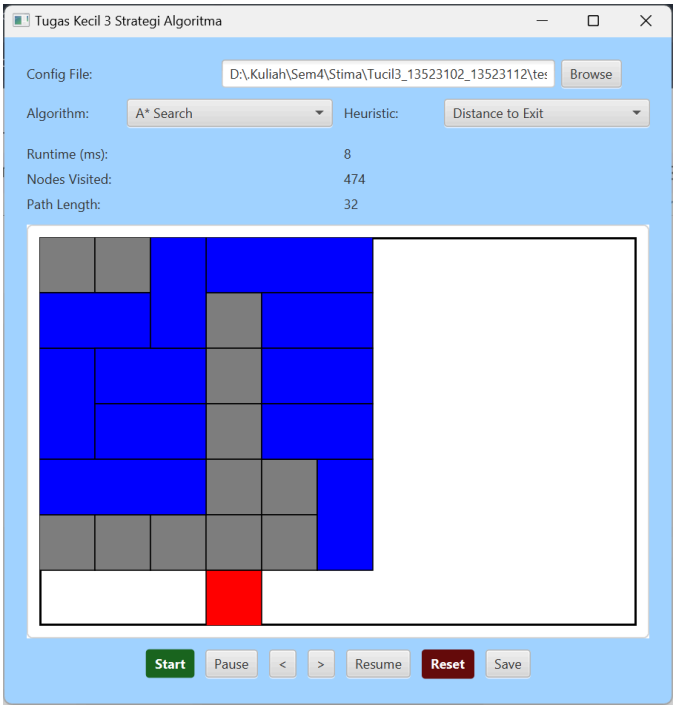


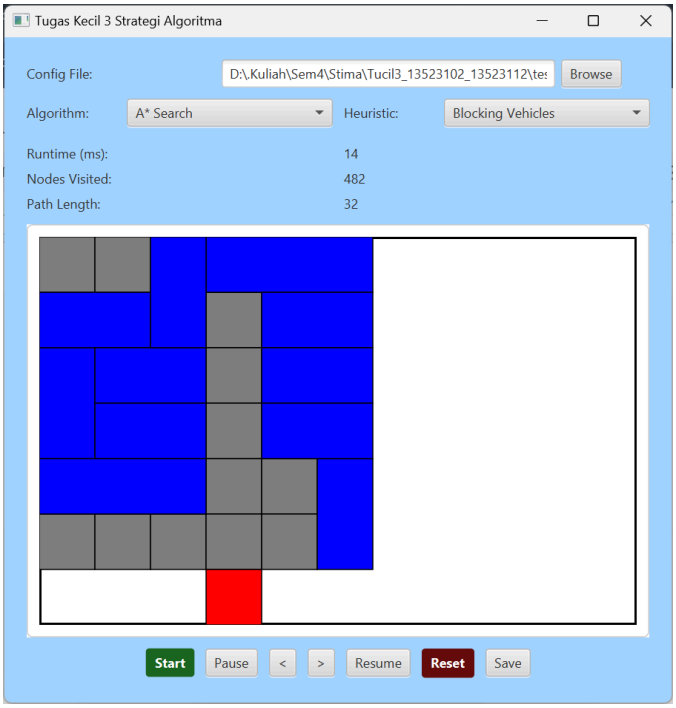
5.3.2 Rush Hour Expert Level 20



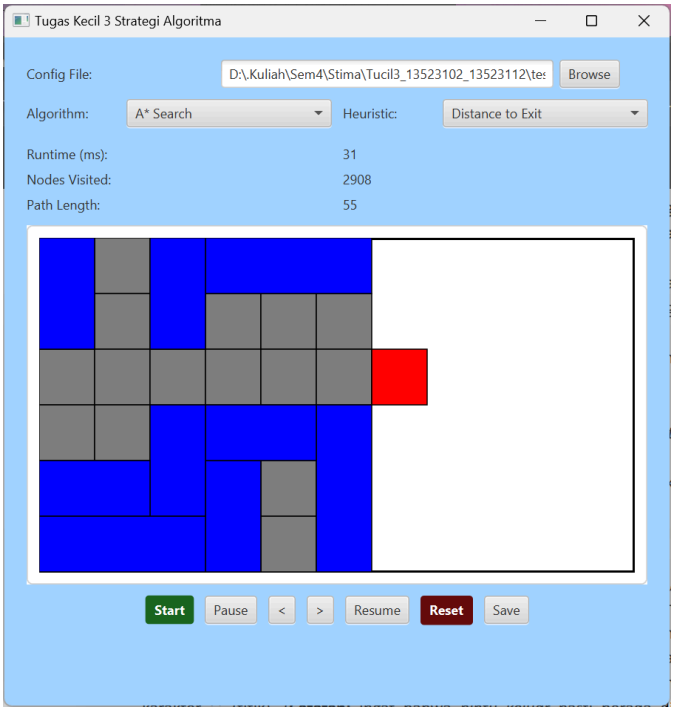


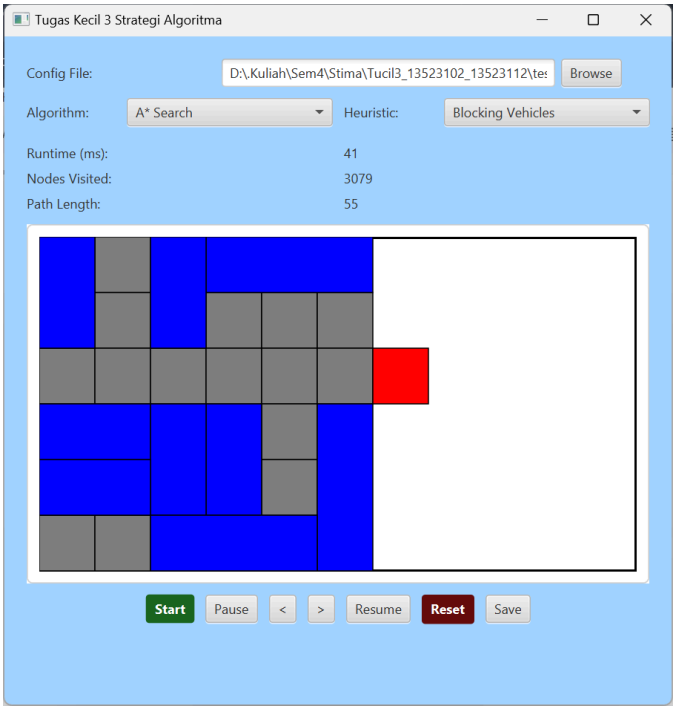
5.3.3 Rush Hour Expert Level 30





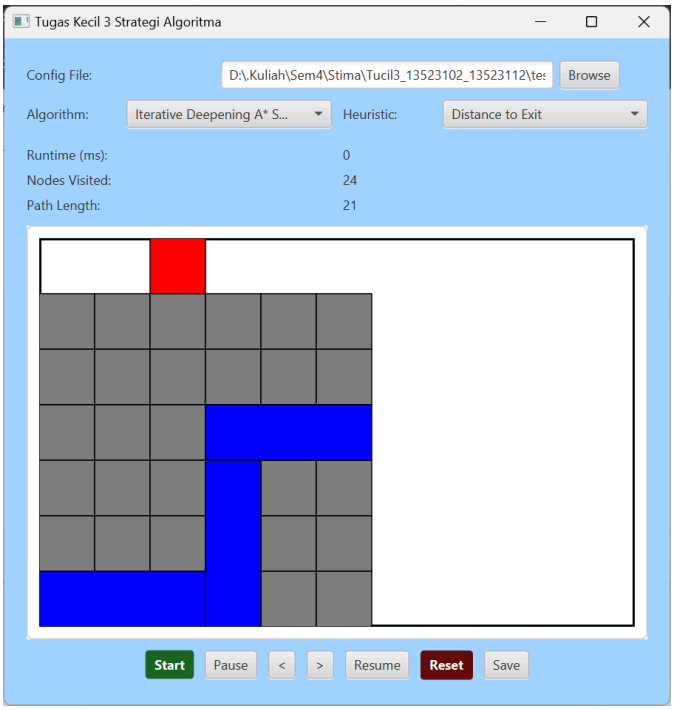
5.3.4 Rush Hour Expert Level 40

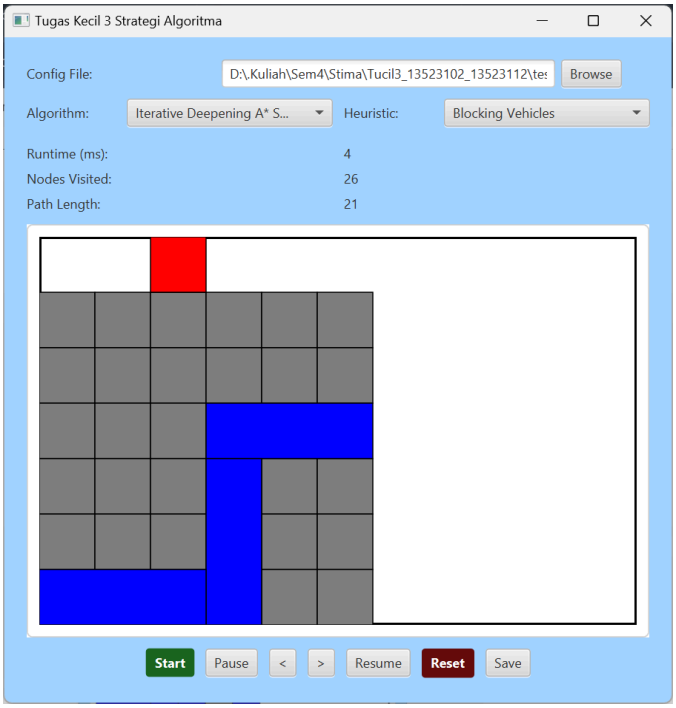




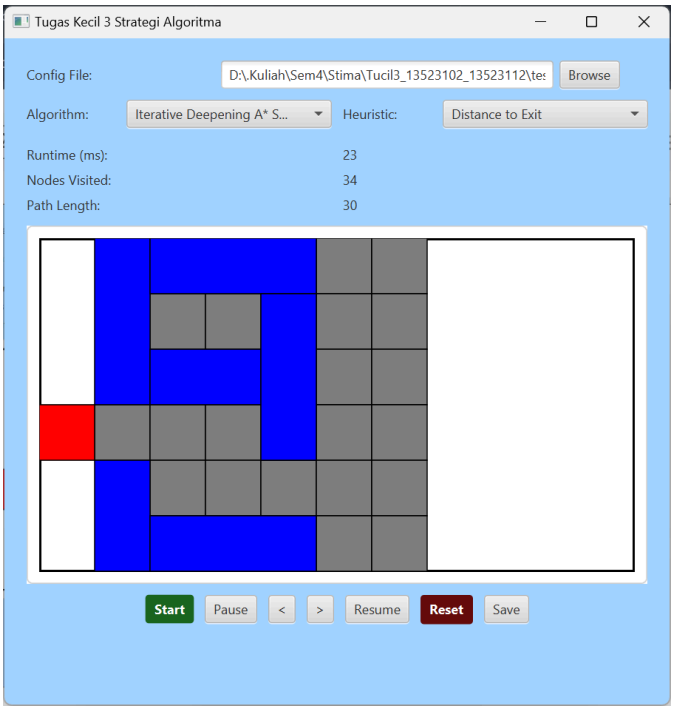
5.4 Algoritma Iterative Deepening A* (IDA*)

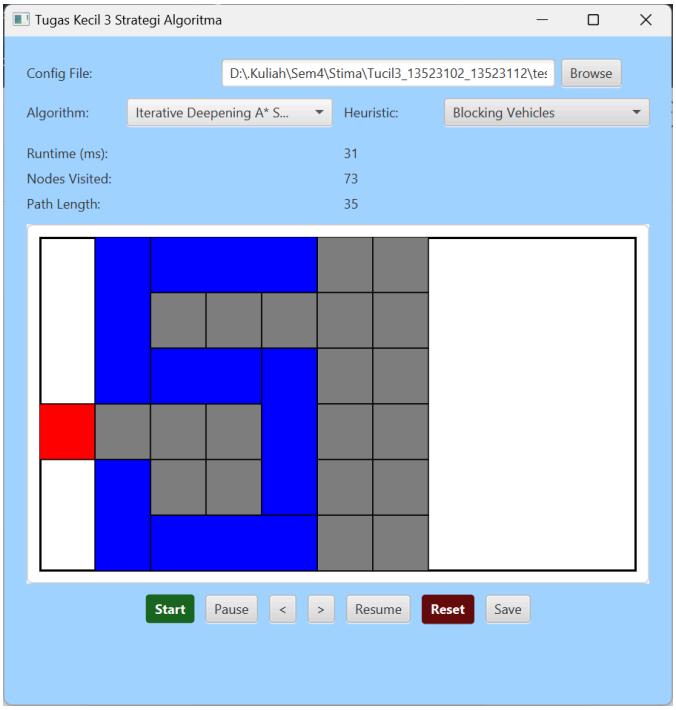
5.4.1 Rush Hour Expert Level 1



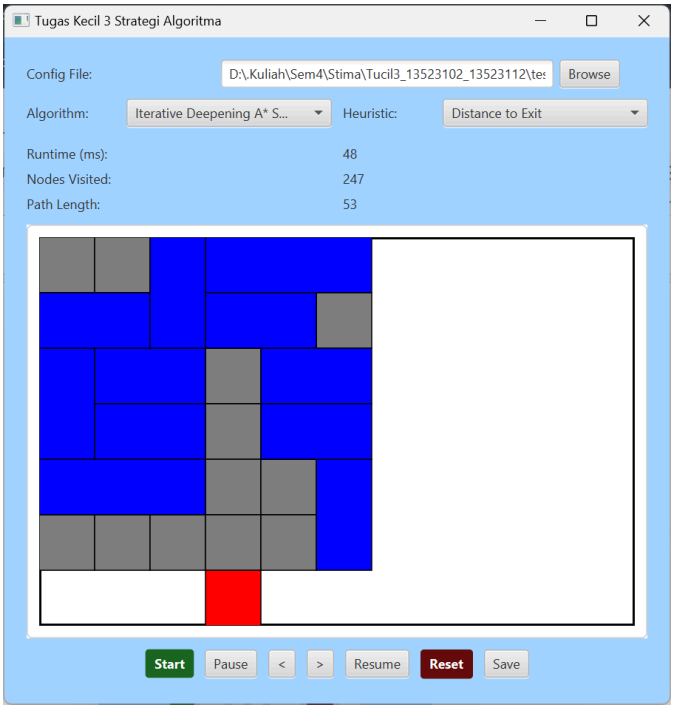


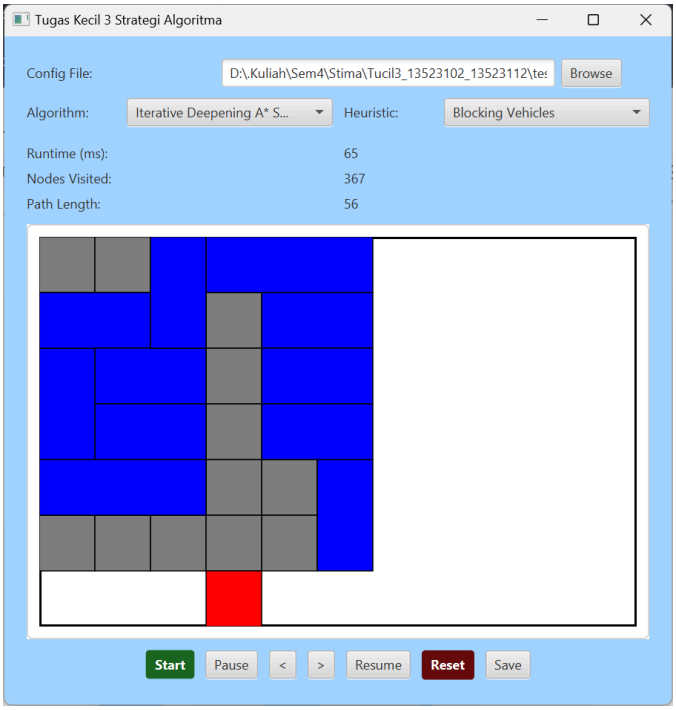
5.4.2 Rush Hour Expert Level 20



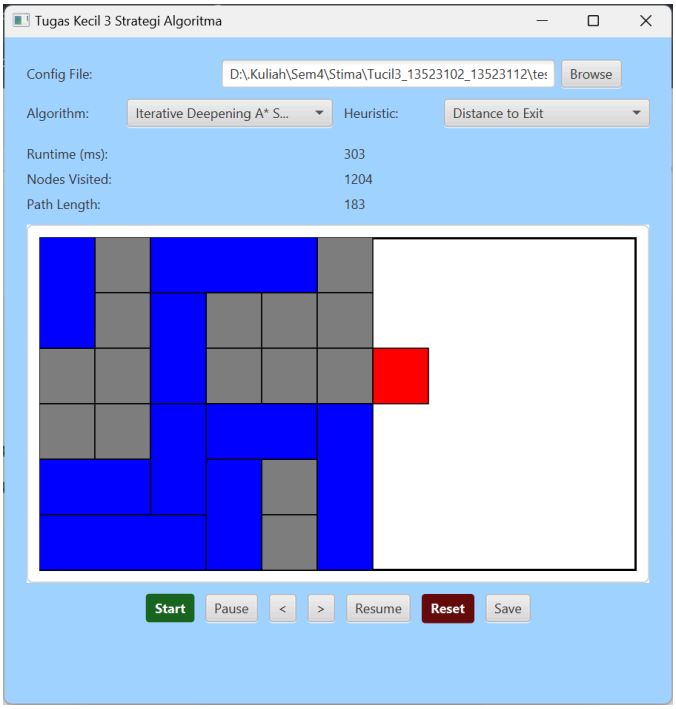


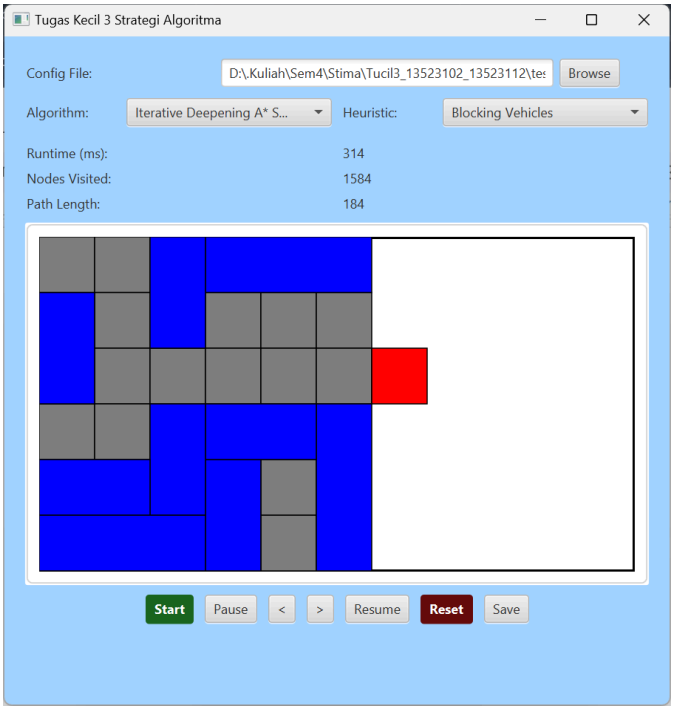
5.4.3 Rush Hour Expert Level 30





5.4.4 Rush Hour Expert Level 40




Tugas Kecil 3 Strategi Algoritma
Config File:
D:\.Kuliah\Sem4\Stima\Tucil3_13523102_13523112\te
Browse
Algorithm:
Iterative Deepening A* S...
Heuristic:
Blocking Vehicles
Runtime (ms):
314
Nodes Visited:
1584
Path Length:
184
Start
Pause
<
>
Resume
Reset
Save

BAB 6: Analisis Hasil Percobaan

Test Case Level 1				
Algoritma	Heuristik	Runtime (ms)	Nodes visited	Path length
UCS	-	14	25	13
GBFS	DistanceToExit	0	20	13
GBFS	BlockingPieces	1	21	13
A*	DistanceToExit	0	24	13
A*	BlockingPieces	0	25	13
IDA*	DistanceToExit	0	24	21
IDA*	BlockingPieces	4	26	21

Test Case Level 20				
Algoritma	Heuristik	Runtime (ms)	Nodes visited	Path length
UCS	-	22	214	17
GBFS	DistanceToExit	2	51	25
GBFS	BlockingPieces	5	58	21
A*	DistanceToExit	16	202	17
A*	BlockingPieces	18	185	17
IDA*	DistanceToExit	23	34	30
IDA*	BlockingPieces	31	73	35

Test Case Level 30				
Algoritma	Heuristik	Runtime (ms)	Nodes visited	Path length
UCS	-	46	485	32
GBFS	DistanceToExit	3	130	44
GBFS	BlockingPieces	15	217	46
A*	DistanceToExit	8	474	32
A*	BlockingPieces	14	482	32
IDA*	DistanceToExit	48	247	53
IDA*	BlockingPieces	65	367	56

Test Case Level 40				
Algoritma	Heuristik	Runtime (ms)	Nodes visited	Path length
UCS	-	290	3122	55
GBFS	DistanceToExit	23	1642	134
GBFS	BlockingPieces	14	923	91
A*	DistanceToExit	31	2908	55
A*	BlockingPieces	41	3079	55
IDA*	DistanceToExit	303	1204	183
IDA*	BlockingPieces	314	1584	184

Hasil percobaan menunjukkan bahwa masing-masing algoritma pathfinding memiliki karakteristik performa yang berbeda dalam hal runtime, jumlah node yang dikunjungi, dan panjang path solusi. Pada algoritma Uniform Cost Search (UCS), solusi yang ditemukan selalu optimal karena UCS mengeksplorasi seluruh kemungkinan berdasarkan biaya riil g(n), tanpa mempertimbangkan arah menuju goal. Namun, karena UCS merupakan algoritma uninformed, eksplorasinya tidak terarah sehingga jumlah node yang dikunjungi cenderung besar, terutama pada konfigurasi yang kompleks. Secara teoritis, kompleksitas waktu dan memori UCS adalah O(b^d), di mana b adalah branching factor dan d adalah kedalaman solusi optimal. Dalam pengujian, UCS membutuhkan waktu dan eksplorasi node yang cukup besar, tetapi tetap konsisten dalam menghasilkan path terpendek.

Algoritma Greedy Best First Search (GBFS) memiliki performa runtime tercepat dalam hampir semua test case karena eksplorasinya hanya diarahkan oleh heuristik h(n) tanpa mempertimbangkan biaya riil g(n). Pendekatan ini membuat Greedy sangat efisien dari sisi waktu, terutama pada test case besar, namun seringkali mengorbankan kualitas solusi. Dalam percobaan, Greedy BFS menghasilkan path yang lebih panjang dibanding algoritma lain, terutama pada konfigurasi yang kompleks. Meskipun rata-rata waktu eksekusinya jauh lebih rendah dibanding UCS dan A*, solusi yang dihasilkan tidak selalu optimal. Secara teoritis, kompleksitas waktu dan memori Greedy BFS adalah O(b^m), di mana m adalah kedalaman

maksimal pencarian, karena eksplorasi tetap dapat menyebar luas bila heuristik tidak cukup akurat.

Sementara itu, algoritma A* memberikan keseimbangan antara efisiensi dan optimalitas. Dengan menggunakan fungsi evaluasi f(n) = g(n) + h(n), A* mampu diarahkan menuju goal secara efisien sembari tetap mempertimbangkan total biaya dari titik awal. Hasil percobaan menunjukkan bahwa A* menghasilkan solusi dengan path yang sama optimalnya seperti UCS, namun dengan runtime yang lebih cepat dan jumlah node yang dikunjungi lebih sedikit. Secara teoritis, kompleksitas waktu dan memori A* adalah O(b^m) karena semua node dengan f(n) ≤ f(goal) harus disimpan dan diperiksa. Oleh karena itu, A* cenderung lebih efisien dari UCS dalam banyak kasus, namun tetap bisa menuntut memori yang besar pada konfigurasi sulit.

Terakhir, algoritma Iterative Deepening A* (IDA*), yang merupakan modifikasi dari A*, menunjukkan performa memori yang sangat efisien. IDA* menghindari penyimpanan semua node di memori dengan melakukan eksplorasi terbatas berdasarkan nilai ambang f(n), yang kemudian dinaikkan secara iteratif. Hasil percobaan menunjukkan bahwa IDA* mampu menghasilkan solusi yang mendekati A* dari sisi kualitas path, tetapi dengan konsumsi memori yang jauh lebih rendah. Namun, waktu eksekusinya dapat meningkat drastis, terutama pada test case yang lebih dalam dan kompleks, karena banyaknya iterasi threshold dan pengulangan eksplorasi. Secara kompleksitas, IDA* memiliki waktu O(b^m) namun memori hanya O(m), menjadikannya ideal untuk sistem dengan keterbatasan memori, meskipun trade-off-nya adalah runtime yang bisa lebih tinggi pada kasus sulit.

Berdasarkan hasil percobaan, performa dari yang terbaik hingga terburuk secara umum dapat dirangkum sebagai berikut:

Runtime: GBFS < A* < UCS ~ IDA*

Path Length: UCS = A* < GBFS < IDA*

Nodes Visited: GBFS < IDA* < A* < UCS

Dari keseluruhan aspek, A* merupakan algoritma yang paling seimbang dan unggul. Ia memberikan solusi optimal, runtime yang efisien, serta eksplorasi node yang terarah. UCS sangat

akurat namun lambat, Greedy sangat cepat namun tidak optimal, dan IDA* sangat hemat memori namun bisa menjadi lambat pada kasus besar. Oleh karena itu, pilihan terbaik tetap tergantung pada prioritas sistem: apakah mengutamakan akurasi solusi, efisiensi waktu, atau keterbatasan memori.

BAB 7: Implementasi Bonus

7.1 Implementasi Algoritma Pathfinding Alternatif

```
package game;

import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

/* Kelas untuk solver menggunakan algoritma Iterative Deepening A* */
public class IDAStar extends Algorithm {
    // Fungsi solver utama, mengembalikan array of Object berisi jalur dan
jumlah gerakan
    public static Object[] solve(BoardState initialState) {
        int threshold = initialState.getValue(); // Threshold awal adalah
nilai heuristik dari state awal

        // Set berisi state yang sudah dikunjungi
        Set<BoardState> visited = new HashSet<>();
        visited.add(initialState); // Setiap state yang dijelajah dimasukkan

        // Map untuk menyimpan parent dari setiap state
        Map<BoardState, BoardState> parentMap = new HashMap<>();

        while (true) {
            // Lakukan eksplorasi dengan batas threshold
            Result result = search(initialState, threshold, visited,
parentMap);

            if (result.isGoalFound) {
                return new Object[]{reconstructPath(parentMap,
result.goalState), result.countNode}; // jika sudah mencapai tujuan,
kembalikan jalur
            }

            if (result.nextThreshold == Integer.MAX_VALUE) {
                // Jika tidak ada threshold baru (tidak ada state yang bisa
dieksplorasi), tidak ada solusi
                return null;
            }

            // Perbarui threshold ke nilai minimum f(n) > threshold
sebelumnya
```

```
            threshold = result.nextThreshold;
            // reset visited dan parentMap
            visited.clear();
            parentMap.clear();
            visited.add(initialState); // Setiap state yang dijelajah
dimasukkan
        }
    }

    // Fungsi rekursif DFS untuk eksplorasi dengan batas threshold
    private static Result search(BoardState currentState, int threshold,
Set<BoardState> visited, Map<BoardState, BoardState> parentMap) {
        int countNode = 0; // menghitung jumlah gerakan yang dieksplorasi
        int f = currentState.getDepth() + currentState.getValue(); // f(n) =
g(n) + h(n)

        if (f > threshold) {
            // Jika f(n) melebihi threshold, kembalikan threshold baru
            return new Result(false, null, f, countNode);
        }

        if (currentState.isGoal()) {
            // Jika goal ditemukan, kembalikan hasil
            return new Result(true, currentState, threshold, countNode);
        }

        visited.add(currentState); // Tandai state sebagai dikunjungi
        int minThreshold = Integer.MAX_VALUE; // Threshold baru untuk
iterasi berikutnya

        // daftar semua langkah
        List<BoardState> possibleMoves = currentState.getPossibleMoves();

        // eksplorasi semua kemungkinan
        for (BoardState nextState : possibleMoves) {
            if (!visited.contains(nextState)) {
                countNode++;
                visited.add(nextState);
                parentMap.put(nextState, currentState);

                Result result = search(nextState, threshold, visited,
parentMap);
                countNode += result.countNode;

                if (result.isGoalFound) {
                    // jika sudah mencapai tujuan, kembalikan jalur
                    result.countNode = countNode;
```

```
                    return result;
                }

                // Perbarui threshold baru
                if (result.nextThreshold < minThreshold) {
                    minThreshold = result.nextThreshold;
                }
            }
        }

        return new Result(false, null, minThreshold, countNode);
    }

    // Kelas untuk menyimpan hasil eksplorasi
    private static class Result {
        boolean isGoalFound;
        BoardState goalState;
        int nextThreshold;
        int countNode;

        Result(boolean isGoalFound, BoardState goalState, int nextThreshold,
int countNode) {
            this.isGoalFound = isGoalFound;
            this.goalState = goalState;
            this.nextThreshold = nextThreshold;
            this.countNode = countNode;
        }
    }
}
```

7.2 Implementasi 2 atau lebih heuristik alternatif

7.2.1 DistanceToExitHeuristic

```
package game;

public class DistanceToExitHeuristic implements Heuristic {
    // Konstruktor
    public DistanceToExitHeuristic() {
    }


    @Override
    public int calcValue(BoardState state) {
        if (state.getPrimaryPiece().isHorizontal()) { // jarak
```

```
horizontal
            return
Math.abs(state.getPrimaryPiece().getCoordinates().get(0).getCol() -
state.getExitCoordinate().getCol());
        } else { // jarak vertikal
            return
Math.abs(state.getPrimaryPiece().getCoordinates().get(0).getRow() -
state.getExitCoordinate().getRow());
        }
    }
}
```

7.2.2 BlockingPiecesHeuristic

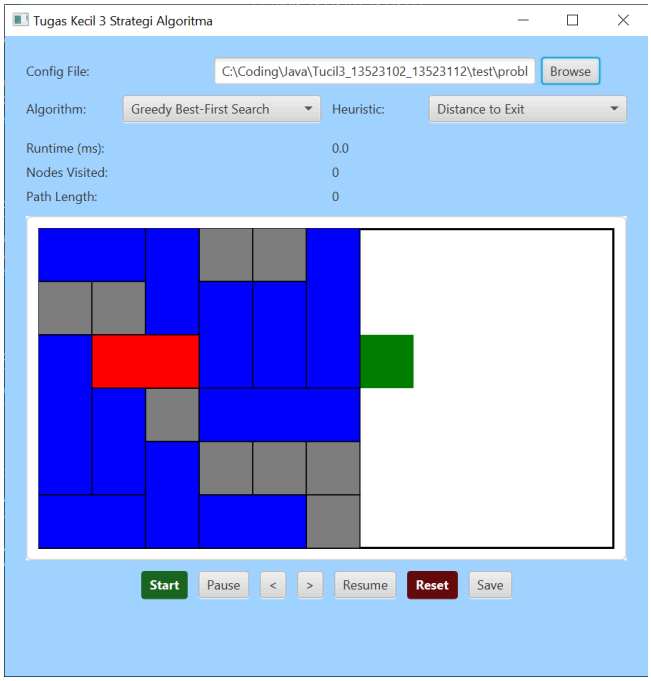
```
package game;

public class BlockingPiecesHeuristic implements Heuristic {
    // Konstruktor
    public BlockingPiecesHeuristic() {
    }

    @Override
    public int calcValue(BoardState state) {
        int blockingPiecesCount = 0;
        for (Piece piece : state.getPieces()) {
            if (piece != state.getPrimaryPiece() &&
state.isPieceBlocking(piece)) {
                blockingPiecesCount++;
            }
        }
        return blockingPiecesCount;
    }

}
```

7.3 GUI

Implementasi *Graphical User Interface* pada bahasa pemrograman Java menggunakan JavaFX dan Maven. JavaFX berperan dengan menyediakan berbagai komponen-komponen antarmuka, seperti yang dapat dilihat pada kelas *Controller.java*, dan memungkinkan integrasi antara antarmuka dengan program sesungguhnya. Di sisi

lain, Maven berperan sebagai *framework* yang mengelola segala *dependency* yang dibutuhkan oleh JavaFX pada file *pom.xml*.

Pengembangan utama GUI dilakukan pada file *mainView.fxml* dan *Controller.java*. File *mainView.fxml* bertanggung jawab untuk membentuk antarmuka pengguna *user interface* dari GUI ini, sedangkan *Controller.java* bertanggung jawab untuk mengelola sistem input pengguna dan output dari program. Pada *Controller.java* ini pun dikelola animasi yang dapat menggerakkan blok-blok pada papan menuju solusi dengan bantuan library *javafx.animation.Timeline*. Ada pun beberapa komponen lainnya, seperti *File Chooser*, *Combobox*, dan *Button* yang memudahkan dan menyederhanakan proses input dari pengguna.



Gambar 3. Tampilan GUI

Lampiran

1. **Spesifikasi Tugas Kecil**

   https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/Tucil3-Stima-2025.pdf

2. **Repository Program**

   https://github.com/TukangLas21/Tucil3_13523102_13523112

3. **Tabel Checklist**

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	✓	

Daftar Pustaka

- https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf
- https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf
- https://theplayfulotter.blogspot.com/2015/03/rush-hour-jr.html