Tugas Kecil 3 IF2211 Strategi Algoritma

Pemecahan Game Rush Hour Puzzle Menggunakan Algoritma Pathfinding

Disusun oleh :

Rafa Abdussalam Danadyaksa	13523133
Muhammad Rizain Firdaus	13523164

Program Studi Teknik Informatika
Sekolah Teknik Elektro Dan Informatika
Institut Teknologi Bandung
Jl. Ganesa 10, Bandung 40132
2025

DAFTAR ISI

BAB 1: Pendahuluan

1. Latar belakang

Gambar 1. Rush Hour

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

Uniform Cost Search (UCS) adalah algoritma pencarian graf yang menjelajahi simpul-simpul berdasarkan biaya jalur terkecil dari simpul awal ke simpul tersebut. UCS menggunakan priority queue untuk menyimpan simpul-simpul yang akan dikunjungi, diurutkan berdasarkan biaya kumulatif (g(n)) dari simpul awal ke simpul tersebut, sehingga simpul dengan biaya terendah selalu diproses terlebih dahulu. Dalam konteks Rush Hour, biaya dapat diartikan sebagai jumlah langkah untuk menggeser piece.

2.2. Algoritma Greedy Best First Search

Greedy Best First Search adalah algoritma pencarian graf yang mengutamakan eksplorasi simpul berdasarkan nilai heuristik (h(n)) terkecil, yaitu estimasi biaya dari simpul saat ini ke simpul tujuan, tanpa mempertimbangkan biaya jalur dari simpul awal (g(n)). Algoritma ini menggunakan priority queue untuk menyimpan simpul-simpul, diurutkan berdasarkan nilai heuristik, sehingga simpul yang dianggap paling dekat dengan tujuan diproses terlebih dahulu. Dalam konteks Rush Hour, heuristik dapat berupa jarak primary piece ke pintu keluar. Greedy Best First Search cepat dalam menemukan solusi karena fokus pada heuristik, tetapi tidak menjamin solusi optimal karena tidak mempertimbangkan biaya jalur sebelumnya.

2.3. Algoritma A*

A* adalah algoritma pencarian graf yang menggabungkan biaya jalur dari simpul awal (g(n)) dengan estimasi biaya ke tujuan (h(n)), menghitung nilai f(n) = g(n) + h(n) untuk setiap simpul. Algoritma ini menggunakan priority queue untuk memprioritaskan simpul dengan nilai f(n) terkecil, memastikan eksplorasi jalur yang paling menjanjikan. Dalam Rush Hour, g(n) dapat mewakili jumlah langkah pergeseran piece, dan h(n) dapat berupa estimasi langkah minimum ke pintu keluar. A* lengkap dan menjamin solusi optimal jika heuristik yang digunakan admissible (tidak melebih-estimasi biaya sebenarnya).

2.4. Algoritma Djikstra

Algoritma Dijkstra adalah algoritma pencarian graf yang digunakan untuk menemukan jalur dengan biaya terkecil dari simpul awal ke semua simpul lain atau simpul tujuan tertentu dalam graf berbobot dengan biaya non-negatif. Algoritma ini menggunakan priority queue untuk menyimpan simpul-simpul, diurutkan berdasarkan biaya kumulatif (g(n)) dari simpul awal ke simpul tersebut, mirip dengan Uniform Cost Search (UCS). Dalam konteks Rush Hour, biaya dapat diartikan sebagai jumlah langkah pergeseran piece untuk mencapai konfigurasi tertentu. Dijkstra memproses simpul dengan biaya terkecil terlebih dahulu, memperbarui biaya jalur ke tetangga-tetangganya jika ditemukan jalur yang lebih murah.

BAB 3: Analisis

3.1. Definisi f(n), h(n), dan g(n)

g(n) adalah biaya aktual dari simpul awal ke simpul n melalui jalur yang telah ditempuh. Dalam konteks Rush Hour, g(n) biasanya dihitung sebagai jumlah langkah pergeseran piece untuk mencapai konfigurasi papan pada simpul n. h(n): Nilai heuristik yang mengestimasi biaya minimum dari simpul n ke tujuan (pintu keluar). Dalam Rush Hour, h(n) dapat berupa estimasi seperti jarak dari primary piece ke pintu keluar atau jumlah piece pemblokir, yang dirancang untuk memandu pencarian tanpa melebih-estimasi biaya sebenarnya jika admissible. Dalam algoritma A*, f(n) didefinisikan sebagai f(n) = g(n) + h(n), di mana h(n) adalah nilai heuristik dan g(n) adalah biaya jalur. f(n) mewakili estimasi total biaya untuk mencapai solusi dari simpul awal melalui n.

3.2. Heuristics Admissible

Heuristik admissible adalah heuristik yang tidak pernah melebihi estimasi biaya sebenarnya untuk mencapai tujuan (h(n) ≤ h(n), di mana h(n) adalah biaya aktual).

3.3. Apakah Algoritma UCS sama dengan BFS

Dalam Rush Hour, UCS dan BFS tidak selalu menghasilkan urutan node yang sama atau jalur yang identik. BFS menjelajahi simpul berdasarkan kedalaman (jumlah langkah), mengunjungi semua simpul pada level tertentu sebelum melanjutkan ke level berikutnya, dengan asumsi setiap langkah memiliki biaya seragam (misalnya, 1 per pergeseran piece). UCS, di sisi lain, menjelajahi simpul berdasarkan biaya jalur terkecil (g(n)), yang dalam Rush Hour biasanya juga dihitung sebagai jumlah langkah (biaya seragam).

3.3. Apakah A* lebih efisien dibanding UCS

Secara teoritis, A* lebih efisien daripada UCS dalam penyelesaian Rush Hour jika heuristik yang digunakan admissible dan informative (mengurangi jumlah simpul yang dieksplorasi). UCS menjelajahi semua simpul berdasarkan biaya jalur terkecil (g(n)) tanpa mempertimbangkan estimasi ke tujuan, sehingga dapat memeriksa lebih banyak simpul, terutama pada papan dengan banyak konfigurasi.

3.4. Apakah Greedy Best First Search Menghasilkan Solusi Optimal

Secara teoritis, Greedy Best First Search tidak menjamin solusi optimal dalam penyelesaian Rush Hour. Algoritma ini hanya mempertimbangkan nilai heuristik (h(n)) untuk memilih simpul berikutnya, mengabaikan biaya jalur dari simpul awal (g(n)). Dalam Rush Hour, ini dapat menyebabkan algoritma memilih jalur yang tampak menjanjikan (misalnya, mendekatkan primary piece ke pintu keluar) tetapi memerlukan lebih banyak langkah karena tidak meminimalkan total biaya.

BAB 4 : Kode Program

4.1. Struktur Program

```
C:.
│   README.md
│   run.bat
│   run.sh
├───bin
│   │   .gitignore
│   │   Main.class
│   ├───algorithms
│   │       A$Node.class
│   │       A.class
│   │       AStar$Node.class
│   │       AStar.class
│   │       GreedyBestFirstSearch$Node.class
│   │       GreedyBestFirstSearch.class
│   │       UCS$Node.class
│   │       UCS.class
│   │
│   ├───core
│   │       Board.class
│   │       Piece.class
│   │       Step.class
│   ├───gui
│   │       ExplorationStatsPanel.class
│   │       MainWindow$1.class
│   │       MainWindow.class
│   │       StepPanel.class
│   ├───heuristics
│   │       BlockingPiecesHeuristic.class
│   │       blockingvehicle.class
│   │       DistanceToExitHeuristic.class
│   │       DistanceWithOrientationHeuristic.class
│   │       Heuristic.class
│   └───io
│           InputHandler.class
│           OutputHandler.class
│           Step.class
│
├───doc
│       initial.txt
├───src
│   │   Main.java
│   │
│   ├───algorithms
│   │       AStar.java
│   │       Dijkstra.java
│   │       GreedyBestFirstSearch.java
│   │       UCS.java
│   │
│   ├───core
│   │       Board.java
│   │       Piece.java
│   │       Step.java
│   ├───gui
│   │       ExplorationStatsPanel.java
│   │       MainWindow.java
│   │       StepPanel.java
│   ├───heuristics
│   │       BlockingPiecesHeuristic.java
│   │       DistanceToExitHeuristic.java
│   │       DistanceWithOrientationHeuristic.java
│   │       Heuristic.java
│   │
│   └───io
│           ErrorHandler.java
```

```
|           InputHandler.java
|           OutputHandler.java
└───test
```

4.2. Implementasi Component

4.2.1. Board.java

```
Board.java
```

```
public class Board {
    private int rows, cols;
    private char[][] grid;
    private List<Piece> pieces;
    private Piece primaryPiece;
    private int exitRow, exitCol;

    public Board(int rows, int cols, char[][] grid, List<Piece> pieces, int
 exitRow, int exitCol) {
        this.rows = rows;
        this.cols = cols;
        this.grid = grid;
        this.pieces = pieces;
        this.exitRow = exitRow;
        this.exitCol = exitCol;
        // Find primary piece
        for (Piece p : pieces) {
            if (p.isPrimary()) {
                this.primaryPiece = p;
                break;
            }
        }
        if (this.primaryPiece == null) {
            throw new IllegalStateException("No primary piece (P) found in the
 board");
        }
    }

    // Deep copy constructor
    public Board(Board other) {
        this.rows = other.rows;
        this.cols = other.cols;
        this.grid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            System.arraycopy(other.grid[i], 0, this.grid[i], 0, cols);
        }
        this.pieces = new ArrayList<>();
        for (Piece p : other.pieces) {
            Piece newPiece = new Piece(p.getId(), p.getRow(), p.getCol(),
 p.getSize(), p.getOrientation(),
```

Board.java

```
                p.isPrimary());
            this.pieces.add(newPiece);
            if (p.isPrimary()) {
                this.primaryPiece = newPiece;
            }
        }
        this.exitRow = other.exitRow;
        this.exitCol = other.exitCol;
    }

    // Check if a move is valid
    public boolean isValidMove(Piece piece, String direction, int steps) {
        int newRow = piece.getRow();
        int newCol = piece.getCol();

        // Calculate new position
        if (piece.getOrientation().equals("horizontal")) {
            if (direction.equals("left")) {
                newCol -= steps;
            } else if (direction.equals("right")) {
                newCol += steps;
            } else {
                return false;
            }
        } else {
            if (direction.equals("up")) {
                newRow -= steps;
            } else if (direction.equals("down")) {
                newRow += steps;
            } else {
                return false;
            }
        }

        // Check boundaries
        if (newRow < 0 || newCol < 0) {
            return false;
        }

        // Check if piece would go out of bounds
        if (piece.getOrientation().equals("horizontal")) {
            if (newCol + piece.getSize() > cols) {
                return false;
            }
        } else {
            if (newRow + piece.getSize() > rows) {
                return false;
            }
        }

        // Create temporary grid without the moving piece
```

Board.java

```java
        char[][] tempGrid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                tempGrid[i][j] = grid[i][j] == 'K' ? 'K' : '.';
            }
        }

        // Place all other pieces
        for (Piece p : pieces) {
            if (p != piece) {
                for (int[] cell : p.getOccupiedCells()) {
                    tempGrid[cell[0]][cell[1]] = p.getId();
                }
            }
        }

        // Check if new position is free
        for (int i = 0; i < piece.getSize(); i++) {
            int r = piece.getOrientation().equals("horizontal") ? newRow :
newRow + i;
            int c = piece.getOrientation().equals("horizontal") ? newCol + i :
newCol;
            if (tempGrid[r][c] != '.' && tempGrid[r][c] != 'K') {
                return false;
            }
        }

        return true;
    }

    // Move a piece and return new board
    public Board movePiece(Piece piece, String direction, int steps) {
        Board newBoard = new Board(this);
        int pieceIndex = pieces.indexOf(piece);
        if (pieceIndex == -1) {
            throw new IllegalArgumentException("Piece not found in the board");
        }
        Piece movedPiece = newBoard.pieces.get(pieceIndex);
        int newRow = movedPiece.getRow();
        int newCol = movedPiece.getCol();
        if (direction.equals("left"))
            newCol -= steps;
        else if (direction.equals("right"))
            newCol += steps;
        else if (direction.equals("up"))
            newRow -= steps;
        else if (direction.equals("down"))
            newRow += steps;

        // Update grid
        for (int i = 0; i < rows; i++) {
```

Board.java

```
            for (int j = 0; j < cols; j++) {
                newBoard.grid[i][j] = newBoard.grid[i][j] == 'K' ? 'K' : '.';
            }
        }
        movedPiece = new Piece(movedPiece.getId(), newRow, newCol,
movedPiece.getSize(), movedPiece.getOrientation(),
                movedPiece.isPrimary());
        newBoard.pieces.set(pieceIndex, movedPiece);
        if (movedPiece.isPrimary()) {
            newBoard.primaryPiece = movedPiece;
        }
        for (Piece p : newBoard.pieces) {
            for (int[] cell : p.getOccupiedCells()) {
                newBoard.grid[cell[0]][cell[1]] = p.getId();
            }
        }
        return newBoard;
    }

    // Get all possible moves
    public List<Object[]> getPossibleMoves() {
        List<Object[]> moves = new ArrayList<>();
        for (Piece piece : pieces) {
            String[] directions = piece.getOrientation().equals("horizontal") ?
new String[] { "left", "right" }
                    : new String[] { "up", "down" };
            for (String dir : directions) {
                int maxSteps = piece.getOrientation().equals("horizontal")
                        ? (dir.equals("left") ? piece.getCol() : cols -
piece.getCol() - piece.getSize())
                        : (dir.equals("up") ? piece.getRow() : rows -
piece.getRow() - piece.getSize());
                for (int steps = 1; steps <= maxSteps; steps++) {
                    if (isValidMove(piece, dir, steps)) {
                        moves.add(new Object[] { piece, dir, steps });
                    } else {
                        break;
                    }
                }
            }
        }
        return moves;
    }

    // Check if goal state is reached
    public boolean isGoalState() {
        // For horizontal primary piece, check if its right end reaches the
last column
        if (primaryPiece.getOrientation().equals("horizontal")) {
            int rightEnd = primaryPiece.getCol() + primaryPiece.getSize() - 1;
            return primaryPiece.getRow() == exitRow && rightEnd == cols - 1;
```

4.2.2. Piece.java

```java
Piece.java
```

```java
public class Piece {
    private char id;
    private int row;
    private int col;
    private int size;
    private String orientation;
    private boolean isPrimary;
    private int length;

    public Piece(char id, int row, int col, int size, String orientation, boolean
isPrimary) {
        this.id = id;
        this.row = row;
        this.col = col;
        this.size = size;
        this.orientation = orientation;
        this.isPrimary = isPrimary;
        this.length = size;
    }

    public char getId() {
        return id;
    }

    public int getRow() {
        return row;
    }

    public int getCol() {
        return col;
    }

    public int getSize() {
        return size;
    }

    public String getOrientation() {
        return orientation;
    }

    public boolean isPrimary() {
        return isPrimary;
    }

    public int getLength() {
        return length;
    }
```

Piece.java

```java

    public boolean isHorizontal() {
        return orientation.equals("H");
    }

    // Get occupied cells
    public int[][] getOccupiedCells() {
        int[][] cells = new int[size][2];
        for (int i = 0; i < size; i++) {
            if (orientation.equals("horizontal")) {
                cells[i][0] = row;
                cells[i][1] = col + i;
            } else {
                cells[i][0] = row + i;
                cells[i][1] = col;
            }
        }
        return cells;
    }
}
```

4.2.2. Step.java

Piece.java

```java
package core;

public class Step {
    private Piece piece;
    private String direction;
    private int steps;
    private Board board;

    public Step(Piece piece, String direction, int steps, Board board) {
        this.piece = piece;
        this.direction = direction;
        this.steps = steps;
        this.board = board;
    }

    public static void printBoard(Board board) {
        char[][] grid = board.getGrid();
        for (int i = 0; i < board.getRows(); i++) {
            for (int j = 0; j < board.getCols(); j++) {
                System.out.print(grid[i][j] + " ");
            }
            System.out.println();
```

Piece.java

```java
        }
        System.out.println();
    }

    public static void printSteps(java.util.List<Step> steps) {
        System.out.println("Papan Awal");
        printBoard(steps.get(0).getBoard());

        for (int i = 1; i < steps.size(); i++) {
            Step step = steps.get(i);
            System.out.println("Gerakan " + i + ": " +
                    step.getPiece().getId() + "-" +
                    step.getDirection() + " " +
                    step.getSteps() + " langkah");
            printBoard(step.getBoard());
        }
    }

    public Piece getPiece() {
        return piece;
    }

    public String getDirection() {
        return direction;
    }

    public int getSteps() {
        return steps;
    }

    public Board getBoard() {
        return board;
    }
}
```

4.3. Implementasi Algoritma

4.3.1. UCS.java

UCS.java

```
public class UCS {
    private static int nodesExplored = 0;
    private static int maxQueueSize = 0;
    private static long startTime;

    private static class Node implements Comparable<Node> {
        Board board;
        List<Step> path;
        int cost;

        Node(Board board, List<Step> path, int cost) {
            this.board = board;
            this.path = path;
            this.cost = cost;
        }

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.cost, other.cost);
        }
    }

    public static List<Step> solve(Board initialBoard) {
        nodesExplored = 0;
        maxQueueSize = 0;
        startTime = System.currentTimeMillis();

        PriorityQueue<Node> queue = new PriorityQueue<>();

        Set<String> visited = new HashSet<>();

        List<Step> initialPath = new ArrayList<>();
        initialPath.add(new Step(null, null, 0, initialBoard));
        queue.add(new Node(initialBoard, initialPath, 0));

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            String state = current.board.getState();
            nodesExplored++;
            maxQueueSize = Math.max(maxQueueSize, queue.size());

            if (visited.contains(state)) {
                continue;
            }
            visited.add(state);
```

UCS.java

```
            System.out.println("\nExploring state " + nodesExplored + ":");
            System.out.println("Current cost: " + current.cost);
            Step.printBoard(current.board);

            if (current.board.isGoalState()) {
                long endTime = System.currentTimeMillis();
                double timeInSeconds = (endTime - startTime) / 1000.0;
                System.out.println("\nSolution found!");
                System.out.printf("Exploration time: %.3f seconds%n",
timeInSeconds);
                System.out.println("Total nodes explored: " + nodesExplored);
                System.out.println("Maximum queue size: " + maxQueueSize);
                return current.path;
            }

            List<Object[]> possibleMoves = current.board.getPossibleMoves();
            System.out.println("Possible moves: " + possibleMoves.size());

            for (Object[] move : possibleMoves) {
                Piece piece = (Piece) move[0];
                String direction = (String) move[1];
                int steps = (int) move[2];
                System.out.println("  " + piece.getId() + " " + direction + " " +
steps);

                Board newBoard = current.board.movePiece(piece, direction, steps);

                if (!visited.contains(newBoard.getState())) {
                    List<Step> newPath = new ArrayList<>(current.path);
                    newPath.add(new Step(piece, direction, steps, newBoard));
                    queue.add(new Node(newBoard, newPath, current.cost + steps));
                }
            }
        }

        long endTime = System.currentTimeMillis();
        double timeInSeconds = (endTime - startTime) / 1000.0;
        System.out.println("\nNo solution found!");
        System.out.printf("Exploration time: %.3f seconds%n", timeInSeconds);
        System.out.println("Total nodes explored: " + nodesExplored);
        System.out.println("Maximum queue size: " + maxQueueSize);
        return null;
    }
    public static int getNodesExplored() {
        return nodesExplored;
    }
    public static int getMaxQueueSize() {
        return maxQueueSize;
    }
}
```

4.3.2. GreedyBestFirstSearch.java

GreedyBestFirstSearch.java

```java
public class GreedyBestFirstSearch {
    private static int nodesExplored = 0;
    private static int maxQueueSize = 0;
    private static long startTime;
    private static Heuristic heuristic;

    private static class Node implements Comparable<Node> {
        Board board;
        List<Step> path;
        int heuristicValue;

        Node(Board board, List<Step> path, int heuristicValue) {
            this.board = board;
            this.path = path;
            this.heuristicValue = heuristicValue;
        }

        public int compareTo(Node other) {
            return Integer.compare(this.heuristicValue, other.heuristicValue);
        }
    }

    public static List<Step> solve(Board initialBoard, Heuristic heuristicFunc) {
        nodesExplored = 0;
        maxQueueSize = 0;
        startTime = System.currentTimeMillis();
        heuristic = heuristicFunc;

        PriorityQueue<Node> queue = new PriorityQueue<>();

        Set<String> visited = new HashSet<>();

        List<Step> initialPath = new ArrayList<>();
        initialPath.add(new Step(null, null, 0, initialBoard));
        int initialHeuristic = heuristic.calculate(initialBoard);
        queue.add(new Node(initialBoard, initialPath, initialHeuristic));

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            String state = current.board.getState();
            nodesExplored++;
            maxQueueSize = Math.max(maxQueueSize, queue.size());

            if (visited.contains(state)) {
                continue;
```

GreedyBestFirstSearch.java

```
            }
            visited.add(state);

            System.out.println("\nExploring state " + nodesExplored + ":");
            System.out.println("Current heuristic value: " +
current.heuristicValue);
            Step.printBoard(current.board);

            if (current.board.isGoalState()) {
                long endTime = System.currentTimeMillis();
                double timeInSeconds = (endTime - startTime) / 1000.0;
                System.out.println("\nSolution found!");
                System.out.printf("Exploration time: %.3f seconds%n",
timeInSeconds);
                System.out.println("Total nodes explored: " + nodesExplored);
                System.out.println("Maximum queue size: " + maxQueueSize);
                return current.path;
            }

            List<Object[]> possibleMoves = current.board.getPossibleMoves();
            System.out.println("Possible moves: " + possibleMoves.size());

            for (Object[] move : possibleMoves) {
                Piece piece = (Piece) move[0];
                String direction = (String) move[1];
                int steps = (int) move[2];
                System.out.println("  " + piece.getId() + " " + direction + " " +
steps);

                Board newBoard = current.board.movePiece(piece, direction, steps);

                if (!visited.contains(newBoard.getState())) {
                    List<Step> newPath = new ArrayList<>(current.path);
                    newPath.add(new Step(piece, direction, steps, newBoard));
                    int newHeuristic = heuristic.calculate(newBoard);
                    queue.add(new Node(newBoard, newPath, newHeuristic));
                }
            }
        }

        long endTime = System.currentTimeMillis();
        double timeInSeconds = (endTime - startTime) / 1000.0;
        System.out.println("\nNo solution found!");
        System.out.printf("Exploration time: %.3f seconds%n", timeInSeconds);
        System.out.println("Total nodes explored: " + nodesExplored);
        System.out.println("Maximum queue size: " + maxQueueSize);
        return null;
    }
}
```

4.3.3. AStar.java

AStar.java

```
public class AStar {
    private static int nodesExplored = 0;
    private static int maxQueueSize = 0;
    private static long startTime;
    private static Heuristic heuristic;

    private static class Node implements Comparable<Node> {
        Board board;
        List<Step> path;
        int cost;
        int heuristicValue;
        int fValue;

        Node(Board board, List<Step> path, int cost, int heuristicValue) {
            this.board = board;
            this.path = path;
            this.cost = cost;
            this.heuristicValue = heuristicValue;
            this.fValue = cost + heuristicValue;
        }

        public int compareTo(Node other) {
            return Integer.compare(this.fValue, other.fValue);
        }
    }

    public static List<Step> solve(Board initialBoard, Heuristic heuristicFunc) {
        nodesExplored = 0;
        maxQueueSize = 0;
        startTime = System.currentTimeMillis();
        heuristic = heuristicFunc;

        PriorityQueue<Node> queue = new PriorityQueue<>();

        Map<String, Integer> Score = new HashMap<>();
        Set<String> visited = new HashSet<>();

        List<Step> initialPath = new ArrayList<>();
        initialPath.add(new Step(null, null, 0, initialBoard));
        int initialH = heuristic.calculate(initialBoard);
        queue.add(new Node(initialBoard, initialPath, 0, initialH));
        Score.put(initialBoard.getState(), 0);

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            String state = current.board.getState();
            nodesExplored++;
            maxQueueSize = Math.max(maxQueueSize, queue.size());
```

AStar.java

```
            if (visited.contains(state)) {
                continue;
            }

            System.out.println("\nExploring state " + nodesExplored + ":");
            System.out.println("g: " + current.cost + ", h: " +
current.heuristicValue + ", f: " + current.fValue);
            Step.printBoard(current.board);

            if (current.board.isGoalState()) {
                long endTime = System.currentTimeMillis();
                double timeInSeconds = (endTime - startTime) / 1000.0;
                System.out.println("\nSolution found!");
                System.out.printf("Exploration time: %.3f seconds%n",
timeInSeconds);
                System.out.println("Total nodes explored: " + nodesExplored);
                System.out.println("Maximum queue size: " + maxQueueSize);
                return current.path;
            }

            visited.add(state);

            List<Object[]> possibleMoves = current.board.getPossibleMoves();
            System.out.println("Possible moves: " + possibleMoves.size());

            for (Object[] move : possibleMoves) {
                Piece piece = (Piece) move[0];
                String direction = (String) move[1];
                int steps = (int) move[2];
                System.out.println("  " + piece.getId() + " " + direction + " " +
steps);

                Board newBoard = current.board.movePiece(piece, direction, steps);
                String newState = newBoard.getState();

                if (visited.contains(newState)) {
                    continue;
                }

                int newCost = current.cost + steps;
                if (!Score.containsKey(newState) || newCost < Score.get(newState)) {
                    Score.put(newState, newCost);

                    List<Step> newPath = new ArrayList<>(current.path);
                    newPath.add(new Step(piece, direction, steps, newBoard));

                    int newH = heuristic.calculate(newBoard);

                    queue.add(new Node(newBoard, newPath, newCost, newH));
                }
            }
```

```
AStar.java
```

```
        }

        long endTime = System.currentTimeMillis();
        double timeInSeconds = (endTime - startTime) / 1000.0;
        System.out.println("\nNo solution found!");
        System.out.printf("Exploration time: %.3f seconds%n", timeInSeconds);
        System.out.println("Total nodes explored: " + nodesExplored);
        System.out.println("Maximum queue size: " + maxQueueSize);
        return null;
    }
}
```

4.3.3. Djikstra.java (Implementasi Bonus)

```
Djikstra.java
```

```
public class Dijkstra {
    private static int nodesExplored = 0;
    private static int maxQueueSize = 0;
    private static long startTime;

    private static class Node implements Comparable<Node> {
        Board board;
        List<Step> path;
        int cost;

        Node(Board board, List<Step> path, int cost) {
            this.board = board;
            this.path = path;
            this.cost = cost;
        }

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.cost, other.cost);
        }
    }

    public static List<Step> solve(Board initialBoard) {
        nodesExplored = 0;
        maxQueueSize = 0;
        startTime = System.currentTimeMillis();

        PriorityQueue<Node> queue = new PriorityQueue<>();

        Set<String> visited = new HashSet<>();
```

```
Djikstra.java
```

```
        Map<String, Integer> distance = new HashMap<>();

        List<Step> initialPath = new ArrayList<>();
        initialPath.add(new Step(null, null, 0, initialBoard));
        queue.add(new Node(initialBoard, initialPath, 0));
        distance.put(initialBoard.getState(), 0);

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            String state = current.board.getState();
            nodesExplored++;
            maxQueueSize = Math.max(maxQueueSize, queue.size());

            if (visited.contains(state) && distance.get(state) < current.cost) {
                continue;
            }
            visited.add(state);

            if (current.board.isGoalState()) {
                long endTime = System.currentTimeMillis();
                double timeInSeconds = (endTime - startTime) / 1000.0;
                System.out.println("\nSolution found!");
                System.out.printf("Exploration time: %.3f seconds%n",
timeInSeconds);
                System.out.println("Total nodes explored: " + nodesExplored);
                System.out.println("Maximum queue size: " + maxQueueSize);
                return current.path;
            }

            List<Object[]> possibleMoves = current.board.getPossibleMoves();

            for (Object[] move : possibleMoves) {
                Piece piece = (Piece) move[0];
                String direction = (String) move[1];
                int steps = (int) move[2];

                Board newBoard = current.board.movePiece(piece, direction, steps);
                String newState = newBoard.getState();
                int newCost = current.cost + steps;

                if (!distance.containsKey(newState) || newCost <
distance.get(newState)) {
                    distance.put(newState, newCost);
                    List<Step> newPath = new ArrayList<>(current.path);
                    newPath.add(new Step(piece, direction, steps, newBoard));
                    queue.add(new Node(newBoard, newPath, newCost));
                }
            }
        }

        long endTime = System.currentTimeMillis();
```

```
Djikstra.java
```

```java
        double timeInSeconds = (endTime - startTime) / 1000.0;
        System.out.println("\nNo solution found!");
        System.out.printf("Exploration time: %.3f seconds%n", timeInSeconds);
        System.out.println("Total nodes explored: " + nodesExplored);
        System.out.println("Maximum queue size: " + maxQueueSize);
        return null;
    }

    public static int getNodesExplored() {
        return nodesExplored;
    }

    public static int getMaxQueueSize() {
        return maxQueueSize;
    }
}
```

4.4. Implementasi Heuristic

4.4.1. DistanceWithOrientationHeuristic.java

Heuristik DistanceWithOrientationHeuristic menghitung estimasi biaya untuk memindahkan primary piece ke pintu keluar pada permainan Rush Hour dengan mempertimbangkan jarak dan orientasi. Fungsi calculate menghitung jarak Manhattan (selisih absolut baris dan kolom) antara posisi primary piece dan pintu keluar. Selain itu, heuristik menambahkan penalti orientasi sebesar 3 jika primary piece tidak sejajar dengan pintu keluar (misalnya, horizontal tetapi barisnya berbeda dari baris pintu keluar). Penalti lain juga diterapkan, yaitu dua kali jarak baris atau kolom jika orientasi tidak sesuai, untuk mencerminkan langkah tambahan yang diperlukan. Nilai heuristik adalah jumlah jarak, penalti orientasi, dan penalti.

```
DistanceWithOrientationHeuristic.java
```

```java
public class DistanceWithOrientationHeuristic implements Heuristic {
    public int calculate(Board board) {
        Piece primaryPiece = board.getPrimaryPiece();
        int exitCol = board.getExitCol();
        int exitRow = board.getExitRow();

        int distance = Math.abs(primaryPiece.getRow() - exitRow) +
                Math.abs(primaryPiece.getCol() - exitCol);
```

```
DistanceWithOrientationHeuristic.java
```

```
        int orientationPenalty = 0;
        if (primaryPiece.isHorizontal() && primaryPiece.getRow() != exitRow) {
            orientationPenalty = 3; // Penalti lebih tinggi untuk orientasi yang
salah
        } else if (!primaryPiece.isHorizontal() && primaryPiece.getCol() != exitCol)
{
            orientationPenalty = 3;
        }

        int alignmentPenalty = 0;
        if (primaryPiece.isHorizontal() && primaryPiece.getRow() != exitRow) {
            alignmentPenalty = Math.abs(primaryPiece.getRow() - exitRow) * 2;
        } else if (!primaryPiece.isHorizontal() && primaryPiece.getCol() != exitCol)
{
            alignmentPenalty = Math.abs(primaryPiece.getCol() - exitCol) * 2;
        }

        return distance + orientationPenalty + alignmentPenalty;
    }

    public String getName() {
        return "Distance with Orientation";
    }
}
```

4.5. Implementasi Bonus Graphical User Interface

4.5.1. DistanceToExitHeuristic.java

Heuristik DistanceToExitHeuristic menghitung jarak primary piece ke pintu keluar pada permainan Rush Hour berdasarkan orientasinya. Fungsi calculate mengevaluasi posisi primary piece, jika orientasinya horizontal, heuristik menghitung jarak dari ujung kanan mobil (kolom terakhirnya) ke batas kanan papan (kolom terakhir). jika vertikal, menghitung jarak dari ujung bawah mobil (baris terakhirnya) ke batas bawah papan (baris terakhir). Nilai jarak ini, yang mengabaikan piece pemblokir, menjadi estimasi biaya minimum untuk mencapai pintu keluar.

```
DistanceToExitHeuristic.java
```

```
public class DistanceToExitHeuristic implements Heuristic {

    public int calculate(Board board) {
        Piece primaryPiece = board.getPrimaryPiece();

        if (primaryPiece.getOrientation().equals("horizontal")) {
            int rightEnd = primaryPiece.getCol() + primaryPiece.getSize() - 1;
```

DistanceToExitHeuristic.java

```
            int distance = board.getCols() - 1 - rightEnd;
            return distance;
        } else {
            int bottomEnd = primaryPiece.getRow() + primaryPiece.getSize() - 1;
            int distance = board.getRows() - 1 - bottomEnd;
            return distance;
        }
    }

    public String getName() {
        return "Distance to Exit";
    }
}
```

4.5.2. BlockingPiecesHeuristic.java

Heuristik BlockingPiecesHeuristic menghitung jumlah piece yang menghalangi primary piece (mobil utama) untuk mencapai pintu keluar pada permainan Rush Hour dengan mengevaluasi sel-sel di grid permainan. Fungsi calculate mengidentifikasi posisi primary piece dan pintu keluar, lalu memeriksa sel-sel di baris yang sama dengan primary piece mulai dari ujung mobil hingga batas papan untuk mendeteksi piece pemblokir horizontal, memberikan bobot 2 untuk setiap piece yang ditemukan karena dampaknya lebih signifikan.

Selain itu, heuristik memeriksa sel-sel di kolom yang sejajar dengan primary piece pada baris lain untuk menghitung piece pemblokir vertikal, dengan bobot 1 per piece. Nilai heuristik adalah jumlah total bobot ini, yang mencerminkan estimasi jumlah langkah minimum untuk mengatasi hambatan.

BlockingPiecesHeuristic.java

```
public class BlockingPiecesHeuristic implements Heuristic {
    public int calculate(Board board) {
        int blockingPieces = 0;
        Piece primaryPiece = board.getPrimaryPiece();
        int exitCol = board.getExitCol();
        int exitRow = board.getExitRow();

        int primaryRow = primaryPiece.getRow();
        int primaryCol = primaryPiece.getCol();
        int primaryLength = primaryPiece.getLength();

        for (int col = primaryCol + primaryLength; col < board.getCols(); col++) {
            if (board.getGrid()[primaryRow][col] != '.') {
                blockingPieces += 2; // Bobot lebih tinggi untuk blok horizontal
            }
```

```
BlockingPiecesHeuristic.java
```

```
        }

        for (int row = 0; row < board.getRows(); row++) {
            if (row != primaryRow) { // Lewati baris piece utama
                for (int col = primaryCol; col < primaryCol + primaryLength; col++)
{
                    if (board.getGrid()[row][col] != '.') {
                        blockingPieces++;
                    }
                }
            }
        }

        return blockingPieces;
    }

    public String getName() {
        return "Blocking Pieces";
    }
}
```

4.6. Implementasi Bonus Graphical User Interface

4.6.1. MainWindow.java

```
MainWindow.java
```

```
public class MainWindow extends JFrame {
    private JPanel mainPanel;
    private JPanel boardPanel;
    private JPanel controlPanel;
    private StepPanel stepPanel;
    private ExplorationStatsPanel statsPanel;
    private JButton loadFileButton;
    private JButton startButton;
    private JButton nextButton;
    private JButton prevButton;
    private JButton autoButton;
    private JButton saveButton;
    private JComboBox<String> algorithmComboBox;
    private JComboBox<String> heuristicComboBox;
    private JLabel statusLabel;
    private JSpinner delaySpinner;
    private Board currentBoard;
    private Board initialBoard;
    private List<Step> solution;
```

MainWindow.java

```
    private int currentStep = 0;
    private Timer animationTimer;
    private static final int DEFAULT_DELAY = 1000; // 1 second per step

    // Exploration statistics
    private long explorationTime;
    private int nodesExplored;
    private int maxQueueSize;

    public MainWindow() {
        setTitle("Rush Hour Puzzle Solver");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(1000, 700);
        setLocationRelativeTo(null);

        // Set dark mode colors
        Color backgroundColor = new Color(43, 43, 43);
        Color textColor = new Color(200, 200, 200);
        Color buttonColor = new Color(60, 60, 60);
        Color buttonTextColor = new Color(200, 200, 200);

        // Main panel with dark background
        mainPanel = new JPanel(new BorderLayout());
        mainPanel.setBackground(backgroundColor);
        add(mainPanel);

        // Control panel
        controlPanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
        controlPanel.setBackground(backgroundColor);

        // Load file button
        loadFileButton = new JButton("Load Config File");
        styleButton(loadFileButton, buttonColor, buttonTextColor);
        loadFileButton.addActionListener(e -> loadConfigFile());

        // Algorithm selection
        String[] algorithms = { "UCS", "Greedy Best First Search", "A*", "Dijkstra"
};
        algorithmComboBox = new JComboBox<>(algorithms);
        algorithmComboBox.setBackground(buttonColor);
        algorithmComboBox.setForeground(buttonTextColor);
        algorithmComboBox.addActionListener(e -> updateHeuristicComboBox());

        // Heuristic selection
        String[] heuristics = { "Distance to Exit", "Blocking Pieces", "Distance
with Orientation" };
        heuristicComboBox = new JComboBox<>(heuristics);
        heuristicComboBox.setBackground(buttonColor);
        heuristicComboBox.setForeground(buttonTextColor);
        heuristicComboBox.setEnabled(false);
```

```
MainWindow.java
```

```
        // Start button
        startButton = new JButton("Start Solving");
        styleButton(startButton, buttonColor, buttonTextColor);
        startButton.addActionListener(e -> startSolving());
        startButton.setEnabled(false);

        // Navigation buttons
        prevButton = new JButton("Previous");
        nextButton = new JButton("Next");
        autoButton = new JButton("Auto Play");
        saveButton = new JButton("Save Solution");
        styleButton(prevButton, buttonColor, buttonTextColor);
        styleButton(nextButton, buttonColor, buttonTextColor);
        styleButton(autoButton, buttonColor, buttonTextColor);
        styleButton(saveButton, buttonColor, buttonTextColor);
        prevButton.setEnabled(false);
        nextButton.setEnabled(false);
        autoButton.setEnabled(false);
        saveButton.setEnabled(false);

        prevButton.addActionListener(e -> showPreviousStep());
        nextButton.addActionListener(e -> showNextStep());
        autoButton.addActionListener(e -> toggleAutoPlay());
        saveButton.addActionListener(e -> saveSolution());

        // Delay spinner
        SpinnerNumberModel delayModel = new SpinnerNumberModel(DEFAULT_DELAY, 100,
5000, 100);
        delaySpinner = new JSpinner(delayModel);
        delaySpinner.setPreferredSize(new Dimension(80, 25));
        delaySpinner.setBackground(buttonColor);
        delaySpinner.setForeground(buttonTextColor);

        // Status label
        statusLabel = new JLabel("Load a config file to begin");
        statusLabel.setForeground(textColor);

        controlPanel.add(loadFileButton);
        controlPanel.add(new JLabel("Algorithm: "));
        controlPanel.add(algorithmComboBox);
        controlPanel.add(new JLabel("Heuristic: "));
        controlPanel.add(heuristicComboBox);
        controlPanel.add(startButton);
        controlPanel.add(new JLabel("Delay (ms): "));
        controlPanel.add(delaySpinner);
        controlPanel.add(prevButton);
        controlPanel.add(nextButton);
        controlPanel.add(autoButton);
        controlPanel.add(saveButton);
        controlPanel.add(statusLabel);
```

MainWindow.java

```java
        mainPanel.add(controlPanel, BorderLayout.NORTH);

        // Center panel for board and step info
        JPanel centerPanel = new JPanel(new BorderLayout());
        centerPanel.setBackground(backgroundColor);

        // Left panel for board and stats
        JPanel leftPanel = new JPanel(new BorderLayout());
        leftPanel.setBackground(backgroundColor);

        // Board panel
        boardPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                if (currentBoard != null) {
                    drawBoard(g);
                }
            }
        };
        boardPanel.setBackground(backgroundColor);
        leftPanel.add(boardPanel, BorderLayout.CENTER);

        // Stats panel
        statsPanel = new ExplorationStatsPanel();
        leftPanel.add(statsPanel, BorderLayout.SOUTH);

        centerPanel.add(leftPanel, BorderLayout.CENTER);

        // Step panel
        stepPanel = new StepPanel();
        centerPanel.add(stepPanel, BorderLayout.EAST);

        mainPanel.add(centerPanel, BorderLayout.CENTER);

        // Animation timer
        animationTimer = new Timer(DEFAULT_DELAY, e -> {
            if (solution != null && currentStep < solution.size()) {
                showNextStep();
                if (currentStep == solution.size()) {
                    animationTimer.stop();
                    autoButton.setText("Auto Play");
                    statusLabel.setText("Solution complete!");
                }
            }
        });
    }

    private void updateHeuristicComboBox() {
        String selectedAlgorithm = (String) algorithmComboBox.getSelectedItem();
        heuristicComboBox
```

MainWindow.java

```
                .setEnabled("Greedy Best First Search".equals(selectedAlgorithm) ||
"A*".equals(selectedAlgorithm));
    }

    private Heuristic getSelectedHeuristic() {
        String selectedHeuristic = (String) heuristicComboBox.getSelectedItem();
        if ("Distance to Exit".equals(selectedHeuristic)) {
            return new DistanceToExitHeuristic();
        } else if ("Blocking Pieces".equals(selectedHeuristic)) {
            return new BlockingPiecesHeuristic();
        } else if ("Distance with Orientation".equals(selectedHeuristic)) {
            return new DistanceWithOrientationHeuristic();
        }
        return null;
    }

    private void saveSolution() {
        if (solution == null)
            return;

        String filename = JOptionPane.showInputDialog(this,
                "Enter filename to save solution (without extension):",
                "Save Solution",
                JOptionPane.QUESTION_MESSAGE);

        if (filename != null && !filename.trim().isEmpty()) {
            try {
                // Add .txt extension if not present
                if (!filename.toLowerCase().endsWith(".txt")) {
                    filename += ".txt";
                }
                OutputHandler.saveSolution(filename, solution, initialBoard,
explorationTime, nodesExplored,
                        maxQueueSize);
                statusLabel.setText("Solution saved to: " + filename);
            } catch (Exception ex) {
                JOptionPane.showMessageDialog(this,
                        "Error saving solution: " + ex.getMessage(),
                        "Error",
                        JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    private void showNextStep() {
        if (solution != null && currentStep < solution.size()) {
            currentStep++;
            updateBoardAndStep();
            updateNavigationButtons();
        }
    }
```

MainWindow.java

```
    private void showPreviousStep() {
        if (currentStep > 0) {
            currentStep--;
            updateBoardAndStep();
            updateNavigationButtons();
        }
    }

    private void updateBoardAndStep() {
        if (currentStep > 0 && currentStep <= solution.size()) {
            currentBoard = solution.get(currentStep - 1).getBoard();
            stepPanel.updateStep(solution.get(currentStep - 1), currentStep,
solution.size());
            boardPanel.repaint();
        }
    }

    private void updateNavigationButtons() {
        prevButton.setEnabled(currentStep > 0);
        nextButton.setEnabled(currentStep < solution.size());
        if (currentStep == solution.size()) {
            autoButton.setEnabled(false);
        }
    }

    private void toggleAutoPlay() {
        if (animationTimer.isRunning()) {
            animationTimer.stop();
            autoButton.setText("Auto Play");
        } else {
            if (solution != null && currentStep < solution.size()) {
                int delay = (Integer) delaySpinner.getValue();
                animationTimer.setDelay(delay);
                animationTimer.start();
                autoButton.setText("Stop");
            }
        }
    }

    private void styleButton(JButton button, Color bgColor, Color textColor) {
        button.setBackground(bgColor);
        button.setForeground(textColor);
        button.setFocusPainted(false);
        button.setBorderPainted(false);
        button.setOpaque(true);
    }

    private void loadConfigFile() {
        JFileChooser fileChooser = new JFileChooser("test/input");
        if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
```

MainWindow.java

```
            File file = fileChooser.getSelectedFile();
            try {
                InputHandler inputHandler = new InputHandler();
                currentBoard = inputHandler.readInput(file.getPath());
                initialBoard = currentBoard; // Save initial board state
                startButton.setEnabled(true);
                statusLabel.setText("File loaded: " + file.getName());
                boardPanel.repaint();
                statsPanel.reset();
            } catch (Exception ex) {
                JOptionPane.showMessageDialog(this, "Error loading file: " +
ex.getMessage(),
                        "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    private void startSolving() {
        if (currentBoard == null) {
            statusLabel.setText("Please load a configuration file first!");
            return;
        }

        String selectedAlgorithm = (String) algorithmComboBox.getSelectedItem();
        List<Step> newSolution = null;
        long startTime = System.currentTimeMillis();

        if (selectedAlgorithm.equals("UCS")) {
            newSolution = UCS.solve(currentBoard);
        } else if (selectedAlgorithm.equals("Greedy Best First Search")) {
            Heuristic heuristic = getSelectedHeuristic();
            newSolution = GreedyBestFirstSearch.solve(currentBoard, heuristic);
        } else if (selectedAlgorithm.equals("A*")) {
            Heuristic heuristic = getSelectedHeuristic();
            newSolution = AStar.solve(currentBoard, heuristic);
        } else if (selectedAlgorithm.equals("Dijkstra")) {
            newSolution = Dijkstra.solve(currentBoard);
        }

        long endTime = System.currentTimeMillis();
        double timeInSeconds = (endTime - startTime) / 1000.0;

        if (newSolution != null) {
            solution = newSolution; // Store the solution
            currentStep = 0;
            stepPanel.updateStep(null, 0, solution.size());
            prevButton.setEnabled(false);
            nextButton.setEnabled(true);
            autoButton.setEnabled(true);
            saveButton.setEnabled(true);
```

MainWindow.java

```
            // Update statistics
            statsPanel.updateStats(
                    (long) (timeInSeconds * 1000),
                    selectedAlgorithm.equals("UCS") ? UCS.getNodesExplored()
                            : selectedAlgorithm.equals("Greedy Best First Search")
                                    ? GreedyBestFirstSearch.getNodesExplored()
                                    : selectedAlgorithm.equals("A*")
                                            ? AStar.getNodesExplored()
                                            : Dijkstra.getNodesExplored(),
                    selectedAlgorithm.equals("UCS") ? UCS.getMaxQueueSize()
                            : selectedAlgorithm.equals("Greedy Best First Search")
                                    ? GreedyBestFirstSearch.getMaxQueueSize()
                                    : selectedAlgorithm.equals("A*")
                                            ? AStar.getMaxQueueSize()
                                            : Dijkstra.getMaxQueueSize());
        } else {
            solution = null;
            stepPanel.updateStep(null, 0, 0);
            prevButton.setEnabled(false);
            nextButton.setEnabled(false);
            autoButton.setEnabled(false);
            saveButton.setEnabled(false);
            statusLabel.setText("No solution found!");
        }
    }

    private void drawBoard(Graphics g) {
        if (currentBoard == null)
            return;

        int cellSize = Math.min(
                boardPanel.getWidth() / currentBoard.getCols(),
                boardPanel.getHeight() / currentBoard.getRows());

        int startX = (boardPanel.getWidth() - cellSize * currentBoard.getCols()) /
2;
        int startY = (boardPanel.getHeight() - cellSize * currentBoard.getRows()) /
2;

        // Draw grid
        g.setColor(new Color(100, 100, 100));
        for (int i = 0; i <= currentBoard.getRows(); i++) {
            g.drawLine(startX, startY + i * cellSize,
                    startX + currentBoard.getCols() * cellSize, startY + i *
cellSize);
        }
        for (int i = 0; i <= currentBoard.getCols(); i++) {
            g.drawLine(startX + i * cellSize, startY,
                    startX + i * cellSize, startY + currentBoard.getRows() *
cellSize);
        }
```

MainWindow.java

```
        // Draw pieces
        char[][] grid = currentBoard.getGrid();
        for (int i = 0; i < currentBoard.getRows(); i++) {
            for (int j = 0; j < currentBoard.getCols(); j++) {
                if (grid[i][j] != '.') {
                    Color pieceColor = getPieceColor(grid[i][j]);
                    g.setColor(pieceColor);
                    g.fillRect(startX + j * cellSize + 1, startY + i * cellSize + 1,
                            cellSize - 2, cellSize - 2);
                    g.setColor(Color.WHITE);
                    g.drawString(String.valueOf(grid[i][j]),
                            startX + j * cellSize + cellSize / 2 - 4,
                            startY + i * cellSize + cellSize / 2 + 4);
                }
            }
        }

        // Draw exit
        g.setColor(new Color(255, 100, 100));
        g.drawString("K", startX + currentBoard.getExitCol() * cellSize + cellSize /
2 - 4,
                startY + currentBoard.getExitRow() * cellSize + cellSize / 2 + 4);
    }

    private Color getPieceColor(char pieceId) {
        // Diff primary piece color
        if (pieceId == 'P') {
            return new Color(0, 255, 107);
        }

        // I'm tired of finding colors so i just used random colors
        Color[] colors = {
                new Color(255, 0, 0),
                new Color(0, 255, 0),
                new Color(0, 0, 255),
                new Color(255, 0, 255),
                new Color(0, 255, 255),
                new Color(255, 128, 0),
                new Color(128, 0, 255),
                new Color(0, 128, 0),
                new Color(128, 128, 0),
                new Color(128, 0, 128),
                new Color(0, 128, 128),
                new Color(255, 165, 0),
                new Color(75, 0, 130),
                new Color(139, 69, 19)
        };

        // Use piece ID to select a color, cycling through the array
        int index = (pieceId - 'A') % colors.length;
```

```
MainWindow.java
```

```
        if (index < 0)
            index += colors.length; // Handle non-letter pieces
        return colors[index];
    }
}
```

4.6.2. StepPanel.java

```
StepPanel.java
```

```
public class StepPanel extends JPanel {
    private Step currentStep;
    private JLabel stepInfoLabel;
    private JLabel moveInfoLabel;
    private JLabel costInfoLabel;

    public StepPanel() {
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        setBackground(new Color(43, 43, 43));
        setForeground(new Color(200, 200, 200));

        stepInfoLabel = new JLabel("Step: 0/0");
        moveInfoLabel = new JLabel("Move: None");
        costInfoLabel = new JLabel("Cost: 0");

        styleLabel(stepInfoLabel);
        styleLabel(moveInfoLabel);
        styleLabel(costInfoLabel);

        add(stepInfoLabel);
        add(Box.createVerticalStrut(10));
        add(moveInfoLabel);
        add(Box.createVerticalStrut(10));
        add(costInfoLabel);
    }

    private void styleLabel(JLabel label) {
        label.setForeground(new Color(200, 200, 200));
        label.setFont(new Font("Arial", Font.PLAIN, 14));
        label.setAlignmentX(Component.CENTER_ALIGNMENT);
    }

    public void updateStep(Step step, int currentStep, int totalSteps) {
        this.currentStep = step;
        stepInfoLabel.setText(String.format("Step: %d/%d", currentStep,
totalSteps));
```

StepPanel.java

```
        if (step != null && step.getPiece() != null) {
            Piece piece = step.getPiece();
            moveInfoLabel.setText(String.format("Move: %s %s %d steps",
                    piece.getId(), step.getDirection(), step.getSteps()));
            costInfoLabel.setText(String.format("Cost: %d", step.getSteps()));
        } else {
            moveInfoLabel.setText("Move: None");
            costInfoLabel.setText("Cost: 0");
        }
    }
}
```

4.6.3. ExplorationStatsPanel.java

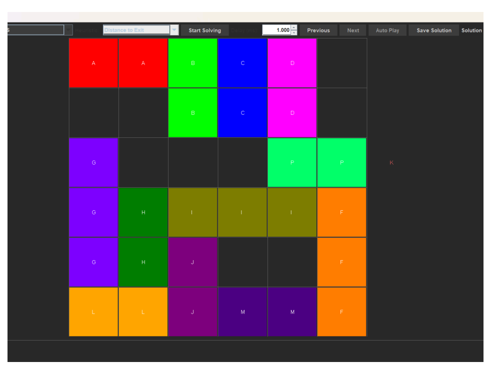
ExplorationStatsPanel.java

```
public class ExplorationStatsPanel extends JPanel {
    private JLabel timeLabel;
    private JLabel nodesLabel;
    private JLabel queueLabel;

    public ExplorationStatsPanel() {
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        setBackground(new Color(43, 43, 43));
        setBorder(BorderFactory.createTitledBorder(
                BorderFactory.createLineBorder(new Color(100, 100, 100)),
                "Exploration Statistics",
                javax.swing.border.TitledBorder.LEFT,
                javax.swing.border.TitledBorder.TOP,
                null,
                new Color(200, 200, 200)));

        // Initialize labels
        timeLabel = new JLabel("Time: -");
        nodesLabel = new JLabel("Nodes: -");
        queueLabel = new JLabel("Max Queue: -");

        // Style labels
        Color textColor = new Color(200, 200, 200);
        Font labelFont = new Font("Arial", Font.PLAIN, 12);
        styleLabel(timeLabel, textColor, labelFont);
        styleLabel(nodesLabel, textColor, labelFont);
        styleLabel(queueLabel, textColor, labelFont);

        // Add labels to panel
        add(Box.createVerticalStrut(5));
        add(timeLabel);
        add(Box.createVerticalStrut(5));
        add(nodesLabel);
        add(Box.createVerticalStrut(5));
```

ExplorationStatsPanel.java

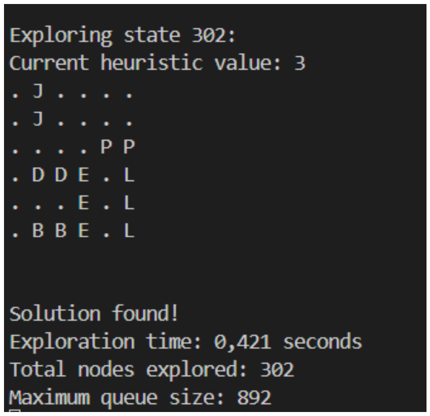
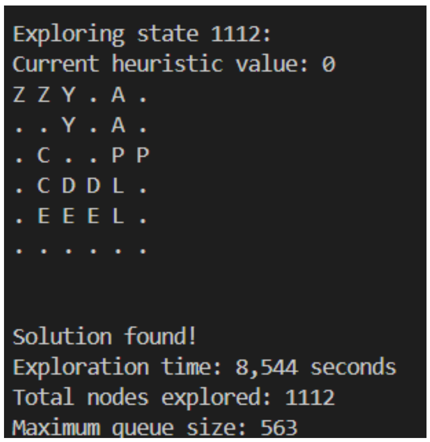
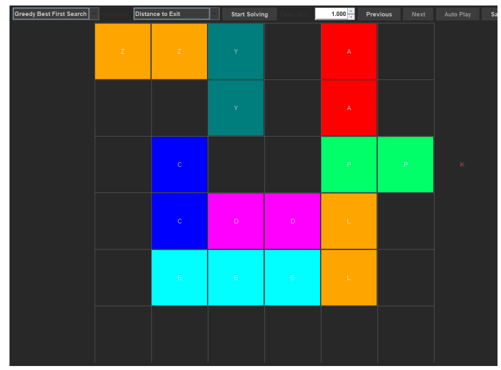
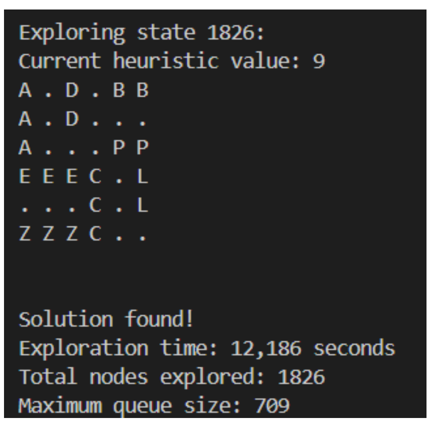
```
        add(queueLabel);
        add(Box.createVerticalStrut(5));
    }

    private void styleLabel(JLabel label, Color textColor, Font font) {
        label.setForeground(textColor);
        label.setFont(font);
        label.setAlignmentX(Component.LEFT_ALIGNMENT);
    }

    public void updateStats(long explorationTime, int nodesExplored, int
maxQueueSize) {
        timeLabel.setText(String.format("Time: %.3f seconds", explorationTime /
1000.0));
        nodesLabel.setText(String.format("Nodes: %d", nodesExplored));
        queueLabel.setText(String.format("Max Queue: %d", maxQueueSize));
    }

    public void reset() {
        timeLabel.setText("Time: -");
        nodesLabel.setText("Nodes: -");
        queueLabel.setText("Max Queue: -");
    }
}
```

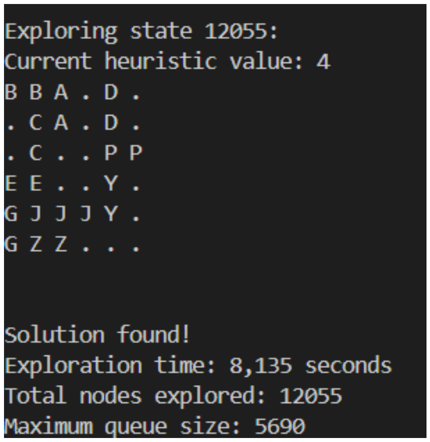
BAB 5: Pengujian

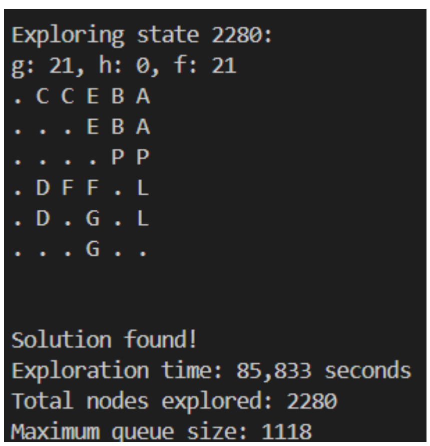
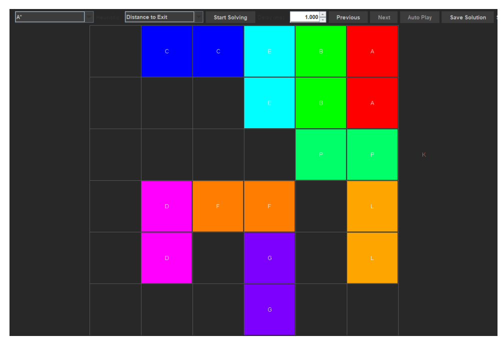
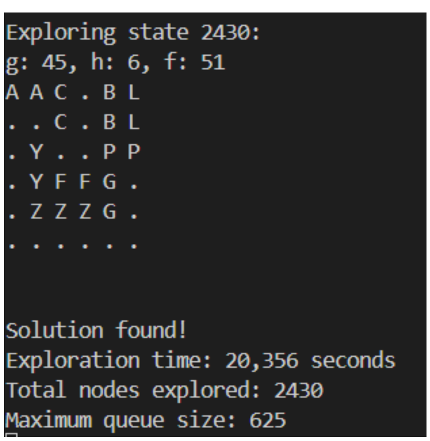
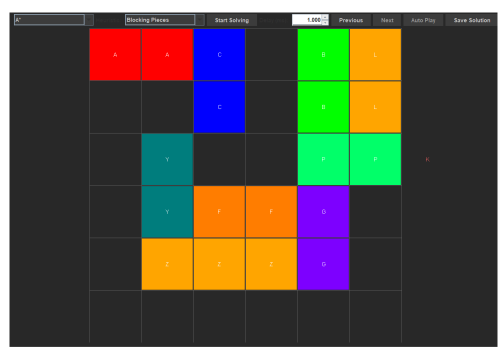
Uniform Cost Search			
No.	Testcase	Langkah2	Hasil
1.	6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Exploring state 1837: Current cost: 9 A A B C D . . . B C D . G . . . P P G H I I I F G H J . . F L L J M M F Solution found! Exploration time: 2,194 seconds Total nodes explored: 1837 Maximum queue size: 921	

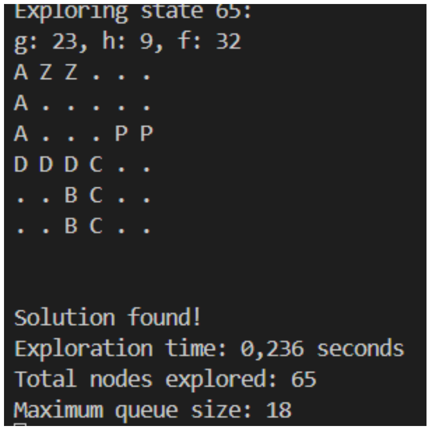
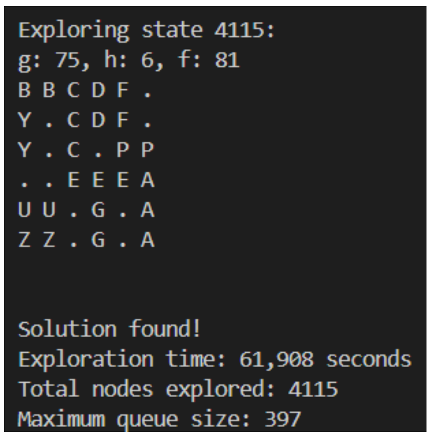
2.	6 6 13 .AABB. CCDDEF MNPPEFK MNJLEF MNJLGG .IIHH.	Exploring state 1592: Current cost: 32 A A J L B B C C J L D D M N . . P P M N . . E F M N G G E F I I H H E F Solution found! Exploration time: 10,678 seconds Total nodes explored: 1592 Maximum queue size: 210	
3.	6 6 10 AA.BBB ...EFF CPPE.ZK C.JGGZ DDJ..Z ..JLLL	Exploring state 16282: Current cost: 69 C A A B B B C . . F F . . . J . P P G G J E . Z D D J E . Z . . L L L Z Solution found! Exploration time: 88,669 seconds Total nodes explored: 16282 Maximum queue size: 1157	
4.	6 6 11 ..BCCC A.BDEE APPD..K .F.DGG HFII.J HLLL.J	Exploring state 21444: Current cost: 46 C C C . . J A E E . . J A . . . P P H F B D G G H F B D I I L L L D . . Solution found! Exploration time: 13,617 seconds Total nodes explored: 21444 Maximum queue size: 3036	

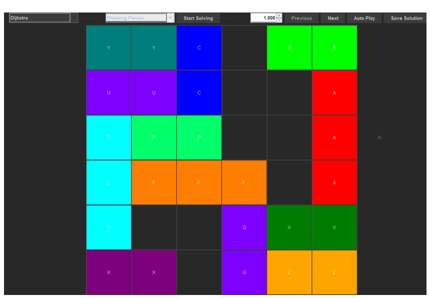
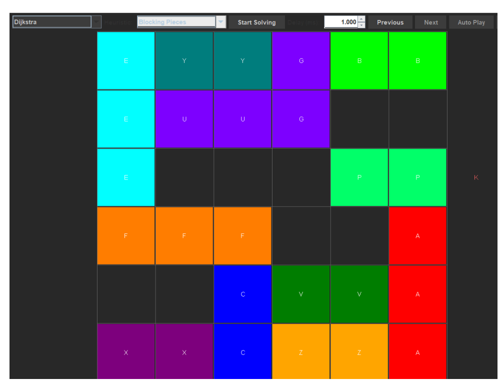
Greedy Best First Search			
No.	Testcase	Langkah2	Hasil

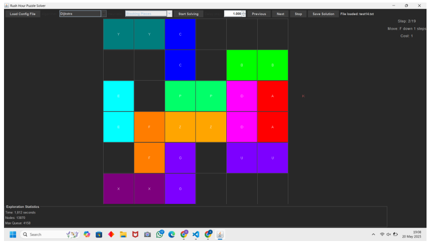
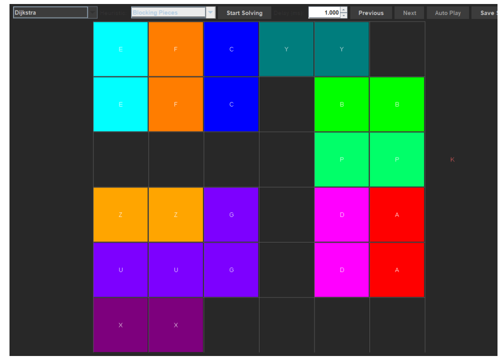
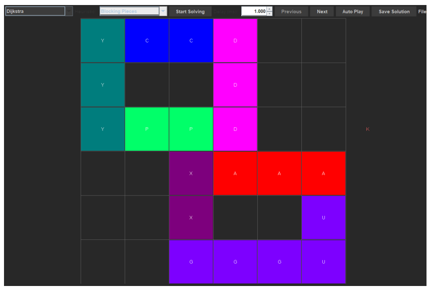
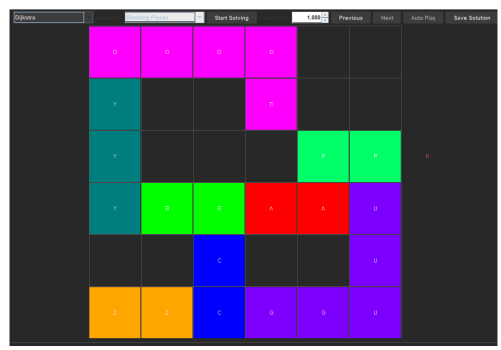
1.	6 6 5 .... .PPE..K .DDE.L .J.E.L .JBB.L	 Exploring state 302: Current heuristic value: 3 . J . . . . . J . . . . P P . D D E . L . . . E . L . B B E . L Solution found! Exploration time: 0,421 seconds Total nodes explored: 302 Maximum queue size: 892	
2.	6 6 7 ..YZZ. ..Y.A. .CPPA.K .CDDL. .EEEL.	 Exploring state 1112: Current heuristic value: 0 Z Z Y . A . . . Y . A . . C . . P P . C D D L . . E E E L . . . . Solution found! Exploration time: 8,544 seconds Total nodes explored: 1112 Maximum queue size: 563	
3.	6 6 7 ABBC.. A..C.. APPC..K ..DEEE ..D..L ..ZZZL	 Exploring state 1826: Current heuristic value: 9 A . D . B B A . D . . . A . . . P P E E E C . L . . . C . L Z Z Z C . . Solution found! Exploration time: 12,186 seconds Total nodes explored: 1826 Maximum queue size: 709	

4.	6 6 9 ..ABB. .CA... GCPPD.K GEE.D. JJJ.Y. ZZ..Y.	 Exploring state 12055: Current heuristic value: 4 B B A . D . . C A . D . . C . . P P E E . . Y . G J J J Y . G Z Z . . . Solution found! Exploration time: 8,135 seconds Total nodes explored: 12055 Maximum queue size: 5690	

A*			
No.	Testcase	Langkah2	Hasil
1.	6 6 8 .DCCBA .D.EBA .PPE.LK ..FF.L ...G.. ...G..	 Exploring state 2280: g: 21, h: 0, f: 21 . C C E B A . . . E B A . . . . P P . D F F . L . D . G . L . . . G . . Solution found! Exploration time: 85,833 seconds Total nodes explored: 2280 Maximum queue size: 1118	
2.	6 6 8 ..CAA. ..C.BL .YPPBLK .YFFG. .ZZZG.	 Exploring state 2430: g: 45, h: 6, f: 51 A A C . B L . . C . B L . Y . . P P . Y F F G . . Z Z Z G . . . . Solution found! Exploration time: 20,356 seconds Total nodes explored: 2430 Maximum queue size: 625	

<table>
<tr><td>3.</td><td>6 6
5
ZZBC..
A.BC..
APPC..K
ADDD..
......
......</td><td>
</td><td></td></tr>
<tr><td>4.</td><td>6 6
10
..CBBA
..CD.A
PPCD.AK
YEEE..
YUUGF.
ZZ.GF.</td><td>
</td><td></td></tr>
</table>

<table>
<tr><th colspan="4">Djikstra</th></tr>
<tr><th>No.</th><th>Testcase</th><th>Langkah2</th><th>Hasil</th></tr>
<tr><td>1.</td><td>6 6
11
YYC.BB
UUC..A
EPP..AK
EFFF.A
E..GVV
XX.GZZ</td><td></td><td></td></tr>
</table>

2.	6 6 11 YYC... ..C.BB EFPPDAK EFZZDA ..G.UU XXG...		
3.	6 6 7 YCCD.. Y..D.. YPPD..K ..XAAA ..X..U ..GGGU		
4.	6 6 8 Y.CDDD Y.CD.. YPPD..K BBAA.U .....U ZZGG.U		

BAB 6: Penutup

5.1. Kesimpulan

Pada Tugas Kecil 3 mata kuliah IF2211 Strategi Algoritma ini, kami berhasil mengembangkan aplikasi Solver untuk permainan Rush Hour .Program ini menggunakan algoritma pathfinding Uniform Cost Search (UCS), Greedy Best First Search, A*, dan Djikstra.

5.2. Saran

Saran untuk pengembangan selanjutnya, sebaiknya dapat dicoba juga untuk melakukan implementasi beberapa heuristics yang berbeda untuk mengoptimalkan performa algoritma pathfinding dalam menyelesaikan Rush Hour. Lalu, untuk pengujian sebaiknya membuat lebih banyak variasi pada papan dengan ukuran bervariasi agar hasil yang diberikan menjadi lebih baik.

5.3. Refleksi

Melalui proyek ini, kami memperoleh banyak pemahaman baru dan pemahaman mendalam mengenai implementasi algoritma pathfinding seperti UCs, Greedy Best First Search, A*, dan Djikstra. Proyek ini berhasil menunjukkan bagaimana strategi algoritma dapat diterapkan secara praktis dalam menyelesaikan permasalahan eksplorasi jalur dan kombinasi dalam permainan berbasis logika.

BAB 7: Lampiran

Link Penting

Repository	https://github.com/inRiza/Tucil3_13523133_13523164

Tabel Checkpoint

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

Munir, Rinaldi. "Route-Planing (2025) Bagian 1." Program Studi Teknik Informatika, STEI ITB. Diakses 19 Mei, 2025.informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

Munir, Rinaldi. "Route-Planing (2025) Bagian 2." Program Studi Teknik Informatika, STEI ITB. Diakses 19 Mei, 2025.informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf