

LAPORAN TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun oleh:

Muhammad Aufa Farabi 13523023

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025

DAFTAR ISI

DAFTAR ISI.....	1
BAGIAN I PENJELASAN ALGORITMA.....	2
BAGIAN II SOURCE PROGRAM.....	5
BAGIAN III HASIL TEST.....	21
BAGIAN IV ANALISIS.....	27
BAGIAN IV IMPLEMENTASI BONUS.....	27
LINK REPOSITORY.....	31
TABEL CHECKLIST.....	31

BAGIAN I

PENJELASAN ALGORITMA

Algoritma Greedy Best First Search, UCS, dan A* merupakan salah tiga algoritma yang digunakan untuk mencari rute dari *starting point* menuju *goals* (tujuan). Algoritma Greedy Best First Search dan A* merupakan algoritma *informed search*, yakni algoritma yang memiliki informasi pada *goal state* yang didapatkan dari fungsi $f(n)$ yang mengestimasi seberapa dekat *node* n terhadap *goal*. Informasi ini berupa dalam bentuk heuristik $h(n)$ yang misalnya menentukan estimasi *cost* dari n menuju *goal* berdasarkan *straight-line distance* atau jenis-jenis heuristik lain. Hal ini berbeda dengan algoritma UCS yang merupakan algoritma *uninformed search*, layaknya *BFS* dan *DFS*, yang tidak memiliki informasi tambahan seperti heuristik sebelumnya. Algoritma Greedy Best First pada dasarnya menentukan *path* “terbaik” secara lokal tanpa mempedulikan apakah *path* tersebut menghasilkan solusi optimum berdasarkan evaluasi biaya *node* saat ini menuju *goals* berdasarkan fungsi heuristik. Oleh karena itu, algoritma ini belum tentu optimal karena bersifat *greedy*. Hal ini berbeda dengan Algoritma UCS yang menentukan *path* menuju *goal* berdasarkan ekspansi node yang menghasilkan *path cost* dari *starting goal* menuju *node* paling minimal sehingga bersifat optimal. Algoritma A* sendiri dapat dikatakan sebagai gabungan Algoritma Greedy Best First Search dan UCS, yakni algoritma ini mengevaluasi *node* yang akan ditelusuri menuju *goal* berdasarkan estimasi *cost* dari n menuju *goal* ditambah *path cost* dari *starting point* menuju n ($f(n) = h(n) + g(n)$). Algoritma A* pasti menghasilkan solusi optimal apabila heuristik yang digunakan *admissible*, artinya fungsi heuristik $h(n)$ selalu meng-*underestimate* nilai minimum *cost* dari n menuju *goal* yang sebenarnya $h^*(n)$.

Ketiga algoritma *pathfinding* tersebut dapat digunakan untuk menyelesaikan puzzle permainan Rush Hour. Langkah-langkah dari penyelesaian puzzle permainan Rush Hour dengan algoritma GBFS, UCS, dan A* dapat dijabarkan seperti berikut:

1. Inisialisasi state awal dari permainan puzzle dengan state awal menjadi root node. Di sini, $f(n)$ sama dengan $g(n)$ sebagai *path cost* yang diset bernilai 0
2. Jika menggunakan Algoritma GBFS/ A*, hitung nilai heuristik $h(n)$ dari state awal kemudian nilai $h(n)$ tersebut ditambahkan ke fungsi *cost* $f(n)$
3. Masukkan state awal sebagai node ke dalam priority queue berdasarkan fungsi *cost* $f(n)$
4. Pop node dari priority queue dan catat node ke dalam set yang berisi visited node
5. Cek apakah state tersebut merupakan node tujuan, dalam konteks puzzle rush hour berarti state yang *primary piece* nya sudah berada di posisi pintu keluar, jika iya maka program selesai
6. Lakukan ekspansi setiap kemungkinan gerakan valid dari setiap piece kemudian cek apakah pergerakan tersebut menghasilkan suatu state (calon node) yang belum dikunjungi (tidak ada di set visited)

7. Jika iya, maka state tersebut dijadikan sebagai node dengan parent node nya adalah state sebelumnya yang dilakukan ekspansi. Nilai cost $g(n)$ pada state sekarang merupakan cost $g(n)$ lama + 1.
8. Apabila menggunakan algoritma GBFS/A*, hitung nilai heuristik $h(n)$ dari node tersebut
9. Masukkan node ke dalam priority queue yang diurutkan berdasarkan perhitungan fungsi evaluasi $f(n)$, dengan ketentuan $f(n)$ untuk algoritma GBFS adalah $h(n)$, algoritma UCS adalah $g(n)$, dan algoritma A* adalah $h(n) + g(n)$
10. Lakukan langkah 4-9 hingga priority queue kosong. Jika priority queue kosong tetapi belum ditemukan node tujuan, maka puzzle dinyatakan tidak memiliki solusi.

Dari langkah-langkah tersebut untuk penyelesaian permainan puzzle Rush Hour, dapat dianalisis bahwa algoritma Greedy Best First Search (GBFS) menggunakan pendekatan yang hanya mengevaluasi nilai $f(n)$ berdasarkan perkiraan cost yang dibutuhkan dari state saat ini menuju *goal node* berdasarkan $h(n)$, yakni fungsi heuristik. Algoritma ini memiliki kelebihan pada waktu pencarian yang lebih singkat, tetapi *irrevocable*, belum tentu optimal atau bahkan dapat terjebak dalam *local minima* karena bersifat *greedy*. Oleh karena itu, heuristik yang dipakai sangat berpengaruh pada hasil dari algoritma GBFS. Lain halnya dengan algoritma sebelumnya, algoritma UCS menyimpan *cost* yang didapat dari state awal hingga state sekarang $g(n)$ dan menentukan node tetangga yang ditinjau berdasarkan perhitungan $g(n)$ tersebut sehingga dijamin mendapat solusi optimal. Namun, kekurangan dari algoritma UCS adalah waktu pencarian yang lebih lama dibandingkan algoritma-algoritma lainnya. Algoritma UCS sekilas mirip dengan Algoritma BFS karena sama-sama *uninformed search* dan menghasilkan solusi optimal, bedanya adalah algoritma BFS menghasilkan solusi optimal ketika $cost = step$ sehingga apabila $cost$ menuju node tetangga yang ditinjau tidak sama nilainya dengan $step$ maka tidak akan menghasilkan solusi optimal. Dalam kata lain, algoritma BFS tidak akan menghasilkan hasil path dan urutan node yang dibangkitkan yang sama dengan algoritma UCS dalam kasus tersebut.

Algoritma ketiga pada program penyelesaian puzzle Rush Hour, yakni algoritma A*, memakai gabungan pendekatan algoritma GBFS dengan $f(n) = h(n)$ serta algoritma UCS dengan $f(n) = g(n)$ sehingga *node* tetangga yang ditelusuri pada algoritma ini ditentukan berdasarkan evaluasi $f(n) = g(n) + h(n)$. Algoritma A* mengambil keuntungan dari dua algoritma sebelumnya di mana waktu pencarian yang dilakukan algoritma A* cukup singkat dan juga menghasilkan solusi optimal dengan syarat heuristik yang dipakai untuk mengevaluasi $h(n)$ bersifat *admissible*. Pada program yang dibuat, terdapat empat heuristik yang dipakai pada algoritma GBFS dan A*, yakni jarak sel antara *primary piece* dengan pintu keluar (1), jumlah *pieces* yang memblokir jalur *primary piece* menuju pintu keluar (2), gabungan heuristik 1 dan 2 (3), dan jumlah *blocking pieces* yang dapat digeser. Di sini, heuristik kedua dan heuristik keempat *admissible* karena jarak minimum yang perlu ditempuh *primary piece* sebanding dengan banyak *piece* yang memblokir jalurnya dengan heuristik keempat hanyalah variasi dari heuristik pertama. Dalam arti lain, kedua heuristik ini meng-*underestimate* cost sebenarnya $h^*(n)$ untuk menuju *goal node* sehingga memenuhi syarat *admissible* suatu heuristik, yakni $h(n) \leq h^*(n)$. Heuristik pertama tidak memenuhi kriteria *admissible* untuk puzzle Rush Hour. Ini karena heuristik tersebut tidak memperhitungkan banyak *piece* yang memblokir jalur keluar *primary piece*, contohnya ketika jarak *primary piece* ke pintu keluar hanya tiga cell, tetapi satu

cell nya ditempati masing-masing oleh suatu *piece*, heuristik ini akan memberikan nilai cost $h(n)$ sebesar 3 yang lebih besar dari cost sebenarnya $h^*(n)$ sehingga heuristik meng-*overestimate* cost path dari state saat ini menuju *goal node*, yang melanggar syarat *admissible*. Heuristik ketiga juga memakai heuristik pertama sehingga tidak dapat dikatakan *admissible*.

Pada program penyelesaian puzzle Rush Hour, diimplementasikan juga algoritma Iterative Deepening Search (IDS). Algoritma ini adalah algoritma *pathfinding* yang mengombinasikan algoritma BFS dan DFS, yakni melakukan DFS untuk setiap batas kedalaman yang ditempuh. Apabila pada suatu kedalaman X belum ditemui *goal node*, maka batas kedalaman ditambah dan dilakukan DFS hingga mencapai batas kedalaman yang baru. Proses algoritma IDS dalam penyelesaian puzzle Rush Hour dapat dijabarkan sebagai berikut:

1. Inisialisasi *depth* dari 0
2. Cek apakah sudah sampai *goal node*
3. Jika belum, naikkan batas *depth* sebanyak satu
4. Lakukan pencarian dengan menelusuri secara DFS dari root node ke node anaknya yang masih dalam batas *depth*
5. Cek kembali apakah sudah sampai *goal node*, jika belum telusuri node hingga mencapai batas *depth*. Jika *depth* sudah mencapai batas *depth*, *backtrack* ke node lainnya yang merupakan *children* dari *parent node*.
6. Lakukan kembali langkah 3-5
7. Jika sudah mencapai *maxDepth* yang ditentukan dan belum ditemukan solusi maka solusi dinyatakan tidak ada.

BAGIAN II

SOURCE PROGRAM

Program utama ditulis dalam bahasa Java dengan memakai *packages* di antaranya:

1. Java.util
2. java.io
3. Javax.swing (GUI)
4. Javax.awt (GUI)

Program ini terdiri atas 8 *files* yang bersesuaian dengan kelas nya, 1 *files* untuk *interface* dan 1 *file* sebagai GUI atau Main

- | | |
|---|---|
| - Piece | - MinBlock (<i>class Heuristic</i>) |
| - Board | - DistanceToExit(<i>class Heuristic</i>) |
| - State | - MoveablePieces (<i>class Heuristic</i>) |
| - Solver | - InputOutput |
| - Heuristic (Interface untuk heuristic) | - Main (GUI) |

Berikut adalah snapshot source program yang berkaitan dengan implementasi algoritma

Interface Heuristic

```
package src;

public interface Heuristic {
    public int calculateHeuristic(State state);
}
```

Kelas DistanceToExit

```
package src;

public class DistanceToExit implements Heuristic {

    public int calculateHeuristic(State state) {
        return state.calculateDistanceToExit();
    }
}
```

Kelas MinBlock

```
package src;

public class MinBlock implements Heuristic {
    private boolean manhattanDistance;

    public MinBlock(boolean manhattanDistance) {
        this.manhattanDistance = manhattanDistance;
    }

    public int calculateHeuristic(State state) {
        Piece primaryPiece = state.getPieces().get(key: 'P');
        char[][] board = state.getStateBoard().getBoard();
        int rows = state.getStateBoard().getRows();
        int cols = state.getStateBoard().getCols();
        int exitY = state.getStateBoard().getExitY();
        int exitX = state.getStateBoard().getExitX();
        int exitDistance = 0;

        if (manhattanDistance) {
            exitDistance = state.calculateDistanceToExit();
            System.out.println("Manhattan distance: " + exitDistance);
        }
    }
}
```

```
int blockingPieces = 0;
if (exitX == 0) {
    int row = primaryPiece.getPieceRow();
    for (int col = 1; col < primaryPiece.getPieceCol(); col++) {
        if (col < state.getStateBoard().getCols() && state.getStateBoard().getBoard()[row][col] != '.'
            && state.getStateBoard().getBoard()[row][col] != 'K') {
            blockingPieces++;
        }
    }
} else if (exitX == cols - 1) {
    int row = primaryPiece.getPieceRow();
    for (int col = cols - 2; col >= primaryPiece.getPieceCol() + primaryPiece.getPieceSize(); col--) {
        if (col > 0 && board[row][col] != '.'
            && board[row][col] != 'K') {
            blockingPieces++;
        }
    }
}
```

```

    } else if (exitY == 0) {
        int col = primaryPiece.getPieceCol();
        for (int row = 1; row < primaryPiece.getPieceRow(); row++) {
            if (row < rows && board[row][col] != '.'
                && board[row][col] != 'K') {
                blockingPieces++;
            }
        }
    } else if (exitY == rows - 1) {
        int col = primaryPiece.getPieceCol();
        for (int row = rows - 2; row >= primaryPiece.getPieceRow() + primaryPiece.getPieceSize(); row--) {
            if (row > 0 && board[row][col] != '.'
                && board[row][col] != 'K') {
                blockingPieces++;
            }
        }
    }
    return exitDistance + blockingPieces;
}
}

```

Kelas MoveablePieces

```

package src;

import java.util.Map;

public class MoveablePieces implements Heuristic {
    public int calculateHeuristic(State state) {
        int exitDistance = state.calculateDistanceToExit();
        int blockingMoves = 0;
        for (Map.Entry<Character, Piece> entry : state.getPieces().entrySet()) {
            Piece piece = entry.getValue();
            if (piece.getPieceID() == 'P') continue;
            if (state.isBlockingExit(piece)) {
                if (piece.isHorizontal()) {
                    if (piece.canMove(state, moveRow:0, moveCol:1) || piece.canMove(state, moveRow:0, -1)) {
                        blockingMoves++;
                    } else {
                        blockingMoves += 2;
                    }
                } else {
                    if (piece.canMove(state, moveRow:1, moveCol:0) || piece.canMove(state, -1, moveCol:0)) {
                        blockingMoves++;
                    } else {
                        blockingMoves += 2;
                    }
                }
            }
        }
        return exitDistance + blockingMoves;
    }
}

```

Kelas Solver


```

package src;
import java.util.*;
import java.util.function.Consumer;
import java.util.function.Supplier;

public class Solver {
    private List<State> solutionPath;

    public List<State> getSolutionPath() {
        return solutionPath;
    }
}

```

```

    public Heuristic setHeuristic(int heuristicType) {
        switch (heuristicType) {
            case 1:
                return new DistanceToExit();
            case 2:
                return new MinBlock(isDistanceHeuristic:false);
            case 3:
                return new MinBlock(isDistanceHeuristic:true);
            case 4:
                return new MoveablePieces();
            default:
                return null;
        }
    }
}

```

```

    public boolean solveWithIDS(State initialState) {
        int depthLimit = 0;
        int MAX_DEPTH = 1000; // kasus unsolvable
        this.solutionPath = null;
        while (depthLimit < MAX_DEPTH) {
            Set<String> visited = new HashSet<>();

            State resultState = depthLimitedSearch(initialState, depthLimit, visited);
            if (resultState != null) {
                this.solutionPath = reconstructPath(resultState);
                return true;
            }
            depthLimit++;
        }
        return false;
    }

    private State depthLimitedSearch(State currentState, int depthLimit, Set<String> visited) {
        if (currentState.isGoal(currentState.getStateBoard().getExitY(), currentState.getStateBoard().getExitX())) {
            return currentState;
        }
        visited.add(currentState.getUniqueStateID());
    }
}

```

```

        for (State nextState : currentState.generateSuccessors()) {
            if (!visited.contains((nextState.getUniqueStateID()))) {
                nextState.setParent(currentState);
                State resultState = depthLimitedSearch(nextState, depthLimit - 1, visited);
                if (resultState != null) {
                    return resultState;
                }
            }
        }
        return null;
    }

    public boolean solveWithGreedyBFS(int heuristicType, State initialState) {
        Heuristic heuristic = setHeuristic(heuristicType);
        if (heuristic == null) {
            return false;
        }
        return solveHelper(Comparator.comparingInt(s -> heuristic.calculateHeuristic(s)), initialState, isGBFS:true);
    }
}

```

```

    public boolean solveWithUCS(State initialState) {
        return solveHelper(Comparator.comparingInt(s -> s.getCost()), initialState, isGBFS:false);
    }

    public boolean solveWithAStar(int heuristicType, State initialState) {
        Heuristic heuristic = setHeuristic(heuristicType);
        if (heuristic == null) {
            return false;
        }
        return solveHelper(Comparator.comparingInt(s -> s.getCost() + heuristic.calculateHeuristic(s)), initialState, isGBFS:false);
    }

    private boolean solveHelper(Comparator<State> comparator, State initialState, boolean isGBFS) {
        PriorityQueue<State> frontier = new PriorityQueue<>(comparator);
        Set<String> visited = new HashSet<>();

        frontier.add(initialState);
        if (isGBFS) {
            visited.add(initialState.getUniqueStateID());
        }

        State goalState = null;
    }

```

```

    this.solutionPath = null;
    while (!frontier.isEmpty()) {
        State currentState = frontier.poll();
        if (!isGBFS) {
            visited.add(currentState.getUniqueStateID());
        }

        if (currentState.isGoal(currentState.getStateBoard().getExitY(), currentState.getStateBoard().getExitX()) {
            goalState = currentState;
            break;
        }

        List<State> successors = currentState.generateSuccessors();
        for (State nextState : successors) {
            if (!visited.contains((nextState.getUniqueStateID()))) {
                if (isGBFS) {
                    visited.add(nextState.getUniqueStateID());
                }
            }
        }
    }

```

```

        nextState.setParent(currentState);
        frontier.add(nextState);
    }
}

if (goalState != null) {
    this.solutionPath = reconstructPath(goalState);
    return true;
} else {
    return false;
}
}

```

```

private List<State> reconstructPath(State goalState) {
    // System.out.println();
    LinkedList<State> path = new LinkedList<>();
    State currentState = goalState;
    while (currentState != null) {
        path.addFirst(currentState);
        currentState = currentState.getParent();
    }
    return path;
}

```

```

public void animateSolution(Consumer<char[][]> drawCallback, Supplier<Integer> delayEffect) {
    if (solutionPath == null || solutionPath.isEmpty()) return;

    for (State state : solutionPath) {
        if (Thread.currentThread().isInterrupted()) {
            return;
        }
        char[][] board = state.getStateBoard().getBoard();
        drawCallback.accept(board);
        try {
            Thread.sleep(delayEffect.get());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            break;
        }
    }
}

```

Kelas Piece

```
package src;
import java.util.ArrayList;
import java.util.List;

public class Piece {
    private char pieceID;
    private int row, col;
    private int size;
    public boolean isHorizontal;

    public Piece(char id, int row, int col, int size, boolean isHorizontal) {
        this.pieceID = id;
        this.row = row;
        this.col = col;
        this.size = size;
        this.isHorizontal = isHorizontal;
    }

    public char getPieceID() {
        return pieceID;
    }
    public int getPieceRow() {
        return row;
    }
    public int getPieceCol() {
        return col;
    }
    public int getPieceSize() {
        return size;
    }

    public Piece copy() {
        return new Piece(pieceID, row, col, size, isHorizontal);
    }

    public List<int[]> getOccupiedCells() {
        List<int[]> cells = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            if (isHorizontal) {
                cells.add(new int[]{row, col + i});
            } else {
                cells.add(new int[]{row + i, col});
            }
        }
        return cells;
    }

    public Piece move(int moveRow, int moveCol) {
        return new Piece(pieceID, row + moveRow, col + moveCol, size, isHorizontal);
    }
}
```

```

public boolean canMove(State state, int moveRow, int moveCol) {
    int rows = state.getStateBoard().getRows();
    int cols = state.getStateBoard().getCols();
    List<int[]> occupiedCells = getOccupiedCells();
    char[][] board = state.getStateBoard().getBoard();

    for (int[] cell : occupiedCells) {
        int newRow = cell[0] + moveRow;
        int newCol = cell[1] + moveCol;

        if (newRow < 0 || newRow >= rows
            || newCol < 0 || newCol >= cols) {
            return false;
        }

        boolean isOriginalCell = false;
        for (int[] originalCell : occupiedCells) {
            if (newRow == originalCell[0] && newCol == originalCell[1]) {
                isOriginalCell = true;
                break;
            }
        }

        char destCell = board[newRow][newCol];
        if (!isOriginalCell && destCell != '.' && destCell != 'K') {
            return false;
        }
    }

    return true;
}

```

```

package src;

public class Board {
    private char[][] board;
    private int exitY, exitX;
    private int rows, cols;

    public Board(char[][] board, int exitY, int exitX) {
        this.rows = board.length;
        this.cols = board[0].length;
        this.exitY = exitY;
        this.exitX = exitX;
        this.board = new char[board.length][board[0].length];
        for (int i = 0; i < board.length; i++) {
            this.board[i] = board[i].clone();
        }
    }

    public char[][] getBoard() {
        return board;
    }

    public int getExitY() {
        return exitY;
    }

    public int getExitX() {
        return exitX;
    }

    public int getRows() {
        return rows;
    }

    public int getCols() {
        return cols;
    }
}

```

```

public Board updateBoard(Piece original, Piece moved) {
    char[][] newBoard = new char[this.rows][this.cols];
    for (int i = 0; i < this.rows; i++) {
        newBoard[i] = board[i].clone();
    }
    for (int[] cell : original.getOccupiedCells()) {
        newBoard[cell[0]][cell[1]] = '.';
    }

    for (int[] cell : moved.getOccupiedCells()) {
        newBoard[cell[0]][cell[1]] = moved.getPieceID();
    }

    return new Board(newBoard, exitY, exitX);
}

```

Kelas State

```

package src;
import java.util.*;

public class State {
    private Board board;
    private Map<Character, Piece> pieces;
    private State parent;
    private List<String> moveLog;
    private int cost;
    private String uniqueID = null;

    // CTOR awal
    public State(char[][] board, Map<Character, Piece> pieces, int exitY, int exitX) {
        this.board = new Board(board, exitY, exitX);
        this.pieces = new HashMap<>();
        for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
            Piece p = entry.getValue();
            this.pieces.put(entry.getKey(), new Piece(p.getPieceID(), p.getPieceRow(), p.getPieceCol(), p.getPieceSize(), p.isHorizontal));
        }
        this.parent = null;
        this.cost = 0;
        this.moveLog = new ArrayList<>();
    }

```

```

    // CTOR buat successor state
    public State(Board stateBoard, Map<Character, Piece> pieces, State parent, int cost, List<String> moveLog) {
        this.board = stateBoard;
        this.pieces = new HashMap<>();
        for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
            Piece p = entry.getValue();
            this.pieces.put(entry.getKey(), new Piece(p.getPieceID(), p.getPieceRow(), p.getPieceCol(), p.getPieceSize(), p.isHorizontal));
        }
        this.parent = parent;
        this.cost = cost;
        this.moveLog = moveLog;
    }

    public State getParent() {
        return parent;
    }

    public void setParent(State newParent) {
        this.parent = newParent;
    }

    public int getCost() {
        return cost;
    }
}

```

```

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

    public Board getStateBoard() {
        return board;
    }

    public List<String> getMoveLog() {
        return moveLog;
    }

    public String getUniqueStateID() {
        if (uniqueID == null) {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < board.getRows(); i++) {
                for (int j = 0; j < board.getCols(); j++) {
                    sb.append(board.getBoard()[i][j]);
                }
            }
            uniqueID = sb.toString();
        }
        return uniqueID;
    }
}

```

```

public int calculateDistanceToExit() {
    Piece primaryPiece = pieces.get(key:'P');
    if (board.getExitX() == 0) {
        return primaryPiece.getPieceCol();
    } else if (board.getExitX() == board.getCols() - 1) {
        return board.getCols() - 1 - (primaryPiece.getPieceCol() + primaryPiece.getPieceSize() - 1);
    } else if (board.getExitY() == 0) {
        return primaryPiece.getPieceRow();
    } else if (board.getExitY() == board.getRows() - 1) {
        return board.getRows() - 1 - (primaryPiece.getPieceRow() + primaryPiece.getPieceSize() - 1);
    }
    return 0;
}

public boolean isBlockingExit(Piece piece) {
    Piece primaryPiece = getPieces().get(key:'P');
    if (board.getExitX() == 0) {
        int row = primaryPiece.getPieceRow();
        for (int col = 1; col < primaryPiece.getPieceCol(); col++) {
            if (board.getBoard()[row][col] == piece.getPieceID()) {
                return true;
            }
        }
    }
}

```

```

    }
} else if (board.getExitX() == board.getCols() - 1) {
    int row = primaryPiece.getPieceRow();
    for (int col = board.getCols() - 2; col >= primaryPiece.getPieceCol() + primaryPiece.getPieceSize(); col--) {
        if (board.getBoard()[row][col] == piece.getPieceID()) {
            return true;
        }
    }
} else if (board.getExitY() == 0) {
    int col = primaryPiece.getPieceCol();
    for (int row = 1; row < primaryPiece.getPieceRow(); row++) {
        if (board.getBoard()[row][col] == piece.getPieceID()) {
            return true;
        }
    }
} else if (board.getExitY() == board.getRows() - 1) {
    int col = primaryPiece.getPieceCol();
    for (int row = board.getRows() - 2; row >= primaryPiece.getPieceRow() + primaryPiece.getPieceSize(); row--) {
        if (board.getBoard()[row][col] == piece.getPieceID()) {
            return true;
        }
    }
}
}

```



```

    }
}
return false;
}

public List<State> generateSuccessors() {
    List<State> successors = new ArrayList<>();
    for (Piece piece : pieces.values()) {
        int[] PossibleMovePoints = {-1, 1};
        for (int possibleMovePoint : PossibleMovePoints) {
            int step = 1;
            while (true) {
                int moveRow = 0;
                int moveCol = 0;
                if (piece.isHorizontal()) {
                    moveCol = possibleMovePoint * step;
                } else {
                    moveRow = possibleMovePoint * step;
                }

                if (!piece.canMove(this, moveRow, moveCol)) {
                    break;
                }
                Piece movedPiece = piece.move(moveRow, moveCol);
                Map<Character, Piece> nextPieces = new HashMap<>(this.pieces);
                nextPieces.put(piece.getPieceID(), movedPiece);
                Board nextStateBoard = board.updateBoard(piece, movedPiece);
                List<String> newMoveLog = new ArrayList<>(this.moveLog);
                String direction = "";
                if (possibleMovePoint < 0) {
                    if (piece.isHorizontal()) {
                        direction = "kiri";
                    } else {
                        direction = "atas";
                    }
                }
            }
        }
    }
}

```

```

                } else if (possibleMovePoint > 0) {
                    if (piece.isHorizontal()) {
                        direction = "kanan";
                    } else {
                        direction = "bawah";
                    }
                }
            }
            newMoveLog.add(piece.getPieceID() + "-" + direction);
            successors.add(new State(nextStateBoard, nextPieces, this, this.cost + 1, newMoveLog));
            step++;
        }
    }
}
return successors;
}

public boolean isGoal(int exitY, int exitX) {
    Piece primaryPiece = pieces.get(key: 'P');
    if (primaryPiece == null) {
        return false;
    }

    for(int[] cell : primaryPiece.getOccupiedCells()) {
        if (cell[0] == exitY && cell[1] == exitX) {
            return true;
        }
    }
    return false;
}
}

```

Kelas InputOutput

```
package src;

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

public class InputOutput {
    private int boardWidth, boardHeight;
    private int superBoardWidth, superBoardHeight;
    private int commonPieceCount;
    private char[][] initialBoard;
    private State initialState;
    private int exitY, exitX;

    public char[][] getInitialBoard() {
        return initialBoard;
    }
    public State getInitialState() {
        return initialState;
    }

    private void copyRowtoInitialBoard(String row, int targetRow, boolean isExitLeft) throws IOException {
        if (row.length() > boardWidth + 1 || row.length() < boardWidth) {
            throw new IOException(message:"Konfigurasi papan (panjang baris) tidak valid.");
        }
        for (int j = 0; j <= this.boardWidth; j++) {
            if (j == this.boardWidth && !isExitLeft) {
                continue;
            }
            char ch = row.charAt(j);
            if (ch != 'K' && ch != ' ') {
                if (isExitLeft) {
                    initialBoard[targetRow][j] = ch;
                } else {
                    initialBoard[targetRow][j+1] = ch;
                }
            } else if (ch == 'K' && j > 0 && j < boardWidth) {
                throw new IOException(message:"Konfigurasi papan (terdapat pintu keluar dalam papan) tidak valid.");
            }
        }
    }
}
```

```

public void setupSolver(String filepath) throws IOException {
    readInputFile(filepath);
    this.initialState = parsePieces();
}

private void readInputFile(String filePath) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
        String[] sizeLine = reader.readLine().split(regex: " ");
        try {
            this.boardHeight = Integer.parseInt(sizeLine[0]);
            this.superBoardHeight = this.boardHeight + 2;
            this.boardWidth = Integer.parseInt(sizeLine[1]);
            this.superBoardWidth = this.boardWidth + 2;
            this.initialBoard = new char[this.superBoardHeight][this.superBoardWidth];
            if (this.boardWidth <= 0 || this.boardHeight <= 0) {
                throw new IOException(message:"Dimensi papan (width dan height) harus positif.");
            }
        } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
            throw new IOException(message:"Dimensi papan tidak valid.");
        }

        try {
            this.commonPieceCount = Integer.parseInt(reader.readLine());
            if (this.commonPieceCount < 0) {
                throw new IOException(message:"Jumlah non-primary pieces tidak boleh 0 atau negatif.");
            }
        } catch (NumberFormatException e) {
            throw new IOException(message:"Format jumlah non-primary pieces tidak valid.");
        }

        for (int i = 0; i < this.superBoardHeight; i++) {
            for (int j = 0; j < this.superBoardWidth; j++) {
                if (i == 0 || i == this.superBoardHeight - 1 || j == 0 || j == this.superBoardWidth - 1) {
                    this.initialBoard[i][j] = '#';
                } else {
                    this.initialBoard[i][j] = '.';
                }
            }
        }
    }
}

```

```

List<String> boardLines = new ArrayList<>();
String line;
while ((line = reader.readLine()) != null) {
    if (!line.isBlank()) {
        boardLines.add(line);
    }
}

if (boardLines.size() < this.boardHeight) {
    throw new IOException(message:"Konfigurasi papan (kolom) tidak valid");
}

if (boardLines.get(index:0).contains(s:"K")) { // pintu keluar di atas
    String topString = boardLines.get(index:0);
    for (int j = 0; j < topString.length(); j++) {
        if (topString.charAt(j) != 'K' && topString.charAt(j) != ' ') {
            throw new IOException(message:"Konfigurasi papan (baris tidak sesuai) tidak valid.");
        }
        if (topString.charAt(j) == 'K') {
            initialBoard[0][j+1] = 'K';
            exitY = 0;
            exitX = j+1;
        }
        // printBoard(initialBoard);
    }
    for (int i = 0; i < boardHeight; i++) {
        copyRowtoInitialBoard(boardLines.get(i+1), i+1, isExitLeft:false);
    }
} else if (boardLines.size() == this.boardHeight + 1 &&
boardLines.get(boardLines.size() - 1).contains(s:"K")) { // pintu keluar di bawah
    String bottomString = boardLines.get(boardLines.size() - 1);
    for (int j = 0; j < bottomString.length(); j++) {
        if (bottomString.charAt(j) != 'K' && bottomString.charAt(j) != ' ') {
            throw new IOException(message:"Konfigurasi papan (baris bawah tidak sesuai) tidak valid.");
        }
    }
}

```

```

        if (bottomString.charAt(j) == 'K') {
            initialBoard[superBoardHeight - 1][j + 1] = 'K';
            exitY = superBoardHeight - 1;
            exitX = j + 1;
        }
    }

    for (int i = 0; i < boardHeight; i++) {
        copyRowToInitialBoard(boardLines.get(i), i + 1, isExitLeft:false);
    }

} else {
    boolean isExitLeft = false;

    for (int i = 0; i < this.boardHeight; i++) {
        String row = boardLines.get(i);
        if (row.length() > 0 && row.charAt(index:0) == 'K') { // pintu keluar di kiri
            initialBoard[i+1][0] = 'K';
            isExitLeft = true;
            // printBoard(initialBoard);
            // System.out.println();
            exitY = i + 1;
            exitX = 0;
        } else if (row.length() == boardWidth + 1 && row.charAt(boardWidth) == 'K') { // pintu keluar di kanan
            initialBoard[i + 1][superBoardWidth - 1] = 'K';
            exitY = i + 1;
            exitX = superBoardWidth - 1;
        }
    }

    for (int i = 0; i < this.boardHeight; i++) {
        if ((exitX == superBoardWidth - 1 && i + 1 != exitY && boardLines.get(i).length() > boardWidth
            && !boardLines.get(i).contains(s:"K"))
            || (exitX == 0 && i + 1 != exitY && boardLines.get(i).length() > boardWidth
            && !boardLines.get(i).contains(s:"K") && !boardLines.get(i).contains(s:" "))) {
            // System.out.println("i: " + i + ", exitY: " + exitY);
            // System.out.println("length: " + boardLines.get(i).length() + ", boardWidth: " + boardWidth);
            throw new IOException(message:"Konfigurasi papan (panjang baris berlebih) tidak valid.");
        }
    }
}

```

```

    }
    copyRowToInitialBoard(boardLines.get(i), i + 1, isExitLeft);
}
}
}

private State parsePieces() {
    Map<Character, Piece> piecesMap = new HashMap<>();
    boolean[][] visited = new boolean[superBoardHeight][superBoardWidth];
    for (int i = 0; i < superBoardHeight; i++) {
        for (int j = 0; j < superBoardWidth; j++) {
            char ch = initialBoard[i][j];
            if (ch != '.' && ch != '#' && ch != 'K' && !visited[i][j]) {
                int size = 0;

                // cek orientasi horizontal
                if (j + 1 < superBoardWidth && initialBoard[i][j + 1] == ch) {
                    int k = j;
                    while (k < superBoardWidth && initialBoard[i][k] == ch) {
                        visited[i][k] = true;
                        size++;
                        k++;
                    }
                    if (size < 2 || size > 3) {
                        throw new IllegalArgumentException(s:"Panjang piece tidak valid.");
                    }
                    piecesMap.put(ch, new Piece(ch, i, j, size, isHorizontal:true)); //cell piece paling kiri
                }
                // cek orientasi vertikal
                else if (i + 1 < superBoardHeight && initialBoard[i + 1][j] == ch) {
                    int k = i;
                    while (k < superBoardHeight && initialBoard[k][j] == ch) {
                        visited[k][j] = true;
                        size++;
                        k++;
                    }
                }
            }
        }
    }

    if (size < 2 || size > 3) {
        throw new IllegalArgumentException(s:"Panjang piece tidak valid.");
    }
    piecesMap.put(ch, new Piece(ch, i, j, size, isHorizontal:false)); //cell piece paling atas
}

return new State(initialBoard, piecesMap, exitY, exitX);
}

public void saveSolutionToFile(String filepath, List<State> solutionPath, String algo, int heuristicType, long duration) throws IOException {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(filepath))) {
        writer.write("Algoritma: " + algo + "\n");
        if (heuristicType == 1) {
            writer.write(str:"Heuristik: Distance To Exit\n");
        } else if (heuristicType == 2) {
            writer.write(str:"Heuristik: Min Blocking Pieces\n");
        } else if (heuristicType == 3) {
            writer.write(str:"Heuristik: Distance To Exit + Min Blocking Pieces\n");
        } else if (heuristicType == 4) {
            writer.write(str:"Heuristik: Min Moveable Blockers\n");
        }
        if (solutionPath == null) {
            writer.write(str:"Solusi tidak ditemukan.\n");
            return;
        }
        for (int i = 0; i < solutionPath.size(); i++) {
            State currentState = solutionPath.get(i);
            if (i == 0) {
                writer.write(str:"Papan awal:\n");
            } else {
                if (!currentState.getMoveLog().isEmpty()) {
                    String lastMove = currentState.getMoveLog().get(currentState.getMoveLog().size() - 1);
                    writer.write("Gerakan " + i + ": " + lastMove + "\n");
                }
            }
        }
    }
}

```

```

    } else {
        writer.write("Gerakan " + i + ": Tidak ada gerakan yang dilakukan.\n");
    }
}

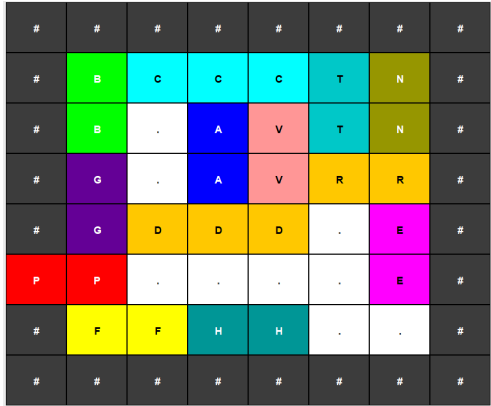
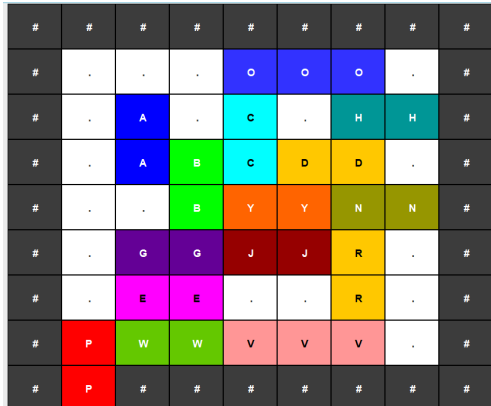
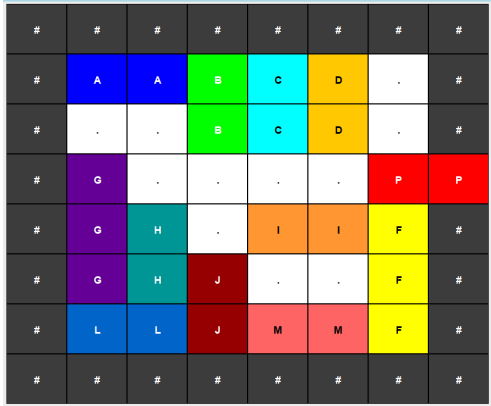
// Tulis papan
char[][] board = currentState.getStateBoard().getBoard();
for (int r = 0; r < board.length; r++) {
    for (int c = 0; c < board[r].length; c++) {
        if (board[r][c] == '#') {
            writer.write(str: " ");
        } else {
            writer.write(board[r][c]);
        }
    }
    writer.write(str: "\n");
}
writer.write(str: "\n");
}
writer.write("Jumlah Gerakan: " + (solutionPath.size() - 1) + ", Waktu: " + duration + " ms\n");
}
}

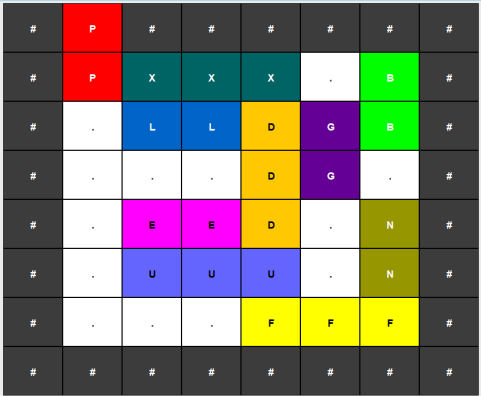
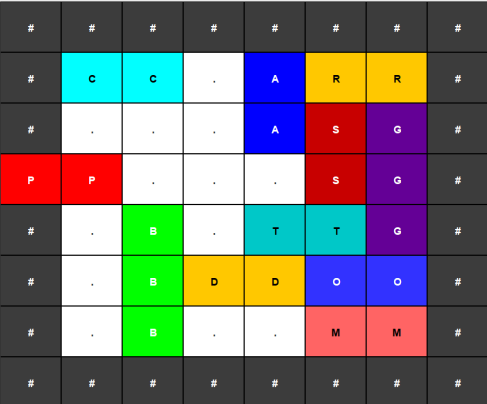
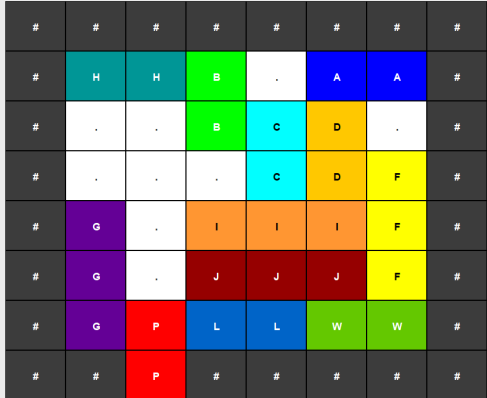
```

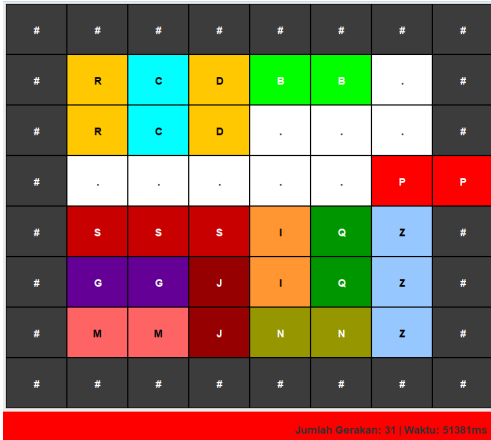
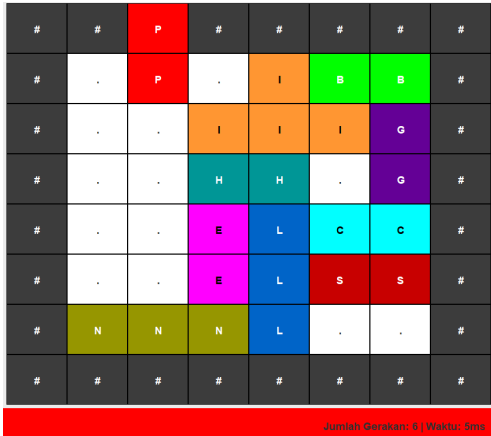
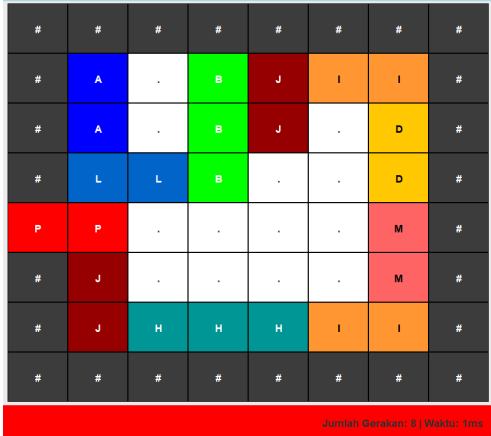
BAGIAN III

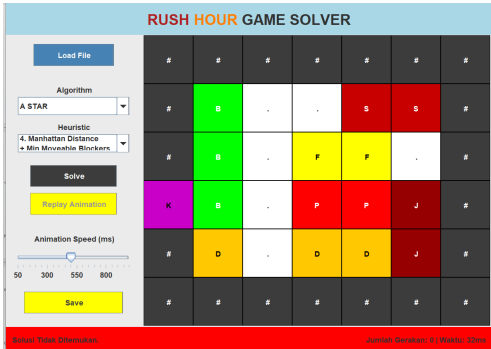
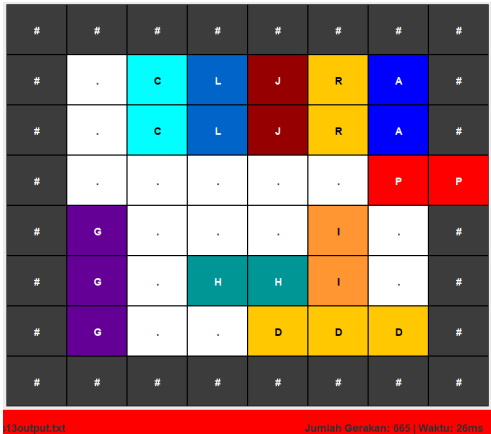
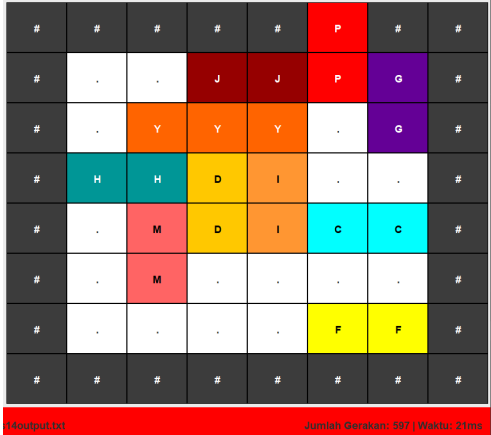
HASIL TEST

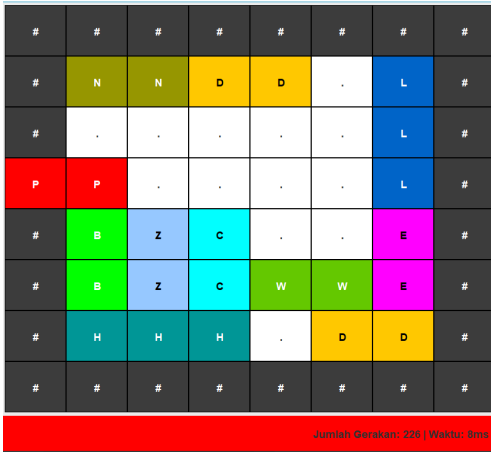
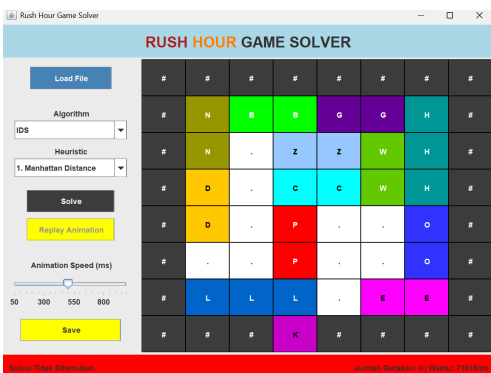
Greedy Best First Search (Greedy BFS)			
No	Test File	Input	Output
1.	tes5.txt	6 6 11 RR..JL BBB.JL .PP.ILK D..FI. D.EFGG CCE.MM	<p>Jumlah Gerakan: 249 Waktu: 81ms</p>
2.	tes6.txt	6 6 10 K AY.SSS AYECC. R.E.GG R.DD.. ...PM. FF.PM.	<p>Output.txt Jumlah Gerakan: 45 Waktu: 15ms</p>

3.	tes7.txt	6 6 12 CCCVT. B..VTN BRR..N G.ADDD KG.APPE .FFHHE	 <p>Output.txt Jumlah Gerakan: 21 Waktu: 2ms</p>
4.	tes8.txt	7 7 15 OOOC... PA.C.HH PA...DD YYNN... GGB..JJ ..BEER. WWVVVR. K	 <p>Jumlah Gerakan: 18 Waktu: 19ms</p>
Uniform Cost Search (UCS)			
1.	tes1.txt	6 6 11 AAB... ..BCDF GPPCDFK GH.IIF GHJ... LLJMM.	 <p>Jumlah Gerakan: 4 Waktu: 10ms</p>

2.	tes2.txt	6 6 9 K XXXD.B LL.DGB P..DG. P.EE.N .UUU.N ...FFF	 <p>Output.txt Jumlah Gerakan: 5 Waktu: 113ms</p>
3.	tes3.txt	6 6 10 CC.ARR .B.AS. K.BPPS. .B.TTG DDOO.G .MM..G	 <p>Jumlah Gerakan: 6 Waktu: 195ms</p>
4.	tes4.txt	6 6 11 HHB.AA .PBC.. .P.CD. GIIDF G.JJF GLLWWF K	 <p>Jumlah Gerakan: 6 Waktu: 108ms</p>
A*			

1.	tes9.txt	6 6 12 .BB.Q. RCD.QZ RCDPPZK SSSI.Z ..JIGG MMJNN.	 <p>Jumlah Gerakan: 51 Waktu: 5138ms</p>
2.	tes10.txt	6 6 11 K ...IBB IIIL.G HH.L.G .PELCC .PESS. .NNN...	 <p>Jumlah Gerakan: 6 Waktu: 5ms</p>
3.	tes11.txt	6 6 9 A..JII A.BJ.. LLB..D KJ.BPPD J....M HHHIIM	 <p>Jumlah Gerakan: 8 Waktu: 1ms</p>

4.	error1.txt	4 5 5 B..SS B.FF. KB.PPJ D.DDJ	
IDS			
1.	tes13.txt	6 6 10 .C..R. .C.JRA PPLJ.AK G.L.I. GHH.I. G..DDD	
2.	tes14.txt	6 6 9 K ...IJJ YYYY.IG HHD..G .MDCC. .M..P. .FF.P.	

3.	tes15.txt	6 6 8 NNDD.L B....L KBZCPPL .ZC... .WW..E HHHDDE	
4.	error2.txt	6 6 11 NBBGGH N.ZZWH D.CCWH D.P..O ..P..O LLL.EE K	

BAGIAN IV

ANALISIS

Berdasarkan percobaan yang dilakukan pada keempat algoritma *pathfinding* dengan pengujian konfigurasi papan puzzle Rush Hour yang bervariasi beserta pemilihan heuristik yang beragam, dapat diketahui secara *general* algoritma UCS memiliki waktu pencarian yang lebih lama dibandingkan algoritma-algoritma lain yang digunakan. Hal ini disebabkan oleh mekanisme algoritma UCS yang menelusuri semua node berdasarkan *cost path* dari state awal (starting point) hingga state saat ini ($g(n)$) tanpa adanya heuristik atau estimasi tujuan. Algoritma yang memiliki waktu pencarian yang lebih cepat adalah algoritma Greedy Best First Search (GBFS) yang mengandalkan fungsi heuristik $h(n)$ untuk menelusuri path yang memiliki estimasi *cost* terendah menuju *goal node*. Namun, algoritma GBFS tidak selalu menghasilkan solusi yang optimal. Algoritma yang paling *balance* dan memiliki kelebihan terbanyak adalah algoritma A* yang memiliki waktu pencarian yang juga cepat dan akan menghasilkan solusi yang optimal jika digunakan heuristik yang *admissible*.

Mengenai kompleksitas algoritma GBFS, UCS, dan A*, hal-hal yang menjadi faktor adalah branching factor (b), depth of the shallowest goal (d), maximum depth of the search space (m), cost of optimal solution (C^*), dan minimum step cost (ϵ). Kompleksitas waktu dari algoritma GBFS adalah $O(b^m)$ dalam kasus terburuk, yakni algoritma GBFS akan mengeksplorasi search space hingga kedalaman maksimum m . Namun, dengan heuristik yang baik, secara praktis algoritma GBFS dapat mendekati $O(b^d)$ jika heuristiknya sangat memandu menuju arah *goal node*. Kompleksitas ruang dari algoritma GBFS juga sama, yakni $O(b^m)$ untuk kasus terburuk di mana perlu disimpan semua *node* hingga kedalaman maksimum m . Untuk algoritma UCS, kompleksitas waktu dan ruang nya sama yaitu $O(b^{C^*/\epsilon})$. Alasan dari kompleksitas algoritma UCS sebesar $O(b^{C^*/\epsilon})$ adalah algoritma UCS bekerja dengan mengeksplorasi semua node yang biaya akumulatifnya dari node awal atau *path cost* hingga saat ini ($g(n)$) kurang dari atau sama dengan biaya solusi optimal. Kompleksitas ruang dari algoritma UCS juga sebesar $O(b^{C^*/\epsilon})$ karena dalam kasus terburuk, UCS mungkin perlu menyimpan semua node yang telah dieksplorasi hingga biaya C^* di dalam memori. Pada ketiga algoritma, terdapat tambahan overhead untuk kompleksitas waktunya, yakni $O(\log n)$ dari operasi pop dan push untuk priority Queue dan $O(1)$ dari penggunaan HashSet untuk pengecekan node yang telah dikunjungi (*visited*).

Algoritma terakhir yaitu algoritma IDS memiliki keunggulan efisiensi ruang dari DFS dan menjamin solusi yang optimal dengan kedalaman minimum seperti BFS. Algoritma ini melakukan pencarian DFS berulang kali untuk setiap batas kedalaman yang ditingkatkan secara inkremental. Kompleksitas waktu dan kompleksitas ruang dari Algoritma IDS adalah $O(b^d)$, yakni algoritma akan menjelajahi semua *node* hingga kedalaman d , sedangkan kompleksitas ruang dari Algoritma IDS adalah $O(bd)$, di mana Algoritma IDS hanya perlu menyimpan jalur *node* dari *root* hingga kedalaman saat ini. Namun, kekurangan dari Algoritma IDS adalah redundansi dalam mengunjungi ulang node untuk tiap iterasi. Hal ini menyebabkan adanya waktu komputasi berulang yang membuat algoritma ini lebih lambat dari algoritma seperti BFS dan A*.

BAGIAN IV

IMPLEMENTASI BONUS

Heuristik Tambahan

```
package src;

import java.util.Map;

public class MoveablePieces implements Heuristic {
    public int calculateHeuristic(State state) {
        int blockingMoves = 0;
        for (Map.Entry<Character, Piece> entry : state.getPieces().entrySet()) {
            Piece piece = entry.getValue();
            if (piece.getPieceID() == 'P') continue;
            if (state.isBlockingExit(piece)) {
                if (piece.isHorizontal()) {
                    if (piece.canMove(state, moveRow:0, moveCol:1) || piece.canMove(state, moveRow:0, -1)) {
                        blockingMoves++;
                    } else {
                        blockingMoves += 2;
                    }
                } else {
                    if (piece.canMove(state, moveRow:1, moveCol:0) || piece.canMove(state, -1, moveCol:0)) {
                        blockingMoves++;
                    } else {
                        blockingMoves += 2;
                    }
                }
            }
        }
        return blockingMoves;
    }
}
```

```
package src;

public class DistanceToExit implements Heuristic {

    public int calculateHeuristic(State state) {
        return state.calculateDistanceToExit();
    }
}
```

Selain menggunakan Heuristik jumlah *pieces* yang memblokir jalur keluar *primary piece*, diimplementasikan heuristik lain, yaitu:

- Jarak Primary Piece dengan pintu keluar
- Jumlah *blocking pieces* yang dapat digeser keluar dari jalur keluar *primary piece*

Heuristik pertama tidak admissible karena estimasi cost menuju *goal node* $h(n)$ dengan heuristik ini meng-*underestimate* cost yang sebenarnya $h^*(n)$, yakni mengabaikan jumlah *pieces* yang memblokir jalur keluar *primary piece*, contohnya $h(n)$ ketika jarak antara primary piece dengan pintu keluar atau *goal state* berjarak tiga cell yang salah satu cellnya ditempati *blocking piece* adalah 3, yang meng-*overestimate* cost sebenarnya, yakni 1. Heuristik kedua admissible karena merupakan variansi dari heuristik jumlah *blocking pieces* pada jalur keluar, bedanya *blocking pieces* yang tidak dapat bergerak akan memiliki nilai heuristik yang lebih besar sedikit sehingga lebih heuristik akan memprioritaskan *state* yang memiliki pergerakan *blocking piece*, yang menjamin tidak meng-*overestimate* cost sebenarnya, yakni $h(n) \leq h^*(n)$.

Algoritma IDS

```
public boolean solveWithIDS(State initialState) {
    int depthLimit = 0;
    int MAX_DEPTH = 1000; // kasus unsolvable
    this.solutionPath = null;
    while (depthLimit < MAX_DEPTH) {
        System.out.println("Depth Limit: " + depthLimit);
        Set<String> visited = new HashSet<>();

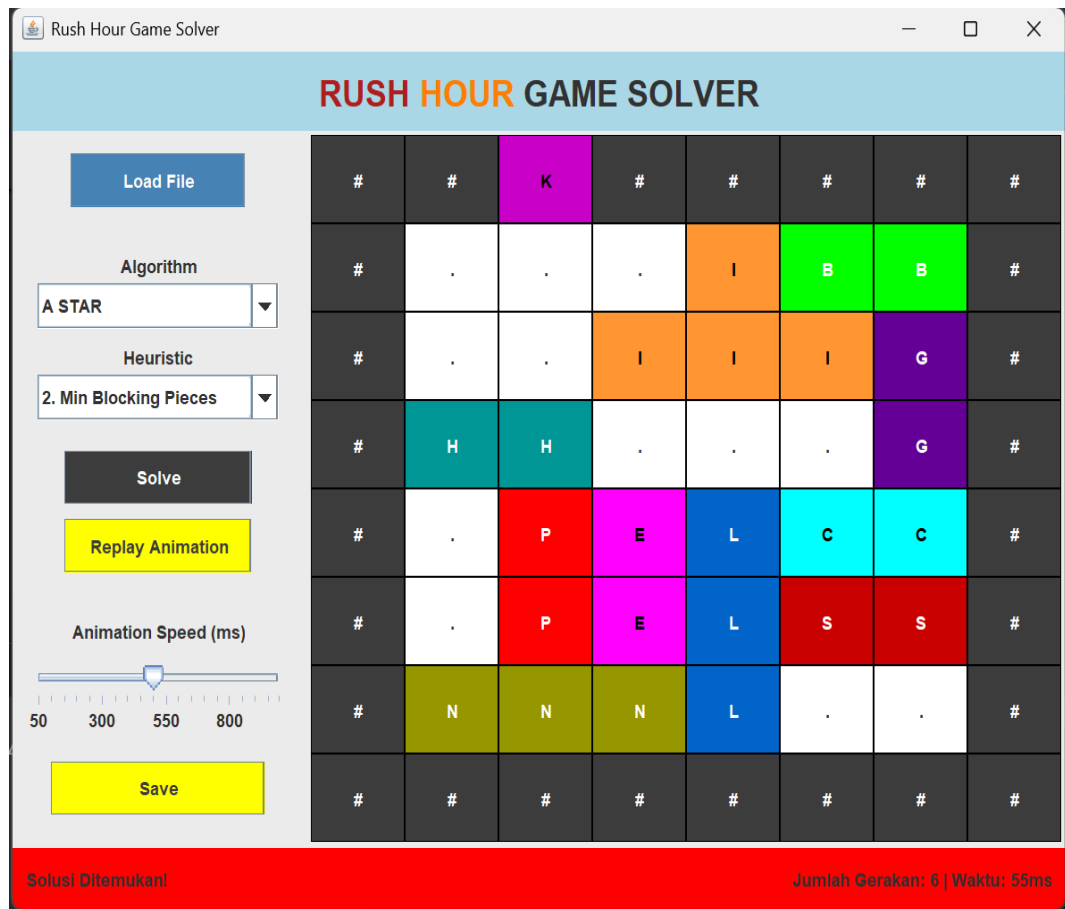
        State resultState = depthLimitedSearch(initialState, depthLimit, visited);
        if (resultState != null) {
            this.solutionPath = reconstructPath(resultState);
            return true;
        }
        depthLimit++;
    }
    return false;
}

private State depthLimitedSearch(State currentState, int depthLimit, Set<String> visited) {
    if (currentState.isGoal(currentState.getStateBoard().getExitY(), currentState.getStateBoard().getExitX())) {
        return currentState;
    }
    visited.add(currentState.getUniqueStateID());

    for (State nextState : currentState.generateSuccessors()) {
        if (!visited.contains(nextState.getUniqueStateID())) {
            nextState.setParent(currentState);
            State resultState = depthLimitedSearch(nextState, depthLimit - 1, visited);
            if (resultState != null) {
                return resultState;
            }
        }
    }
    return null;
}
```

Algoritma Iterative Deepening Search (IDS) adalah algoritma yang melakukan DFS untuk setiap batas kedalaman yang dimulai dari 0 yang kemudian dinaikkan batasnya secara inkremental hingga ditemukan *goal node* pada kedalaman tertentu. Algoritma ini menggabungkan keunggulan BFS dalam menemukan solusi yang optimal (terpendek atau kedalaman terendah) dan DFS untuk penggunaan ruang memori yang efisien.

Bonus GUI



Program dijalankan pada GUI memanfaatkan package Swing yang sudah disediakan oleh bahasa Java. GUI memiliki fitur load file berbasis filechooser yang memungkinkan pengguna memilih file konfigurasi permainan dalam .txt sebagai input pada direktori yang dipilih. Selain itu, terdapat fitur dropdown untuk memilih algoritma dan jenis heuristik yang dipakai (heuristik tidak berpengaruh pada algoritma UCS dan IDS). Tombol Solve digunakan untuk menghasilkan solusi dari input puzzle Rush Hour yang dipilih. Hasil solusi akan tampak sebagai solusi papan pada bagian kanan dalam bentuk animasi pergerakan piece untuk mencapai solusi dan informasi jumlah gerakan beserta waktu proses pada bagian bottom-bar. Selain itu, Tombol Replay Animation dapat digunakan untuk mengulangi animasi pergerakan piece untuk mencapai solusi kembali dengan kecepatan animasi (dalam milisekon) dapat diatur dengan slider di bawahnya. Pengguna dapat menggunakan fitur tombol save untuk menyimpan file hasil solusi permainan ke dalam direktori yang dipilih.

LINK REPOSITORY

https://github.com/AgungLucker/Tucil1_13523023

TABEL CHECKLIST

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	✓	
2	Program berhasil dijalankan	✓	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4	Program dapat membaca masukan berkas .txt serta menyimpan solusi dalam berkas .txt	✓	
5	[Bonus] Implementasi algoritma pathfinding alternatif	✓	
6	[Bonus] Implementasi 2 atau lebih heuristik alternatif	✓	
7	[Bonus] Program memiliki GUI	✓	
9	Program dan laporan dibuat (kelompok) sendiri	✓	