

Laporan Tugas Kecil IF2211 Strategi Algoritma

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun Oleh:

Athian Nugraha Muarajuang (13523106)

Andri Nurdianto (13523145)

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

Daftar Isi

Daftar Isi	2
Penjelasan Algoritma	3
Uniform Cost Search	3
Greedy Best First Search	4
A*	5
Beam Search Algorithm	6
Analisis Algoritma	8
Source Program	9
Input dan Output	73
Hasil Analisis	85
Implementasi Bonus	86
Implementasi algoritma pathfinding alternatif	86
Implementasi 2 atau lebih heuristik alternatif	86
Program memiliki GUI	86
Lampiran	87
Pranala Repository Github	87
Tabel Ketercapaian	87

Penjelasan Algoritma

Uniform Cost Search

Uniform cost search merupakan algoritma pencarian jalur yang digunakan untuk menemukan solusi dengan biaya total terkecil dari titik awal ke tujuan dalam sebuah graf berbobot. UCS mempertimbangkan jumlah langkah atau transisi antar state. Dalam permainan rush hour state digambarkan sebagai GameState. Dalam prosesnya UCS menggunakan priority queue dengan mempertimbangkan biaya terendah akan diproses terlebih dahulu. UCS akan terus mengeksplorasi node berdasarkan urutan biaya terendah hingga mencapai tujuan.

Algoritma:

1. Inisialisasi dengan menyiapkan priority queue yang menyimpan state yang akan dieksplorasi. Urutan priority queue berdasarkan cost terendah.
2. Validasi nilai initialState dan initialBoard. Jika bernilai null hasil akan langsung mengembalikan nilai false.
3. InisialState dimasukan ke variabel frontier
4. Gamestate mengambil biaya terendah dari frontier.
5. Cek apakah current sudah mencapai goal state. Jika belum BoardState akan dibuat ulang sesuai jalur solusi. Jika berhasil, proses akan berhenti dan mengembalikan nilai SolutionResult.
6. Ambil semua seccesor dari current. Setiap succesor akan dimasukan ke frontier.
7. Jika frontier kosong, tidak ada jalur ke K dan SolutionResult gagal.

Pseudocode

```
function solve(initialState):
    start_time ← current_time
    visited_nodes ← 0
    frontier ← priority queue with initial_state
    visited_boards ← empty set

    if initial_state is null or initial_state.board is null:
        return failure_result

    add initial_state.board to visited_boards

    nodes_since_last_publish ← 0
    PUBLISH_INTERVAL ← 500

    while frontier is not empty:
        current ← frontier.poll()
        visited_nodes += 1
        nodes_since_last_publish += 1

        if nodes_since_last_publish ≥ PUBLISH_INTERVAL:
            publishVisitedNodesCount()
            nodes_since_last_publish ← 0
```

```

        if current is goal state:
            execution_time ← current_time - start_time
            if nodes_since_last_publish > 0:
                publishVisitedNodesCount()
            path ← reconstructBoardStates(current)
            return success_result(current.moves,
visited_nodes, execution_time, path)

        successors ← current.getSuccessors()
        for each successor in successors:
            if successor.board not in visited_boards:
                add successor.board to visited_boards
                add successor to frontier

    return failure_result

```

Greedy Best First Search

Greedy best first search adalah algoritma yang menggunakan heuristik untuk menentukan langkah yang sebaiknya diambil terlebih dahulu. Greedy best first search hanya fokus pada seberapa dekat suatu state ke tujuan berdasarkan heuristik $h(n)$. Algoritma ini berjalan tanpa memedulikan cost dari titik awal.

Algoritma

1. Inisialisasi dengan menyiapkan priority queue untuk menyimpan state. Urutan priority queue berdasarkan cost heuristic terkecil.
2. Validasi nilai initialState dan initialBoard. Jika bernilai null hasil akan langsung mengembalikan nilai false.
3. InisialState dimasukan ke variabel frontier.
4. Jika frontier tidak kosong. Ambil state dengan cost heuristic terkecil.
5. Cek apakah current sudah mencapai goal state. Jika belum BoardState akan dibuat ulang sesuai jalur solusi. Jika berhasil, proses akan berhenti dan mengembalikan nilai SolutionResult.
6. Ambil succesor dan nilai heuristicnya dihitung. Jika nilai heuristicnya valid dan node belum dikunjungi, masukan succesor ke frontier untuk diproses berikutnya.
7. Jika frontier kosong, tidak ada jalur ke K dan SolutionResult gagal.

Pseudocode

```

function solve(initialState, heuristic):
    startTime ← current time
    visitedNodesCount ← 0
    frontier ← priority queue with heuristic cost
    visited ← empty set

    if initialState is null or board is null:

```

```

        return failure

    add initialState to frontier
    add initialState.board to visited

    while frontier not empty:
        current ← frontier.pop()
        visitedNodesCount += 1

        if current is goal:
            executionTime ← current time - startTime
            return solution

        for each successor in current.getSuccessors():
            if successor.board not in visited:
                h ← heuristic(successor)
                if h == ∞:
                    continue
                successor.hCost ← h
                add successor to visited
                add successor to frontier

    executionTime ← current time - startTime
    return failure_result

```

A*

A* adalah algoritma pencarian yang menggunakan kombinasi antara biaya perjalanan dari titik awal ke node yang sedang dikunjungi $g(n)$ dan estimasi cost dari node saat ini ke tujuan $h(n)$ untuk menentukan urutan eksplorasi node. Nilai total untuk menentukan prioritas nodenya adalah $f(n) = g(n) + h(n)$.

Algoritma

1. Inisialisasi dengan menyiapkan priority queue untuk menyimpan state. Urutan priority queue berdasarkan cost heuristic terkecil.
2. Validasi nilai initialState dan initialBoard. Jika bernilai null hasil akan langsung mengembalikan nilai false.
3. InisialState dimasukan ke variabel frontier.
4. Jika frontier tidak kosong. Ambil state dengan cost $f(n)$ terkecil.
5. Cek apakah current sudah mencapai goal state. Jika belum BoardState akan dibuat ulang sesuai jalur solusi. Jika berhasil, proses akan berhenti dan mengembalikan nilai SolutionResult.
6. Ambil succesor lalu nilai $h(n)$ dan $f(n)$ dihitung. Jika nilai $h(n)$, $f(n)$ valid, dan node belum dikunjungi, masukan succesor ke frontier untuk diproses berikutnya.
7. Jika frontier kosong, tidak ada jalur ke K dan SolutionResult gagal.

Pseudocode

```
function solve(initialState, heuristic):
    start_time ← current_time()
    visited_nodes ← 0
    frontier ← PriorityQueue ordered by f(n)
    visited ← empty_set

    if initial_state is invalid:
        return failure result

    frontier.add(initial_state)
    visited.add(initial_state.board)

    while frontier is not empty:
        current ← frontier.pop_lowest_f()

        visited_nodes ← visited_nodes + 1

        if current is goal:
            execution_time ← current_time() - start_time
            return success result with path and stats

        for each successor in current.get_successors():
            if successor.board not in visited:
                h ← heuristic(successor)
                if h == ∞:
                    continue
                successor.hCost ← h
                successor.fCost ← successor.gCost + h
                visited.add(successor.board)
                frontier.add(successor)

    execution_time ← current_time() - start_time
    return failure result
```

Beam Search Algorithm

Beam search adalah algoritma pencarian dengan heuristic yang menelusuri graf dengan memperluas node yang terbaik secara sistematis dalam suatu set. Algoritma ini menggabungkan BFS untuk membangun pohon pencariannya dengan menghasilkan successor di setiap level. Algoritma ini hanya mengevaluasi dan memperluas sejumlah set dari node terbaik pada setiap level berdasarkan cost heuristicnya.

Algoritma

1. Inisialisasi dengan menyiapkan priority queue untuk menyimpan state. Urutan priority queue berdasarkan cost heuristic terkecil.
2. Simpan state yang sedang dihitung. Pada setiap iterasi successor dikembangkan dari setiap state dalam beam lalu hitung heuristic semua successor

3. Pilih beamWidth successor terbaik berdasarkan heuristic untuk dilakukan iterasi selanjutnya.
4. Iterasi dihentikan jika state menemukan goal state yang artinya mengembalikan nilai solusi yang berhasil atau tidak ada successor yang tersisa untuk diperluas yang menandakan solusi tidak ditemukan

Pseudocode

```

function solve(initialState, heuristic, beamWidth):
    startTime ← current time
    visited ← empty set
    beam ← empty list

    if initialState is null or heuristic is null:
        return failure result

    initialCost ← heuristic(initialState)
    if initialCost is ∞:
        return failure result

    set heuristic cost of initialState to initialCost
    add initialState to beam
    add initialState.board to visited

    while beam is not empty:
        candidates ← empty priority queue (by heuristic cost)

        for each state in beam:
            if state is goal:
                return success result with path

            successors ← generate successors of state
            for each successor in successors:
                if successor.board not in visited:
                    cost ← heuristic(successor)
                    if cost ≠ ∞:
                        set heuristic cost of successor
                        add successor to candidates
                        add successor.board to visited

            beam ← empty list
            for i from 1 to beamWidth:
                if candidates is empty:
                    break
                best ← remove state with lowest heuristic from
candidates
                add best to beam

    return failure result

```

Analisis Algoritma

Pada program Rush Hour Solver ini, $f(n)$ didefinisikan sebagai total cost menuju goal state. Pada algoritma A* dan Beam Search Algorithm $f(n)$ adalah jumlah dari $h(n)$ dan $g(n)$, sedangkan pada algoritma Greedy Best First Search $f(n) = h(n)$ karena algoritma ini hanya mementingkan cost dari heuristic.

Heuristic pada algoritma A* admissible jika tidak melebihi biaya minimum yang sebenarnya $h(n) \leq h^*(n)$. Pada program Rush Hour Solver A* menghitung $f(n)$ minimum yang merupakan jumlah dari $h(n) + g(n)$ hal ini bisa saja membuat algoritma A* tidak admissible jika $g(n)$ adalah nilai yang kecil sedangkan $h(n)$ adalah nilai yang besar.

UCS memilih node dengan $g(n)$ terkecil sedangkan BFS menjelajahi semua node pada level yang sama sebelum menuju level selanjutnya. jika setiap langkah mobil dihitung sebagai 1 langkah, UCS dan BFS akan membangkitkan node yang sama dan menghasilkan path yang sama. Namun, pada program ini panjang gerakan tidak sama dengan jumlah langkah sehingga UCS berbeda dengan BFS.

A* bisa lebih efisien dibanding UCS jika admissible. UCS membangkitkan node dengan hanya mempertimbangkan $g(n)$ sedangkan A* mempertimbangkan jumlah $g(n)$ dan $h(n)$. Dengan begitu A* lebih cepat menuju ke arah goal bukan menyebar seperti UCS.

Greedy best first search hanya mempertimbangkan $h(n)$ tanpa mempertimbangkan sudah seberapa banyak cost yang ada sampai statenya. Jadi, bisa saja memilih jalur dengan $h(n)$ yang kecil tetapi dengan total yang lebih tinggi sehingga greedy best first search tidak menjamin solusi optimal.

Beam Search, berbeda dengan greedy best first search yang hanya mempertimbangkan node terbaik, mempertimbangkan jumlah kandidat node sesuai dengan *beam width* (lebar beam). Keuntungan dari algoritma ini adalah pengelolaan memory yang lebih terkontrol dibandingkan dengan algoritma UCS dan A*, terutama dalam kasus branch yang sangat lebar. Namun Beam Search tidak menjamin mendapatkan solusi optimal atau bahkan tidak menemukan solusi jika jalur solusi optimal tidak dalam *scope beam*.

Source Program

RushHourMain.java

```
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import view.gui.RushHourGUI;

public class RushHourMain {
    public static void main(String[] args) {
        runGUIMode();
    }
    private static void runGUIMode() {
        try {

            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            System.err.println("Gagal mengatur Look and Feel sistem.");
        }

        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new RushHourGUI();
            }
        });
    }
}
```

RushHourGUI.java

```
package view.gui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

import model.core.Board;
import model.GameState;
```

```

import model.core.Move;
import controller.SolutionResult;
import controller.solver.*;
import controller.heuristic.*;
import utils.FileHandler;

public class RushHourGUI {
    private JFrame frame;
    private BoardPanel boardPanel;
    private ControlPanel controlPanel;
    private StatusPanel statusPanel;

    private Board currentBoard;
    private SolutionResult currentSolution;
    private Timer animationTimer;
    private int animationStepIndex;

    private final FileHandler fileHandler;

    public RushHourGUI() {
        fileHandler = new FileHandler();
        initialize();
    }

    public JFrame getFrame() {
        return frame;
    }

    public void initialize() {
        frame = new JFrame("Rush Hour Puzzle Solver");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout(10, 10));

        boardPanel = new BoardPanel(50);
        controlPanel = new ControlPanel(this);
        statusPanel = new StatusPanel();

        boardPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        controlPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 0, 10));

        statusPanel.setBorder(BorderFactory.createEmptyBorder(0, 10, 10, 10));

        frame.add(controlPanel, BorderLayout.NORTH);
        frame.add(boardPanel, BorderLayout.CENTER);
    }
}

```

```

        frame.add(statusPanel, BorderLayout.SOUTH);

        frame.setMinimumSize(new Dimension(650, 600));
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    /**
     * Memuat papan dari path file yang diberikan dan
    memperbarui GUI.
     * Metode ini dipanggil dari ControlPanel.
     * @param filePath Path absolut ke file konfigurasi
    papan.
     * @return true jika papan berhasil dimuat, false jika
    gagal.
     */
    public boolean loadBoard(String filePath) {
        System.out.println("Mencoba memuat papan dari: " +
        filePath);
        Board loadedBoard =
        fileHandler.readBoardFromFile(filePath);

        if (loadedBoard != null) {
            this.currentBoard = loadedBoard;
            boardPanel.setBoard(this.currentBoard);
            statusPanel.updateStatus("Papan berhasil
            dimuat dari: " + new File(filePath).getName());
            statusPanel.clearStatistics();
            controlPanel.enableSolveButton(true);
            controlPanel.enableAnimateButton(false);
            this.currentSolution = null;
            if (animationTimer != null &&
            animationTimer.isRunning()) {
                animationTimer.stop();
            }
            System.out.println("Papan dimuat dan GUI
            diperbarui.");
            return true;
        } else {
            this.currentBoard = null;
            boardPanel.setBoard(null);
            statusPanel.updateStatus("Gagal memuat papan.
            Periksa konsol untuk error.");
            JOptionPane.showMessageDialog(frame,
                "Tidak dapat memuat papan dari file:
                " + new File(filePath).getName() + "\nSilakan periksa format
                file dan konsol untuk error.",
                "Error Memuat Papan",
                JOptionPane.ERROR_MESSAGE);
        }
    }

```

```

        controlPanel.enableSolveButton(false);
        controlPanel.enableAnimateButton(false);
        System.err.println("Gagal memuat papan dari: "
+ filePath);
        return false;
    }
}

/**
 * Memulai proses penyelesaian puzzle.
 * Metode ini dipanggil dari ControlPanel.
 * Akan membuat solver berdasarkan pilihan pengguna dan
memanggil solvePuzzle.
 */
public void initiateSolveProcess() {
    if (currentBoard == null) {
        JOptionPane.showMessageDialog(frame, "Silakan
muat papan terlebih dahulu.", "Papan Belum Dimuat",
JOptionPane.WARNING_MESSAGE);
        return;
    }

    String algorithmName =
controlPanel.getSelectedAlgorithm();
    String heuristicName =
controlPanel.getSelectedHeuristic();

    if (algorithmName == null) {
        JOptionPane.showMessageDialog(frame, "Silakan
pilih algoritma.", "Algoritma Belum Dipilih",
JOptionPane.WARNING_MESSAGE);
        return;
    }

    GameState initialState = new GameState(0,
currentBoard.clone(), new ArrayList<>());

    Heuristic heuristic = null;
    if (heuristicName != null &&
controlPanel.isHeuristicEnabled()) {
        heuristic = createHeuristic(heuristicName);
        if (heuristic == null &&
(algorithmName.equals("A*") || algorithmName.equals("Greedy
BFS") || algorithmName.equals("Beam Search (10 beam
width)"))) {
            JOptionPane.showMessageDialog(frame,
"Heuristik '" + heuristicName + "' tidak dikenal.", "Error
Heuristik", JOptionPane.ERROR_MESSAGE);
            return;
        }
    }
}

```

```

    }

    if (heuristic == null &&
        (algorithmName.equals("A*") || algorithmName.equals("Greedy
        BFS") || algorithmName.equals("Beam Search (10 beam
        width)"))) {
        JOptionPane.showMessageDialog(frame,
        "Algoritma " + algorithmName + " memerlukan heuristik.
        Silakan pilih satu.", "Heuristik Diperlukan",
        JOptionPane.WARNING_MESSAGE);
        return;
    }

    RushHourSolver solver = createSolver(algorithmName,
    initialState, heuristic);

    if (solver != null) {
        solvePuzzle(solver);
    } else {
        JOptionPane.showMessageDialog(frame, "Gagal
        membuat solver untuk algoritma: " + algorithmName, "Error
        Solver", JOptionPane.ERROR_MESSAGE);
        statusPanel.updateStatus("Gagal membuat
        solver.");
    }
}

/**
 * Membuat instance Heuristic berdasarkan nama.
 * @param heuristicName Nama heuristik.
 * @return Instance Heuristic atau null jika nama tidak
        dikenal.
 */
private Heuristic createHeuristic(String heuristicName)
{
    if (heuristicName == null) return null;
    switch (heuristicName) {
        case "Manhattan Distance" -> {
            return new ManhattanDistance();
        }
        case "Blocking Pieces" -> {
            return new BlockingPiecesHeuristic();
        }
        default -> {
            System.err.println("Heuristik tidak
            dikenal: " + heuristicName);
            return null;
        }
    }
}
}

```

```

    /**
     * Membuat instance RushHourSolver berdasarkan nama
    algoritma, keadaan awal, dan heuristik.
     * @param algorithmName Nama algoritma.
     * @param initialState Keadaan awal permainan.
     * @param heuristic Heuristik yang akan digunakan (bisa
    null).
     * @return Instance RushHourSolver atau null jika nama
    tidak dikenal.
     */
    private RushHourSolver createSolver(String
algorithmName, GameState initialState, Heuristic heuristic) {
        if (algorithmName == null) return null;
        switch (algorithmName) {
            case "UCS" -> {
                return new UCSolver(initialState);
            }
            case "A*" -> {
                return new AStarSolver(initialState,
heuristic);
            }
            case "Greedy BFS" -> {
                return new
GreedyBestFirstSearchSolver(initialState, heuristic);
            }
            case "Beam Search (10 beam width)" -> {
                return new BeamSearchSolver(initialState,
heuristic);
            }
            default -> {
                System.err.println("Algoritma tidak
dikenal: " + algorithmName);
                return null;
            }
        }
    }

    public void solvePuzzle(RushHourSolver solver) {
        if (solver == null) {
            statusPanel.updateStatus("Solver tidak
valid.");
            return;
        }

        statusPanel.updateStatus("Menyelesaikan (" +
solver.getName() + ")... Node: 0");
        controlPanel.enableSolveButton(false);
        controlPanel.enableAnimateButton(false);
        controlPanel.enableLoadButton(false);
    }

```

```

        SwingWorker<SolutionResult, Integer> worker = new
SwingWorker<SolutionResult, Integer>() {
            @Override
            protected SolutionResult doInBackground()
throws Exception {
                Consumer<Integer> progressCallback = data
-> publish(data);

solver.setProgressEventConsumer(progressCallback);

                return solver.solve();
            }

            @Override
            protected void process(List<Integer> chunks) {
                if (chunks != null && !chunks.isEmpty())
{
                    Integer latestVisitedNodesCount =
chunks.get(chunks.size() - 1);

statusPanel.updateStatus("Menyelesaikan (" + solver.getName()
+ ")... Node: " + latestVisitedNodesCount);
                }

            @Override
            protected void done() {
                try {
                    currentSolution = get();
                    if (currentSolution != null) {

displaySolution(currentSolution);
                    } else {

statusPanel.updateStatus("Solver tidak mengembalikan hasil
solusi.");

JOptionPane.showMessageDialog(frame, "Solver tidak
mengembalikan hasil solusi.", "Hasil Tidak Valid",
JOptionPane.WARNING_MESSAGE);
                }
                catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    statusPanel.updateStatus("Proses
penyelesaian (" + solver.getName() + ") dibatalkan.");
                    System.err.println("Penyelesaian ("
+ solver.getName() + ") dibatalkan: " + e.getMessage());
                }
                catch (Exception e) {
                    e.printStackTrace();

```

```

        statusPanel.updateStatus("Error saat
menyelesaikan (" + solver.getName() + "): " +
e.getClass().getSimpleName());
        JOptionPane.showMessageDialog(frame,
"Terjadi error saat menyelesaikan (" + solver.getName() +
"):\n" + e.getMessage(), "Error Penyelesaian",
JOptionPane.ERROR_MESSAGE);
    } finally {

controlPanel.enableSolveButton(true);
        controlPanel.enableLoadButton(true);
        if (solver != null) {

solver.setProgressEventConsumer(null);
            }
            if (currentSolution != null &&
currentSolution.isSolved() && currentSolution.getMoves() !=
null && !currentSolution.getMoves().isEmpty()) {

controlPanel.enableAnimateButton(true);
                } else {

controlPanel.enableAnimateButton(false);
                    }
            }
        }
    };
    worker.execute();
}

/**
 * Menampilkan hasil solusi di GUI.
 * @param result Hasil solusi dari solver.
 */
public void displaySolution(SolutionResult result) {
    this.currentSolution = result;
    if (result != null) {
        statusPanel.updateStatus(result.isSolved() ?
"Solusi ditemukan!" : "Solusi tidak ditemukan.");
        showStatistics(result);
        boardPanel.setBoard(currentBoard);
        if (result.isSolved() && result.getMoves() !=
null && !result.getMoves().isEmpty()) {
            controlPanel.enableAnimateButton(true);
        } else {
            controlPanel.enableAnimateButton(false);
        }
    } else {
        statusPanel.updateStatus("Solver gagal atau
tidak mengembalikan hasil.");
    }
}

```



```

        }
    }

    public void animateSolution() {
        if (currentSolution == null ||
!currentSolution.isSolved() ||
            currentSolution.getBoardStates() == null ||
currentSolution.getBoardStates().isEmpty() ||
            currentSolution.getMoves() == null ||
currentSolution.getMoves().isEmpty()) {
            JOptionPane.showMessageDialog(frame, "Tidak
ada solusi untuk dianimasikan atau solusi kosong.", "Error
Animasi", JOptionPane.WARNING_MESSAGE);
            return;
        }

        if (animationTimer != null &&
animationTimer.isRunning()) {
            animationTimer.stop();
        }

        animationStepIndex = 0;
        boardPanel.setBoard(currentBoard);
        boardPanel.highlightPiece(' ', false);

        int delay = controlPanel.getAnimationSpeed();

        animationTimer = new Timer(delay, new
ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (animationStepIndex <
currentSolution.getBoardStates().size()) {
                    Board nextBoardState =
currentSolution.getBoardStates().get(animationStepIndex);
                    Move lastMove =
currentSolution.getMoves().get(animationStepIndex);

                    boardPanel.setBoard(nextBoardState);

                    boardPanel.highlightPiece(lastMove.getPieceId(), true);

                    statusPanel.updateStatus("Animasi
langkah: " + (animationStepIndex + 1) + "/" +
currentSolution.getMoves().size());
                    animationStepIndex++;
                } else {
                    animationTimer.stop();
                    statusPanel.updateStatus("Animasi

```

```

selesai.");
                                if
(!currentSolution.getBoardStates().isEmpty()){

boardPanel.setBoard(currentSolution.getBoardStates().get(curr
entSolution.getBoardStates().size() - 1));
                                }
                                boardPanel.highlightPiece(' ',
false);

controlPanel.enableAnimateButton(true);

controlPanel.enableSolveButton(true);
                                }
                                }
                                });
                                animationTimer.setInitialDelay(Math.max(0,
delay/2));
                                animationTimer.start();
                                controlPanel.enableAnimateButton(false);
                                controlPanel.enableSolveButton(false);
                                }

/**
 * Menampilkan statistik solusi di StatusPanel.
 * @param result Hasil solusi.
 */
public void showStatistics(SolutionResult result) {
    if (result != null) {
        statusPanel.updateStatistics(result);
    } else {
        statusPanel.clearStatistics();
    }
}

/**
 * Mengatur kecepatan animasi.
 * Dipanggil dari ControlPanel saat slider diubah.
 * @param speed Jeda dalam milidetik antar langkah
animasi.
 */
public void setAnimationSpeed(int speed) {
    if (animationTimer != null &&
animationTimer.isRunning()) {
        animationTimer.setDelay(speed);
    }
    System.out.println("Kecepatan animasi diatur ke: "
+ speed + "ms");
}
}

```

StatusPanel.java

```
package view.gui;

import controller.SolutionResult;
import java.awt.*;
import javax.swing.*;

public class StatusPanel extends JPanel {
    private final JLabel statusLabel;
    private final JLabel timeLabel;
    private final JLabel nodesLabel;
    private final JLabel movesLabel;

    public StatusPanel() {
        setLayout(new FlowLayout(FlowLayout.LEFT, 15, 5)); //
        Layout dengan padding
        setBorder(BorderFactory.createEtchedBorder()); //
        Tambahkan border untuk visual

        // Inisialisasi JLabel dengan teks awal
        statusLabel = new JLabel("Status: Idle");
        statusLabel.setToolTipText("Status terkini dari
        aplikasi");

        timeLabel = new JLabel("Waktu Eksekusi: - ms");
        timeLabel.setToolTipText("Waktu yang dibutuhkan untuk
        mencari solusi");

        nodesLabel = new JLabel("Node Dikunjungi: -");
        nodesLabel.setToolTipText("Jumlah state/node yang
        dieksplorasi oleh solver");

        movesLabel = new JLabel("Langkah: -");
        movesLabel.setToolTipText("Jumlah langkah dalam
        solusi yang ditemukan");

        // Menambahkan JLabel ke panel
        add(statusLabel);
        add(createVerticalSeparator());
        add(movesLabel);
        add(createVerticalSeparator());
        add(nodesLabel);
        add(createVerticalSeparator());
        add(timeLabel);
    }

    private JSeparator createVerticalSeparator() {
```

```

        JSeparator separator = new
        JSeparator(SwingConstants.VERTICAL);
        separator.setPreferredSize(new Dimension(2, 20));
        return separator;
    }

    /**
     * Memperbarui label status.
     * @param status Pesan status baru.
     */
    public void updateStatus(String status) {
        if (status != null && !status.isEmpty()) {
            statusLabel.setText("Status: " + status);
        } else {
            statusLabel.setText("Status: Idle");
        }
    }

    /**
     * Memperbarui label statistik berdasarkan
     * SolutionResult.
     * @param result Hasil solusi dari solver.
     */
    public void updateStatistics(SolutionResult result) {
        if (result != null) {
            timeLabel.setText("Waktu Eksekusi: " +
result.getExecutionTime() / 1_000_000 + " ms");
            nodesLabel.setText("Node Dikunjungi: " +
result.getVisitedNodesCount());
            if (result.isSolved() && result.getMoves() !=
null) {
                movesLabel.setText("Langkah: " +
result.getMoves().size());
            } else if (result.isSolved() && result.getMoves()
== null) { // Solved tapi tidak ada langkah (misal, sudah di
state akhir)
                movesLabel.setText("Langkah: 0");
            }
            else {
                movesLabel.setText("Langkah: -");
            }
        } else {
            // Reset ke default jika result null
            clearStatistics();
        }
    }

    /**
     * Mengosongkan semua label statistik dan mengatur status
     ke Idle.

```

```

        */
    public void clearStatistics() {
        updateStatus("Idle"); // Atau status default lainnya
        timeLabel.setText("Waktu Eksekusi: - ms");
        nodesLabel.setText("Node Dikunjungi: -");
        movesLabel.setText("Langkah: -");
    }
}

```

BoardPanel.java

```

package view.gui;

import java.awt.*;
import java.util.HashMap;
import java.util.Map;
import javax.swing.*;
import model.core.Board;
import model.core.Orientation;
import model.core.Piece;
import model.core.Position;

public class BoardPanel extends JPanel {
    private Board board;
    private final int cellSize;
    private Map<Character, Color> pieceColorMap;
    private char highlightedPieceId = ' ';
    private boolean shouldHighlight = false;

    private int xOffset = 0;
    private int yOffset = 0;

    public static final int DEFAULT_CELL_SIZE = 50;

    public BoardPanel(int cellSize) {
        this.cellSize = cellSize > 0 ? cellSize :
DEFAULT_CELL_SIZE;
        setBackground(new Color(220, 220, 220));
        initPieceColors();
        setPreferredSize(new Dimension(this.cellSize * 6,
this.cellSize * 6));
    }

    private void initPieceColors() {
        pieceColorMap = new HashMap<>();
        pieceColorMap.put('P', new Color(220, 60, 60));
        pieceColorMap.put('A', new Color(70, 130, 180));
        pieceColorMap.put('B', new Color(60, 179, 113));
    }
}

```

```

        pieceColorMap.put('C', new Color(255, 165, 0));
        pieceColorMap.put('D', new Color(147, 112, 219));
        pieceColorMap.put('E', new Color(240, 230, 140));
        pieceColorMap.put('F', new Color(0, 191, 255));
        pieceColorMap.put('G', new Color(218, 112, 214));
        pieceColorMap.put('H', new Color(127, 255, 0));
        pieceColorMap.put('I', new Color(255, 105, 180));
        pieceColorMap.put('J', new Color(32, 178, 170));
        pieceColorMap.put('L', new Color(139, 69, 19));
        pieceColorMap.put('M', new Color(112, 128, 144));
        pieceColorMap.put('N', new Color(190, 128, 144));
        pieceColorMap.put('O', new Color(112, 255, 144));
        pieceColorMap.put('Q', new Color(112, 125, 144));
        pieceColorMap.put('R', new Color(112, 128, 144));
        pieceColorMap.put('S', new Color(239, 128, 123));
        pieceColorMap.put('T', new Color(233, 128, 32));
        pieceColorMap.put('U', new Color(178, 128, 189));
        pieceColorMap.put('V', new Color(234, 128, 1));
        pieceColorMap.put('W', new Color(112, 32, 43));
        pieceColorMap.put('X', new Color(1, 178, 144));
        pieceColorMap.put('Y', new Color(0, 128, 128));
        pieceColorMap.put('Z', new Color(135, 196, 182));
        pieceColorMap.put('K', new Color(0, 100, 0));
    }

    private Color getPieceColor(char pieceId) {
        if (pieceId == 'P' &&
pieceColorMap.containsKey('P')) {
            return pieceColorMap.get('P');
        }
        return pieceColorMap.computeIfAbsent(pieceId, k ->
            new Color(Math.abs(k * 30 + 50) % 206 + 50,
Math.abs(k * 50 + 30) % 206 + 50, Math.abs(k * 70 + 20) % 206
+ 50)
        );
    }

    public void setBoard(Board board) {
        this.board = board;
        if (board != null) {
        }
        calculateOffsets();
        revalidate();
        repaint();
    }

    public void highlightPiece(char pieceId, boolean
highlight) {
        this.highlightedPieceId = pieceId;
        this.shouldHighlight = highlight;
    }

```

```

        repaint();
    }

    private void calculateOffsets() {
        if (board == null) {
            xOffset = 0;
            yOffset = 0;
            return;
        }
        int boardPixelWidth = board.getWidth() * cellSize;
        int boardPixelHeight = board.getHeight() *
cellSize;

        xOffset = Math.max(0, (getWidth() -
boardPixelWidth) / 2);
        yOffset = Math.max(0, (getHeight() -
boardPixelHeight) / 2);
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D) g;

        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

        calculateOffsets();

        if (board == null) {
            String msg = "Papan belum dimuat.";
            FontMetrics fm = g2d.getFontMetrics();
            int msgWidth = fm.stringWidth(msg);
            g2d.drawString(msg, (getWidth() - msgWidth) /
2, getHeight() / 2);
        } else {
            drawGrid(g2d, xOffset, yOffset);
            drawPieces(g2d, xOffset, yOffset);
            drawExit(g2d, xOffset, yOffset);
        }
    }

    private void drawGrid(Graphics2D g2d, int offsetX, int
offsetY) {
        g2d.setColor(Color.GRAY.brighter());
        int boardPixelWidth = board.getWidth() * cellSize;
        int boardPixelHeight = board.getHeight() *
cellSize;

        for (int i = 0; i <= board.getWidth(); i++) {

```

```

        g2d.drawLine(offsetX + i * cellSize, offsetY,
offsetX + i * cellSize, offsetY + boardPixelHeight);
    }
    for (int i = 0; i <= board.getHeight(); i++) {
        g2d.drawLine(offsetX, offsetY + i * cellSize,
offsetX + boardPixelWidth, offsetY + i * cellSize);
    }
}

private void drawPieces(Graphics2D g2d, int offsetX, int
offsetY) {
    for (Piece piece : board.getPieces()) {
        if (piece.getPositions() == null ||
piece.getPositions().isEmpty()) continue;

        Color pieceColor =
getPieceColor(piece.getId());
        g2d.setColor(pieceColor);

        int minX = Integer.MAX_VALUE, minY =
Integer.MAX_VALUE;
        int maxX = Integer.MIN_VALUE, maxY =
Integer.MIN_VALUE;

        for (Position pos : piece.getPositions()) {
            minX = Math.min(minX, pos.getX());
            minY = Math.min(minY, pos.getY());
            maxX = Math.max(maxX, pos.getX());
            maxY = Math.max(maxY, pos.getY());
        }

        int piecePixelX = offsetX + minX * cellSize;
        int piecePixelY = offsetY + minY * cellSize;
        int piecePixelWidth = (maxX - minX + 1) *
cellSize;
        int piecePixelHeight = (maxY - minY + 1) *
cellSize;

        int inset = 3;
        g2d.fillRoundRect(piecePixelX + inset,
piecePixelY + inset,
                                piecePixelWidth - 2 *
inset, piecePixelHeight - 2 * inset,
                                15, 15);

        g2d.setColor(pieceColor.darker());
        g2d.setStroke(new BasicStroke(1.5f));
        g2d.drawRoundRect(piecePixelX + inset,
piecePixelY + inset,
                                piecePixelWidth - 2 *

```



```

inset, piecePixelHeight - 2 * inset,
                                15, 15);

        if (shouldHighlight && piece.getId() ==
highlightedPieceId) {
            g2d.setColor(new Color(255, 255, 0,
100));
            g2d.fillRoundRect(piecePixelX + inset,
piecePixelY + inset,
                                piecePixelWidth - 2
* inset, piecePixelHeight - 2 * inset,
                                15, 15);
            g2d.setColor(Color.ORANGE.darker());
            g2d.setStroke(new BasicStroke(2.5f));
            g2d.drawRoundRect(piecePixelX + inset,
piecePixelY + inset,
                                piecePixelWidth - 2
* inset, piecePixelHeight - 2 * inset,
                                15, 15);
        }

        g2d.setColor(Color.BLACK);
        String pieceIdStr =
String.valueOf(piece.getId());
        Font font = new Font("SansSerif", Font.BOLD,
Math.max(10, cellSize / 3));
        g2d.setFont(font);
        FontMetrics fm = g2d.getFontMetrics();
        int stringX = piecePixelX + (piecePixelWidth -
fm.stringWidth(pieceIdStr)) / 2;
        int stringY = piecePixelY + (piecePixelHeight
- fm.getHeight()) / 2 + fm.getAscent();
        g2d.drawString(pieceIdStr, stringX, stringY);
    }
}

private void drawExit(Graphics2D g2d, int offsetX, int
offsetY) {
    if (board == null || board.getExitPosition() ==
null) return;

    Position exitPos = board.getExitPosition();
    int exitX = exitPos.getX();
    int exitY = exitPos.getY();

    Piece primaryPiece = board.getPrimaryPiece();
    if (primaryPiece == null) return;

    g2d.setColor(getPieceColor('K'));
    Font exitFont = new Font("SansSerif", Font.BOLD,

```

```

cellSize * 2 / 3);
    g2d.setFont(exitFont);
    FontMetrics fm = g2d.getFontMetrics();
    String exitChar = "K";
    int charWidth = fm.stringWidth(exitChar);
    int charHeight = fm.getAscent() - fm.getDescent();

    int drawKx;
    int drawKy;

    if (primaryPiece.getOrientation() ==
Orientation.HORIZONTAL) {
        int yPosCell = offsetY + exitY * cellSize;

        if (exitX == -1) {
            drawKx = offsetX - cellSize / 2 -
charWidth / 2;
            drawKy = yPosCell + cellSize / 2 +
charHeight / 2;
        } else if (exitX == board.getWidth()) {
            drawKx = offsetX + board.getWidth() *
cellSize + cellSize / 2 - charWidth / 2;
            drawKy = yPosCell + cellSize / 2 +
charHeight / 2;
        } else {
            return;
        }
    } else {
        int xPosCell = offsetX + exitX * cellSize;

        if (exitY == -1) {

            drawKx = xPosCell + cellSize / 2 -
charWidth / 2;
            drawKy = offsetY - cellSize / 2 +
charHeight / 2;

        } else if (exitY == board.getHeight()) {
            drawKx = xPosCell + cellSize / 2 -
charWidth / 2;
            drawKy = offsetY + board.getHeight() *
cellSize + cellSize / 2 + charHeight / 2;
        }
        else if (exitY == 0 && exitX >=0 && exitX <
board.getWidth()) {
            drawKx = xPosCell + cellSize / 2 -
charWidth / 2;
            drawKy = offsetY - cellSize / 2 +

```

```

charHeight / 2;
        } else if (exitY == board.getHeight() - 1 &&
exitX >=0 && exitX < board.getWidth()) {
            drawKx = xPosCell + cellSize / 2 -
charWidth / 2;
            drawKy = offsetY + board.getHeight() *
cellSize + cellSize / 2 + charHeight / 2;
        }
        else {
            return;
        }
    }

    if (drawKx != 0 || drawKy != 0) {
        g2d.drawString(exitChar, drawKx, drawKy);
    }
}

@Override
public void doLayout() {
    super.doLayout();
    calculateOffsets();
}
}

```

ControlPanel.java

```

package view.gui;

import javax.swing.*.*;
import java.awt.*.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;

public class ControlPanel extends JPanel {
    private JComboBox<String> algorithmSelector;
    private JComboBox<String> heuristicSelector;
    private JButton loadButton;
    private JButton solveButton;
    private JButton animateButton;
    private JSlider animationSpeedSlider;
    private final RushHourGUI parent;

    public ControlPanel(RushHourGUI parent) {
        this.parent = parent;
        setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));
        initializeComponents();
    }
}

```

```

        addListeners();
    }

    private void initializeComponents() {
        String[] algorithms = {"UCS", "A*", "Greedy BFS",
"Beam Search (10 beam width)"};
        algorithmSelector = new JComboBox<>(algorithms);
        algorithmSelector.setToolTipText("Pilih algoritma
pencarian");

        String[] heuristics = {"Manhattan Distance",
"Blocking Pieces"};
        heuristicSelector = new JComboBox<>(heuristics);
        heuristicSelector.setToolTipText("Pilih heuristik
(jika diperlukan oleh algoritma)");
        heuristicSelector.setEnabled(false);

        loadButton = new JButton("Muat Papan");
        loadButton.setToolTipText("Muat konfigurasi papan
dari file .txt");

        solveButton = new JButton("Selesaikan");
        solveButton.setToolTipText("Cari solusi untuk papan
yang dimuat");
        solveButton.setEnabled(false);

        animateButton = new JButton("Animasi");
        animateButton.setToolTipText("Animaskan
langkah-langkah solusi");
        animateButton.setEnabled(false);

        animationSpeedSlider = new
JSlider(JSlider.HORIZONTAL, 100, 2000, 500);
        animationSpeedSlider.setMajorTickSpacing(500);
        animationSpeedSlider.setMinorTickSpacing(100);
        animationSpeedSlider.setPaintTicks(true);
        animationSpeedSlider.setPaintLabels(true);
        animationSpeedSlider.setToolTipText("Kecepatan
animasi (jeda dalam milidetik)");
        animationSpeedSlider.setEnabled(false);

        add(new JLabel("Algoritma:"));
        add(algorithmSelector);
        add(new JLabel("Heuristik:"));
        add(heuristicSelector);
        add(loadButton);
        add(solveButton);
        add(animateButton);
        add(new JLabel("Kecepatan Animasi (ms):"));
        add(animationSpeedSlider);
    }

```

```

    }

    private void addListeners() {
        loadButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JFileChooser fileChooser = new
JFileChooser();
                fileChooser.setCurrentDirectory(new
File(System.getProperty("user.dir")));
                fileChooser.setDialogTitle("Pilih File
Papan Rush Hour (.txt)");
                fileChooser.setFileFilter(new
javax.swing.filechooser.FileNameExtensionFilter("File Teks
(*.txt)", "txt"));
                int returnValue =
fileChooser.showOpenDialog(parent.getFrame());
                if (returnValue ==
JFileChooser.APPROVE_OPTION) {
                    File selectedFile =
fileChooser.getSelectedFile();
                    if (parent != null) {
parent.loadBoard(selectedFile.getAbsolutePath());
                    }
                }
            }
        });

        solveButton.addActionListener(new ActionListener()
{
            @Override
            public void actionPerformed(ActionEvent e) {
                if (parent != null) {
                    parent.initiateSolveProcess();
                }
            }
        });

        animateButton.addActionListener(new
ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (parent != null) {
                    parent.animateSolution();
                }
            }
        });

        algorithmSelector.addActionListener(new

```

```

ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String selectedAlgorithm =
getSelectedAlgorithm();
        if ("A*".equals(selectedAlgorithm) ||
"Greedy BFS".equals(selectedAlgorithm) || "Beam Search (10
beam width)".equals(selectedAlgorithm)) {
            heuristicSelector.setEnabled(true);
        } else {
            heuristicSelector.setEnabled(false);
        }
        heuristicSelector.setSelectedIndex(-1);
    }
});

    animationSpeedSlider.addChangeListener(e -> {
        if
(!animationSpeedSlider.getValueIsAdjusting() && parent !=
null) {
parent.setAnimationSpeed(getAnimationSpeed());
        }
    });

    public void enableLoadButton(boolean enable) {
        loadButton.setEnabled(enable);
    }

    public void enableSolveButton(boolean enable) {
        solveButton.setEnabled(enable);
    }

    public void enableAnimateButton(boolean enable) {
        animateButton.setEnabled(enable);
        animationSpeedSlider.setEnabled(enable);
    }

    public String getSelectedAlgorithm() {
        if (algorithmSelector.getSelectedItem() != null) {
            return (String)
algorithmSelector.getSelectedItem();
        }
        return null;
    }

    public String getSelectedHeuristic() {
        if (heuristicSelector.getSelectedItem() != null &&

```

```

    heuristicSelector.isEnabled()) {
        return (String)
    heuristicSelector.getSelectedItem();
    }
    return null;
}

    public int getAnimationSpeed() {
        return animationSpeedSlider.getValue();
    }

    public boolean isHeuristicEnabled() {
        return heuristicSelector.isEnabled();
    }
}

```

FileHandler.java

```

package utils;

import controller.SolutionResult;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import model.core.Board;
import model.core.Direction;
import model.core.Move;
import model.core.Orientation;
import model.core.Piece;
import model.core.Position;

public class FileHandler {

    private static final Map<Direction, String>
    directionToIndonesian = new HashMap<>();

    static {
        directionToIndonesian.put(Direction.UP, "atas");
        directionToIndonesian.put(Direction.DOWN, "bawah");
        directionToIndonesian.put(Direction.LEFT, "kiri");
    }
}

```

```

        directionToIndonesian.put(Direction.RIGHT,
"kanan");
    }

    public Board readBoardFromFile(String path) {
        List<String> allFileLinesOriginal = new
ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new
FileReader(path))) {
            String line;
            while ((line = reader.readLine()) != null) {
                allFileLinesOriginal.add(line);
            }
        } catch (IOException e) {
            System.err.println("Error membaca file papan:
" + e.getMessage());
            return null;
        }

        if (allFileLinesOriginal.isEmpty()) {
            System.err.println("Format file papan tidak
valid: File kosong.");
            return null;
        }

        List<String> allFileLines = new ArrayList<>();
        for (String line : allFileLinesOriginal) {
            allFileLines.add(line.trim());
        }
        allFileLines.removeIf(String::isEmpty);

        if (allFileLines.size() < 2) {
            System.err.println("Format file papan tidak
valid: Jumlah baris kurang untuk dimensi dan N setelah
trim.");
            return null;
        }

        try {
            String[] dimensions =
allFileLines.get(0).split("\\s+");
            if (dimensions.length != 2) {
                String dimLine = allFileLines.get(0);
                if (dimLine.length() == 2 &&
Character.isDigit(dimLine.charAt(0)) &&
Character.isDigit(dimLine.charAt(1))) {
                    dimensions = new
String[]{String.valueOf(dimLine.charAt(0)),
String.valueOf(dimLine.charAt(1))};
                    System.out.println("Peringatan:

```



```

Dimensi papan diparsing sebagai digit tunggal yang
digabungkan (misalnya, '66'). Format yang lebih disukai
adalah dengan spasi (misalnya, '6 6').");
        } else {
            System.err.println("Format dimensi
papan tidak valid. Diharapkan 'Baris Kolom' (misalnya, '6
6'). Ditemukan: " + allFileLines.get(0));
            return null;
        }
    }
    int numRowsA =
Integer.parseInt(dimensions[0]);
    int numColsB =
Integer.parseInt(dimensions[1]);

    if (numRowsA <= 0 || numColsB <= 0) {
        System.err.println("Dimensi papan tidak
valid: Baris dan kolom harus positif.");
        return null;
    }

    Position exitPosition = null;
    int exitRow = -1, exitCol = -1;
    boolean exitIsHorizontal = false;

    List<String> gridContentLines = new
ArrayList<>();
    int fileContentStartIndex = 2;

    if (allFileLines.size() >
fileContentStartIndex) {
        String lineToTest =
allFileLinesOriginal.get(fileContentStartIndex);
        int kVisualIndex = -1;
        int kCount = 0;

        for(int i=0; i < lineToTest.length();
i++) {
            if(lineToTest.charAt(i) == 'K') {
                kVisualIndex = i;
                kCount++;
            }
        }

        if (kCount == 1) {
            boolean isDedicatedKLine = true;
            for (int i = 0; i <
lineToTest.length(); i++) {
                if (i != kVisualIndex &&
lineToTest.charAt(i) != ' ' && lineToTest.charAt(i) != '.') {

```

```

        isDedicatedKLine = false;
        break;
    }
}
if (isDedicatedKLine) {
    if (kVisualIndex >= numColsB) {
System.err.println("Error: Posisi 'K' (" + kVisualIndex + ") di
luar batas kolom papan (" + numColsB + ") pada baris K di atas
grid.");

        return null;
    }
    exitPosition = new
Position(kVisualIndex, -1);
    exitCol = kVisualIndex;
    exitRow = -1;
    exitIsHorizontal = false;
    fileContentStartIndex++;
}
}
}

    if (allFileLinesOriginal.size() <
fileContentStartIndex + numRowsA) {
        System.err.println("Format file papan
tidak valid: Jumlah baris kurang untuk konfigurasi papan
inti. Tersedia (setelah N dan K atas): " +
(allFileLinesOriginal.size() - fileContentStartIndex) + ",
Butuh: " + numRowsA);
        return null;
    }
    for (int i = 0; i < numRowsA; i++) {
gridContentLines.add(allFileLinesOriginal.get(fileContentStar
tIndex + i));
    }
    int nextLineAfterGridIndex =
fileContentStartIndex + numRowsA;

    if (exitPosition == null &&
allFileLinesOriginal.size() > nextLineAfterGridIndex) {
        String lineToTest =
allFileLinesOriginal.get(nextLineAfterGridIndex);
        int kVisualIndex = -1;
        int kCount = 0;
        for(int i=0; i < lineToTest.length();
i++) {
            if(lineToTest.charAt(i) == 'K') {
                kVisualIndex = i;

```

```

        kCount++;
    }
}
if (kCount == 1) {
    boolean isDedicatedKLine = true;
    for (int i = 0; i <
lineToTest.length(); i++) {
        if (i != kVisualIndex &&
lineToTest.charAt(i) != ' ' && lineToTest.charAt(i) != '.') {
            isDedicatedKLine = false;
            break;
        }
    }
    if (isDedicatedKLine) {
        if (kVisualIndex >= numColsB) {
System.err.println("Error: Posisi 'K' (" + kVisualIndex + ") di
luar batas kolom papan (" + numColsB + ") pada baris K di bawah
grid.");

            return null;
        }
        exitPosition = new
Position(kVisualIndex, numRowsA);
        exitCol = kVisualIndex;
        exitRow = numRowsA;
        exitIsHorizontal = false;
    }
}

    ArrayList<Piece> pieces = new ArrayList<>();
    Piece primaryPieceObj = null;
    char[][] gridForValidation = new
char[numRowsA][numColsB];
    for (int r_grid = 0; r_grid < numRowsA;
r_grid++) {
        for (int c_grid = 0; c_grid < numColsB;
c_grid++) {
            gridForValidation[r_grid][c_grid] =
'.';
        }
    }
    Map<Character, ArrayList<Position>>
pieceCandidatePositions = new HashMap<>();

    for (int r = 0; r < numRowsA; r++) {
        String currentBoardLineOriginal =
gridContentLines.get(r);
        String lineForPieceParsing =
currentBoardLineOriginal;

```

```

        boolean currentRowHasHorizontalExit =
false;
        if (exitPosition == null) {
            if (lineForPieceParsing.length() ==
numColsB + 1 && lineForPieceParsing.endsWith("K")) {
                exitPosition = new
Position(numColsB, r);
                exitRow = r;
                exitCol = numColsB;
                exitIsHorizontal = true;
                lineForPieceParsing =
lineForPieceParsing.substring(0, numColsB);
                currentRowHasHorizontalExit =
true;
            } else if
(lineForPieceParsing.length() == numColsB + 1 &&
lineForPieceParsing.startsWith("K")) {
                exitPosition = new Position(-1,
r);
                exitRow = r;
                exitCol = -1;
                exitIsHorizontal = true;
                lineForPieceParsing =
lineForPieceParsing.substring(1);
                currentRowHasHorizontalExit =
true;
            }
        } else if ((lineForPieceParsing.length()
== numColsB + 1 && (lineForPieceParsing.endsWith("K") ||
lineForPieceParsing.startsWith("K")))) {
            System.err.println("Error: Lebih
dari satu Pintu Keluar 'K' ditemukan (K atas/bawah/sebelumnya
dan K samping).");
            return null;
        }

        if (lineForPieceParsing.length() !=
numColsB) {
            System.err.println("Error: Baris
grid " + (r + 1) + " (\"" + currentBoardLineOriginal + "\")
setelah proses K samping memiliki panjang " +
lineForPieceParsing.length() + ", diharapkan " + numColsB +
".");
            return null;
        }

        for (int c = 0; c < numColsB; c++) {
            char cellChar =
lineForPieceParsing.charAt(c);

```

```

        if (cellChar == 'K') {
            if (exitPosition == null) {
                if (r != 0 && r !=
numRowsA - 1) {

System.err.println("Error: Pintu Keluar 'K' vertikal di (" +
c + "," + r + ") tidak berada di dinding atas atau bawah grid
utama.");

                return null;
            }
            exitPosition = new
Position(c, r);

            exitRow = r;
            exitCol = c;
            exitIsHorizontal = false;
            gridForValidation[r][c] =
'.';

        } else {
            System.err.println("Error:
Lebih dari satu Pintu Keluar 'K' ditemukan (K sebelumnya dan
K dalam sel grid).");

            return null;
        }
    } else if (cellChar != '.' &&
cellChar != ' ') {

pieceCandidatePositions.putIfAbsent(cellChar, new
ArrayList<>());

pieceCandidatePositions.get(cellChar).add(new Position(c,
r));

        if (gridForValidation[r][c] !=
'.') {

            System.err.println("Error:
Piece tumpang tindih di (" + c + "," + r + "). Sel ditempati
oleh '" + gridForValidation[r][c] + "' dan '" + cellChar +
"'");

            return null;
        }
        gridForValidation[r][c] =
cellChar;

    }

}

}

        if (exitPosition == null) {
            System.err.println("Error: Tidak ada
Pintu Keluar 'K' yang ditemukan dalam konfigurasi.");
            return null;
        }
    }
}

```

```

        for (Map.Entry<Character, ArrayList<Position>>
entry : pieceCandidatePositions.entrySet()) {
            char id = entry.getKey();
            ArrayList<Position> positions =
entry.getValue();
            if (positions.isEmpty()) continue;

            Collections.sort(positions,
Comparator.comparingInt(Position::getY).thenComparingInt(Posi
tion::getX));

            Orientation orientation;
            int pieceLength = positions.size();

            boolean allYSame = true;
            boolean allXSame = true;
            if (pieceLength > 0) {
                int firstY =
positions.get(0).getY();
                int firstX =
positions.get(0).getX();

                for (int i = 1; i < pieceLength;
i++) {
                    if (positions.get(i).getY() !=
firstY) allYSame = false;
                    if (positions.get(i).getX() !=
firstX) allXSame = false;
                }

                if (pieceLength == 1) {
                    if (exitIsHorizontal)
orientation = Orientation.HORIZONTAL;
                    else orientation =
Orientation.VERTICAL;
                } else if (allYSame && !allXSame) {
                    orientation =
Orientation.HORIZONTAL;
                }

                for (int i = 0; i < pieceLength
- 1; i++) {
                    if
(positions.get(i+1).getX() - positions.get(i).getX() != 1 ||
positions.get(i+1).getY() != positions.get(i).getY()) {
                        System.err.println("Error: Piece horizontal '" + id + "'
tidak kontinu. Posisi: " + positions);
                        return null;
                    }
                }
            }
        }
    }

```

```

        } else if (allXSame && !allYSame) {
            orientation =
Orientation.VERTICAL;
            for (int i = 0; i < pieceLength
- 1; i++) {
                if
(positions.get(i+1).getY() - positions.get(i).getY() != 1 ||
positions.get(i+1).getX() != positions.get(i).getX()) {
System.err.println("Error: Piece vertikal '" + id + "' tidak
kontinu. Posisi: " + positions);
                    return null;
                }
            }
        } else {
            System.err.println("Error:
Piece '" + id + "' memiliki tata letak yang tidak valid
(bukan garis lurus atau panjang tidak sesuai). Posisi: " +
positions);
                return null;
            }
        } else { continue; }

        boolean isPrimary = (id == 'P');
        Piece currentPiece = new Piece(id,
positions, orientation, isPrimary);
        pieces.add(currentPiece);
        if (isPrimary) {
            if (primaryPieceObj != null) {
                System.err.println("Error:
Lebih dari satu Primary Piece 'P' ditemukan.");
                return null;
            }
            primaryPieceObj = currentPiece;
        }
    }

    if (primaryPieceObj == null) {
        System.err.println("Error: Tidak ada
Primary Piece 'P' yang ditemukan.");
        return null;
    }

    Orientation primaryOrientation =
primaryPieceObj.getOrientation();

    if (exitIsHorizontal) {
        if (primaryOrientation !=
Orientation.HORIZONTAL) {
            System.err.println("Error: Pintu

```

```

Keluar 'K' berada di dinding horizontal (kiri/kanan), tetapi
Primary Piece 'P' (" + primaryPieceObj.getPositions() + ")
berorientasi vertikal.");
        return null;
    }
    boolean rowMatch = false;
    for (Position pPos :
primaryPieceObj.getPositions()) {
        if (pPos.getY() == exitRow) {
            rowMatch = true;
            break;
        }
    }
    if (!rowMatch) {
        System.err.println("Error: Pintu
Keluar 'K' horizontal di baris grid " + exitRow + " tidak
sejajar dengan baris Primary Piece 'P'.");
        return null;
    }
    } else {
        if (primaryOrientation !=
Orientation.VERTICAL) {
            System.err.println("Error: Pintu
Keluar 'K' berada di jalur vertikal (atas/bawah), tetapi
Primary Piece 'P' (" + primaryPieceObj.getPositions() + ")
berorientasi horizontal.");
            return null;
        }
        boolean colMatch = false;
        for (Position pPos :
primaryPieceObj.getPositions()) {
            if (pPos.getX() == exitCol) {
                colMatch = true;
                break;
            }
        }
        if (!colMatch) {
            System.err.println("Error: Pintu
Keluar 'K' vertikal di kolom grid " + exitCol + " tidak
sejajar dengan kolom Primary Piece 'P'.");
            return null;
        }
    }
    return new Board(numColsB, numRowsA, pieces,
exitPosition);

    } catch (NumberFormatException e) {
        System.err.println("Error memparsing angka
untuk dimensi atau N: " + e.getMessage());
    } catch (IndexOutOfBoundsException e) {

```



```

        System.err.println("Error memproses baris file
(kemungkinan terlalu sedikit baris atau format salah): " +
e.getMessage());
        e.printStackTrace();
    } catch (Exception e) {
        System.err.println("Terjadi error tak terduga
saat parsing papan: " + e.getMessage());
        e.printStackTrace();
    }
    return null;
}
}

```

GameState.java

```

package model;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import model.core.Board;
import model.core.Move;

public class GameState implements Comparable<GameState> {
    private final int cost;
    private final Board board;
    private final List<Move> moves;
    private transient int hCost;
    private transient int fCost;

    public GameState() {
        this.cost = 0;
        this.board = null;
        this.moves = new ArrayList<>();
        this.hCost = Integer.MAX_VALUE;
        this.fCost = Integer.MAX_VALUE;
    }

    public GameState(int cost, Board board, List<Move>
moves) {
        this.cost = cost;
        this.board = board;
        this.moves = new ArrayList<>(moves);
        this.hCost = Integer.MAX_VALUE;
        if (this.cost != Integer.MAX_VALUE) {
            this.fCost = this.cost + this.hCost;
        } else {
            this.fCost = Integer.MAX_VALUE;
        }
    }
}

```

```

    }

    public List<GameState> getSuccessors() {
        List<GameState> successors = new ArrayList<>();
        if (this.board == null) {
            System.err.println("Peringatan: Mencoba
mendapatkan successor dari GameState dengan board null.");
            return successors;
        }

        List<Move> validMoves = this.board.getValidMoves();

        for (Move move : validMoves) {
            Board newBoard = this.board.clone();

            boolean moveApplied =
newBoard.movePiece(move.getPieceId(), move.getDirection(),
move.getSteps());

            if (moveApplied) {
                List<Move> newMovesList = new
ArrayList<>(this.moves);
                newMovesList.add(move);

                successors.add(new GameState(this.cost +
1, newBoard, newMovesList));
            } else {
                System.err.println("Peringatan: Move yang
dianggap valid (" + move.getPieceId() +
                " ke " +
move.getDirection() + " sebanyak " + move.getSteps() +
                " langkah) gagal
diterapkan pada board kloningan.");
            }
        }
        return successors;
    }

    public Boolean isGoalState() {
        if (this.board == null) {
            return false;
        }
        return this.board.isPrimaryPieceAtExit();
    }

    public void setHeuristicCosts(int hCost) {
        this.hCost = hCost;
        if (this.cost == Integer.MAX_VALUE || this.hCost ==
Integer.MAX_VALUE) {
            this.fCost = Integer.MAX_VALUE;
        }
    }

```

```

        } else {
            this.fCost = this.cost + this.hCost;
        }
    }

    public List<Move> getMoves() {
        return new ArrayList<>(this.moves);
    }

    public Board getBoard() {
        return this.board;
    }

    public int getCost() {
        return this.cost;
    }

    public int gethCost() {
        return this.hCost;
    }

    public int getfCost() {
        return this.fCost;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;
        GameState gameState = (GameState) o;
        return Objects.equals(board, gameState.board);
    }

    @Override
    public int hashCode() {
        return Objects.hash(board);
    }

    @Override
    public int compareTo(GameState other) {
        int fCompare = Integer.compare(this.fCost,
other.fCost);
        if (fCompare == 0) {
            return Integer.compare(this.cost, other.cost);
        }
        return fCompare;
    }

    @Override

```

```

        public String toString() {
            return "GameState{" +
                "cost=" + cost +
                ", hCost=" + hCost +
                ", fCost=" + fCost +
                ", movesCount=" + moves.size() +
                ", boardHash=" + (board != null ?
board.hashCode() : "null") +
                '}';
        }
    }
}

```

Board.java

```

package model.core;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Objects;
import java.util.Set;

public class Board {
    private final int width;
    private final int height;
    private final ArrayList<Piece> pieces;
    private final Position exitPosition;
    private Piece primaryPiece;
    private final Piece[][] grid;

    public Board(int width, int height, ArrayList<Piece>
pieces, Position exitPosition) {
        this.width = width;
        this.height = height;
        this.exitPosition = new
Position(exitPosition.getX(), exitPosition.getY());

        this.pieces = new ArrayList<>();
        for (Piece p : pieces) {
            Piece pieceCopy = p.clone();
            this.pieces.add(pieceCopy);

            if (pieceCopy.isPrimary()) {
                this.primaryPiece = pieceCopy;
            }
        }
        this.grid = new Piece[this.height][this.width];
        populateGridFromPieces();
    }
}

```

```

    }

    private void populateGridFromPieces() {
        for (int r = 0; r < this.height; r++) {
            for (int c = 0; c < this.width; c++) {
                this.grid[r][c] = null;
            }
        }
        for (Piece piece : this.pieces) {
            if (piece.getPositions() != null) {
                for (Position pos : piece.getPositions())
                {
                    if (pos.getY() >= 0 && pos.getY() <
this.height && pos.getX() >= 0 && pos.getX() < this.width) {
this.grid[pos.getY()][pos.getX()] = piece;
                    } else {
                        System.err.println("Peringatan:
Posisi bidak " + piece.getId() + " di (" + pos.getX() + "," +
pos.getY() + ") berada di luar batas papan saat mengisi
grid.");
                    }
                }
            }
        }
    }

    public boolean movePiece(char pieceId, Direction
direction, int steps) {
        if (steps < 1) {
            System.err.println("Error: Jumlah langkah
tidak valid (" + steps + ") untuk movePiece.");
            return false;
        }

        Piece pieceToMove = this.getPieceById(pieceId);

        if (pieceToMove == null) {
            System.err.println("Error: Bidak dengan ID '"
+ pieceId + "' tidak ditemukan di papan untuk dipindahkan.");
            return false;
        }

        List<Position> currentPiecePositions =
pieceToMove.getPositions();
        for (Position pos : currentPiecePositions) {
            if (pos.getY() >= 0 && pos.getY() <
this.height && pos.getX() >= 0 && pos.getX() < this.width) {
                this.grid[pos.getY()][pos.getX()] = null;
            }
        }
    }

```

```

    }

    pieceToMove.moveBySteps(direction, steps);

    List<Position> newPiecePositions =
pieceToMove.getPositions();
    for (Position pos : newPiecePositions) {
        if (pos.getY() >= 0 && pos.getY() <
this.height && pos.getX() >= 0 && pos.getX() < this.width) {
            if (this.grid[pos.getY()][pos.getX()] !=
null && this.grid[pos.getY()][pos.getX()] != pieceToMove) {
                System.err.println("Error Kritis:
Tabrakan terdeteksi saat memindahkan piece " + pieceId + " ke
("+pos.getX()+", "+pos.getY()+") yang sudah ditempati oleh " +
this.grid[pos.getY()][pos.getX()].getId() + ". Rollback tidak
diimplementasikan di sini.");

                return false;
            }
            this.grid[pos.getY()][pos.getX()] =
pieceToMove;
        } else {
            System.err.println("Peringatan Kritis:
Posisi baru bidak " + pieceToMove.getId() + " di (" +
pos.getX() + ", " + pos.getY() + ") berada di luar batas papan
SETELAH bergerak. Ini mengindikasikan masalah pada validasi
sebelumnya.");

            for (Position oldPos :
currentPiecePositions) {
                if (oldPos.getY() >= 0 &&
oldPos.getY() < this.height && oldPos.getX() >= 0 &&
oldPos.getX() < this.width) {

this.grid[oldPos.getY()][oldPos.getX()] = pieceToMove;
                }
            }
            return false;
        }
    }
    return true;
}

public boolean isOccupied(Position position) {
    int x = position.getX();
    int y = position.getY();

    if (y >= 0 && y < this.height && x >= 0 && x <
this.width) {
        return this.grid[y][x] != null;
    }
}

```

```

        return true;
    }

    public boolean isPrimaryPieceAtExit() {
        if (primaryPiece == null) return false;

        for (Position p : primaryPiece.getPositions()) {
            if (p.equals(exitPosition)) {
                return true;
            }
        }

        for (Position p : primaryPiece.getPositions()) {
            if (primaryPiece.getOrientation() ==
Orientation.HORIZONTAL) {
                if (p.getY() == exitPosition.getY()) {
                    if (exitPosition.getX() ==
this.width && p.getX() == this.width -
primaryPiece.getPositions().size()) {

                        List<Position> pPositions =
primaryPiece.getPositions();
                        Position rightMostP =
pPositions.get(pPositions.size()-1);
                        if (exitPosition.getX() ==
this.width && rightMostP.getX() == this.width -1) return
true;

                        Position leftMostP =
pPositions.get(0);
                        if (exitPosition.getX() == -1
&& leftMostP.getX() == 0) return true;

                    }
                }
            } else {
                if (p.getX() == exitPosition.getX()) {
                    List<Position> pPositions =
primaryPiece.getPositions();
                    Position bottomMostP =
pPositions.get(pPositions.size()-1);
                    if (exitPosition.getY() ==
this.height && bottomMostP.getY() == this.height -1) return
true;

                    Position topMostP =
pPositions.get(0);
                    if (exitPosition.getY() == -1 &&

```

```

topMostP.getY() == 0) return true;
        }
    }
    return false;
}

public ArrayList<Move> getValidMoves() {
    ArrayList<Move> validMoves = new ArrayList<>();
    for (Piece piece : this.pieces) {
        List<Position> piecePositions =
piece.getPositions();
        if (piecePositions.isEmpty()) {
            continue;
        }

        for (Direction direction : Direction.values())
        {
            if ((piece.getOrientation() ==
Orientation.HORIZONTAL && (direction == Direction.UP ||
direction == Direction.DOWN)) ||
                (piece.getOrientation() ==
Orientation.VERTICAL && (direction == Direction.LEFT ||
direction == Direction.RIGHT))) {
                continue;
            }

            for (int s = 1; ; s++) {
                Position positionToTest;

                switch (direction) {
                    case UP:
                        Position currentTop =
piecePositions.get(0);
                        positionToTest = new
Position(currentTop.getX(), currentTop.getY() - s);
                        break;
                    case DOWN:
                        Position currentBottom =
piecePositions.get(piecePositions.size() - 1);
                        positionToTest = new
Position(currentBottom.getX(), currentBottom.getY() + s);
                        break;
                    case LEFT:
                        Position currentLeft =
piecePositions.get(0);
                        positionToTest = new
Position(currentLeft.getX() - s, currentLeft.getY());
                        break;
                    case RIGHT:

```



```

        Position currentRight =
piecePositions.get(piecePositions.size() - 1);
        positionToTest = new
Position(currentRight.getX() + s, currentRight.getY());
        break;
        default:
            throw new
IllegalStateException("Arah tidak dikenal: " + direction);
    }

    if (positionToTest.getX() < 0 ||
positionToTest.getX() >= this.width ||
        positionToTest.getY() < 0 ||
positionToTest.getY() >= this.height) {
        break;
    }

    if
(this.grid[positionToTest.getY()][positionToTest.getX()] !=
null) {

        break;
    }

        validMoves.add(new
Move(piece.getId(), direction, s));
    }
}
return validMoves;
}

@Override
public Board clone() {
    ArrayList<Piece> piecesCopy = new ArrayList<>();
    for (Piece p : this.pieces) {
        piecesCopy.add(p.clone());
    }
    Position exitPositionCopy = new
Position(this.exitPosition.getX(), this.exitPosition.getY());
    return new Board(this.width, this.height,
piecesCopy, exitPositionCopy);
}

public List<Piece> getPieces() {
    ArrayList<Piece> piecesCopy = new ArrayList<>();
    for(Piece p : this.pieces){
        piecesCopy.add(p.clone());
    }
    return piecesCopy;
}

```

```

    }

    public Piece getPrimaryPiece() {
        return this.primaryPiece;
    }

    public Piece getPieceById(char id) {
        for (Piece p : this.pieces) {
            if (p.getId() == id) {
                return p;
            }
        }
        return null;
    }

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

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

    public Position getExitPosition() {
        return new Position(this.exitPosition.getX(),
this.exitPosition.getY());
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;
        Board board = (Board) o;

        if (width != board.width || height != board.height)
{
            return false;
        }
        if (!Objects.equals(exitPosition,
board.exitPosition)) {
            return false;
        }

        if (this.pieces == null && board.pieces == null) {
            return false;
        } else if (this.pieces == null || board.pieces ==
null) {
            return false;
        } else {
            if (this.pieces.size() != board.pieces.size())

```

```

{
    return false;
}
    if (!new HashSet<>(this.pieces).equals(new
HashSet<>(board.pieces))) {
        return false;
    }
}
return true;
}

@Override
public int hashCode() {
    Set<Piece> piecesSet = (this.pieces == null) ? null
: new HashSet<>(this.pieces);
    return Objects.hash(width, height, piecesSet,
exitPosition);
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    char[][] charGrid = new char[height][width];
    for(int r=0; r<height; r++) {
        for(int c=0; c<width; c++) {
            charGrid[r][c] = '.';
        }
    }
    for(Piece p : this.pieces) {
        for(Position pos : p.getPositions()) {
            if(pos.getX() >= 0 && pos.getX() < width
&& pos.getY() >=0 && pos.getY() < height) {
                charGrid[pos.getY()][pos.getX()] =
p.getId();
            }
        }
    }
    for(int r=0; r<height; r++) {
        for(int c=0; c<width; c++) {
            sb.append(charGrid[r][c]);
        }
        if (exitPosition.getY() == r &&
exitPosition.getX() == width) sb.append('K');
        if (exitPosition.getY() == r &&
exitPosition.getX() == -1) sb.insert(0, 'K');
        sb.append("\n");
    }
    if (exitPosition.getY() == -1) {
        StringBuilder topExitLine = new
StringBuilder();

```

```

        for (int c=0; c<width; c++)
topExitLine.append(c == exitPosition.getX() ? 'K' : '.');
        sb.insert(0, topExitLine.toString() + "\n");
    }
    if (exitPosition.getY() == height) {
        StringBuilder bottomExitLine = new
StringBuilder();
        for (int c=0; c<width; c++)
bottomExitLine.append(c == exitPosition.getX() ? 'K' : '.');
sb.append(bottomExitLine.toString()).append("\n");
    }
    return sb.toString();
}
}

```

Direction.java

```

package model.core;

public enum Direction {
    UP,
    DOWN,
    LEFT,
    RIGHT;
}

```

Move.java

```

package model.core;

public class Move {
    // Fields
    private final char pieceId;
    private final Direction direction;
    private final int steps;

    // Constructor
    public Move(char pieceId, Direction direction, int steps)
    {
        if (steps < 1) {
            throw new IllegalArgumentException("Jumlah
langkah (steps) harus minimal 1.");
        }
        this.pieceId = pieceId;
    }
}

```

```

        this.direction = direction;
        this.steps = steps;
    }

    // Methods
    public char getPieceId() {
        return this.pieceId;
    }

    public Direction getDirection() {
        return this.direction;
    }

    public int getSteps() {
        return this.steps;
    }

    @Override
    public String toString() {
        return "Move{" +
            "pieceId=" + pieceId +
            ", direction=" + direction +
            ", steps=" + steps +
            '}';
    }
}

```

Orientation.java

```

package model.core;

public enum Orientation {
    HORIZONTAL,
    VERTICAL;
}

```

Piece.java

```

package model.core;

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

```

```

public class Piece {
    private final char id;
    private final ArrayList <Position> positions;
    private final Orientation orientation;
    private final boolean isPrimary;

    public Piece(char id, ArrayList<Position> positions,
Orientation orientation, boolean isPrimary) {
        this.id = id;
        this.positions = new ArrayList<>();
        for (Position p : positions) {
            this.positions.add(new Position(p.getX(),
p.getY()));
        }
        this.orientation = orientation;
        this.isPrimary = isPrimary;
    }

    public void moveBySteps(Direction direction, int steps)
{
        if (steps == 0) {
            return;
        }

        int dx = 0, dy = 0;
        switch (direction) {
            case UP:
                dy = -steps;
                break;
            case DOWN:
                dy = steps;
                break;
            case LEFT:
                dx = -steps;
                break;
            case RIGHT:
                dx = steps;
                break;
        }

        for (int i = 0; i < positions.size(); i++) {
            Position p = positions.get(i);
            positions.set(i, new Position(p.getX() + dx,
p.getY() + dy));
        }
    }

    public boolean canMove(Direction direction, Board board)
{

```

```

        Position edge;
        switch (direction) {
            case UP:
                if (orientation != Orientation.VERTICAL)
return false;

                if (positions.isEmpty()) return false;
                edge = positions.get(0);
                return edge.getY() > 0 &&
                    !board.isOccupied(new
Position(edge.getX(), edge.getY() - 1));

            case DOWN:
                if (orientation != Orientation.VERTICAL)
return false;

                if (positions.isEmpty()) return false;
                edge = positions.get(positions.size() -
1);

                return edge.getY() < board.getHeight() -
1 &&
                    !board.isOccupied(new
Position(edge.getX(), edge.getY() + 1));

            case LEFT:
                if (orientation !=
Orientation.HORIZONTAL) return false;
                if (positions.isEmpty()) return false;
                edge = positions.get(0);
                return edge.getX() > 0 &&
                    !board.isOccupied(new
Position(edge.getX() - 1, edge.getY()));

            case RIGHT:
                if (orientation !=
Orientation.HORIZONTAL) return false;
                if (positions.isEmpty()) return false;
                edge = positions.get(positions.size() -
1);

                return edge.getX() < board.getWidth() - 1
&&
                    !board.isOccupied(new
Position(edge.getX() + 1, edge.getY()));

            default:
                return false;
        }
    }

    @Override
    public Piece clone() {
        ArrayList<Position> positionsCopy = new

```

```

ArrayList<>();
    for (Position p : this.positions) {
        positionsCopy.add(new Position(p.getX(),
p.getY()));
    }
    return new Piece(this.id, positionsCopy,
this.orientation, this.isPrimary);
}

public List<Position> getPositions() {
    return new ArrayList<>(this.positions);
}

public Orientation getOrientation() {
    return this.orientation;
}

public boolean isPrimary() {
    return this.isPrimary;
}

public char getId() {
    return this.id;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return
false;
    Piece piece = (Piece) o;
    return id == piece.id &&
        isPrimary == piece.isPrimary &&
        orientation == piece.orientation &&
        Objects.equals(positions, piece.positions);
}

@Override
public int hashCode() {
    return Objects.hash(id, positions, orientation,
isPrimary);
}
}

```

Position.java

```
package model.core;
```



```

import java.util.Objects;
public class Position {
    private int x;
    private int y;

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

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;

        if (o == null || getClass() != o.getClass()) return
false;

        Position position = (Position) o;

        return position.getX() == this.x && position.getY()
== this.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.x, this.y);
    }
}

```

SolutionResult.java

```

package controller;

import java.util.ArrayList;
import java.util.List;
import model.core.Board;
import model.core.Move;

public class SolutionResult {
    private final boolean solved;

```

```

        private final ArrayList<Move> moves;
        private final int visitedNodesCount;
        private final long executionTime;
        private final ArrayList<Board> boardStates;

        public SolutionResult(boolean solved, List<Move> moves,
int visitedNodesCount, long executionTime, List<Board>
boardStates) {
            this.solved = solved;
            this.moves = new ArrayList<>(moves);
            this.visitedNodesCount = visitedNodesCount;
            this.executionTime = executionTime;
            this.boardStates = new ArrayList<>(boardStates);
        }

        public boolean isSolved() {
            return solved;
        }

        public ArrayList<Move> getMoves() {
            return this.moves;
        }

        public int getVisitedNodesCount() {
            return this.visitedNodesCount;
        }

        public long getExecutionTime() {
            return this.executionTime;
        }

        public List<Board> getBoardStates() {
            return this.boardStates;
        }
    }
}

```

UCSolver.java

```

package controller.solver;

import controller.SolutionResult;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;
import model.GameState;

```

```

import model.core.Board;

public class UCSolver extends RushHourSolver {
    public UCSolver(GameState initialState) {
        super(initialState);
    }

    @Override
    public SolutionResult solve() {
        long startTime = System.nanoTime();
        this.visitedNodesCount = 0;

        PriorityQueue<GameState> frontier = new
PriorityQueue<>(Comparator.comparingInt(GameState::getCost));
        Set<Board> visitedBoards = new HashSet<>();

        if (this.initialState == null ||
this.initialState.getBoard() == null) {
            System.err.println("UCSolver Error: Initial
state or initial board is null.");
            this.executionTime = System.nanoTime() -
startTime;
            return new SolutionResult(false, new
ArrayList<>(), 0, this.executionTime, new ArrayList<>());
        }

        frontier.add(this.initialState);
        visitedBoards.add(this.initialState.getBoard());

        int nodesProcessedSinceLastPublish = 0;
        final int PUBLISH_INTERVAL_NODES = 500;

        while (!frontier.isEmpty()) {
            GameState current = frontier.poll();
            this.visitedNodesCount++;
            nodesProcessedSinceLastPublish++;

            if (nodesProcessedSinceLastPublish >=
PUBLISH_INTERVAL_NODES) {
                publishVisitedNodesCount();
                nodesProcessedSinceLastPublish = 0;
            }

            if (current.isGoalState()) {
                this.executionTime = System.nanoTime() -
startTime;

                if (nodesProcessedSinceLastPublish > 0) {
                    publishVisitedNodesCount();
                }
                List<Board> boardStatesPath =

```

```

reconstructBoardStates(current);
        return new SolutionResult(true,
current.getMoves(), this.visitedNodesCount,
this.executionTime, boardStatesPath);
    }

    List<GameState> successors =
current.getSuccessors();
    for (GameState successor : successors) {
        if
(!visitedBoards.contains(successor.getBoard())) {
visitedBoards.add(successor.getBoard());
        frontier.add(successor);
    }
    }

    this.executionTime = System.nanoTime() - startTime;
    if (nodesProcessedSinceLastPublish > 0) {
        publishVisitedNodesCount();
    }
    return new SolutionResult(false, new ArrayList<>(),
this.visitedNodesCount, this.executionTime, new
ArrayList<>());
}

@Override
public String getName() {
    return "Uniform Cost Search Solver";
}
}

```

GreedyBestFirstSearch.java

```

package controller.solver;

import controller.SolutionResult;
import controller.heuristic.*;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;
import model.GameState;
import model.core.Board;

```

```

public class GreedyBestFirstSearchSolver extends
RushHourSolver{
    private final Heuristic heuristic;

    public GreedyBestFirstSearchSolver(GameState
initialState, Heuristic heuristic) {
        super(initialState);
        this.heuristic = heuristic;
    }

    @Override
    public SolutionResult solve() {
        long startTime = System.nanoTime();
        this.visitedNodesCount = 0;

        PriorityQueue<GameState> frontier = new
PriorityQueue<>(Comparator.comparingInt (GameState::gethCost))
;
        Set<Board> visitedBoards = new HashSet<>();

        if (this.initialState == null ||
this.initialState.getBoard() == null) {
            System.err.println("GBFSolver Error: Initial
state or initial board is null.");
            this.executionTime = System.nanoTime() -
startTime;
            return new SolutionResult(false, new
ArrayList<>(), 0, this.executionTime, new ArrayList<>());
        }

        frontier.add(this.initialState);
        visitedBoards.add(this.initialState.getBoard());

        int nodesProcessedSinceLastPublish = 0;
        final int PUBLISH_INTERVAL_NODES = 500;

        while(!frontier.isEmpty()) {
            GameState currentGameState = frontier.poll();
            this.visitedNodesCount++;
            nodesProcessedSinceLastPublish++;

            if (nodesProcessedSinceLastPublish >=
PUBLISH_INTERVAL_NODES) {
                publishVisitedNodesCount();
                nodesProcessedSinceLastPublish = 0;
            }

            if (currentGameState.isGoalState()) {
                this.executionTime = System.nanoTime() -
startTime;

```

```

        if (nodesProcessedSinceLastPublish > 0) {
            publishVisitedNodesCount();
        }
        List<Board> boardStatesPath =
reconstructBoardStates(currentGameState);
        return new SolutionResult(true,
currentGameState.getMoves(), this.visitedNodesCount,
this.executionTime, boardStatesPath);
    }

    List<GameState> successors =
currentGameState.getSuccessors();
    for (GameState successor : successors) {
        if
(!visitedBoards.contains(successor.getBoard())) {
            int hCostSuccessor =
heuristic.calculate(successor);
            if (hCostSuccessor == Integer.MAX_VALUE)
{
                continue;
            }
            successor.setHeuristicCosts(hCostSuccessor);
            visitedBoards.add(successor.getBoard());
            frontier.add(successor);
        }
    }

    this.executionTime = System.nanoTime() - startTime;
    if (nodesProcessedSinceLastPublish > 0) {
        publishVisitedNodesCount();
    }
    return new SolutionResult(false, new ArrayList<>(),
this.visitedNodesCount, this.executionTime, new
ArrayList<>());
}

@Override
public String getName() {
    return "Greedy Best First Search Solver";
}
}

```

BeamSearchSolver.java

```
package controller.solver;
```

```

import controller.SolutionResult;
import controller.heuristic.Heuristic;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;
import model.GameState;
import model.core.Board;

public class BeamSearchSolver extends RushHourSolver {
    private final Heuristic heuristic;
    private final int beamWidth;

    private int nodesProcessedSinceLastPublish = 0;
    private static final int PUBLISH_INTERVAL_NODES = 500;

    public BeamSearchSolver(GameState initialState,
Heuristic heuristic, int beamWidth) {
        super(initialState);
        this.heuristic = heuristic;
        this.beamWidth = beamWidth > 0 ? beamWidth : 1;
    }

    public BeamSearchSolver(GameState initialState,
Heuristic heuristic) {
        this(initialState, heuristic, 10);
    }

    @Override
    public SolutionResult solve() {
        long startTime = System.nanoTime();
        this.visitedNodesCount = 0;
        this.nodesProcessedSinceLastPublish = 0;

        if (this.initialState == null ||
this.initialState.getBoard() == null) {
            System.err.println("BeamSearchSolver Error:
Initial state atau initial board adalah null.");
            this.executionTime = System.nanoTime() -
startTime;
            return new SolutionResult(false, new
ArrayList<>(), 0, this.executionTime, new ArrayList<>());
        }

        if (this.heuristic == null) {
            System.err.println("BeamSearchSolver Error:

```

```

Heuristic tidak boleh null.");
        this.executionTime = System.nanoTime() -
startTime;
        return new SolutionResult(false, new
ArrayList<>(), 0, this.executionTime, new ArrayList<>());
    }

    List<GameState> currentBeam = new ArrayList<>();
    int initialHCost =
heuristic.calculate(this.initialState);
    if (initialHCost == Integer.MAX_VALUE) {
        this.executionTime = System.nanoTime() -
startTime;
        publishFinalProgress();
        return new SolutionResult(false, new
ArrayList<>(), this.visitedNodesCount, this.executionTime,
new ArrayList<>());
    }
    this.initialState.setHeuristicCosts(initialHCost);
    currentBeam.add(this.initialState);

    Set<Board> visitedBoards = new HashSet<>();
    visitedBoards.add(this.initialState.getBoard());

    while (!currentBeam.isEmpty()) {
        PriorityQueue<GameState> candidates = new
PriorityQueue<>(Comparator.comparingInt(GameState::gethCost))
;

        for (GameState currentState : currentBeam) {
            this.visitedNodesCount++;
            this.nodesProcessedSinceLastPublish++;

            if (nodesProcessedSinceLastPublish >=
PUBLISH_INTERVAL_NODES) {
                publishVisitedNodesCount();
                nodesProcessedSinceLastPublish = 0;
            }

            if (currentState.isGoalState()) {
                this.executionTime =
System.nanoTime() - startTime;
                publishFinalProgress();
                List<Board> boardStatesPath =
reconstructBoardStates(currentState);
                return new SolutionResult(true,
currentState.getMoves(), this.visitedNodesCount,
this.executionTime, boardStatesPath);
            }
        }
    }

```



```

        List<GameState> successors =
currentState.getSuccessors();
        for (GameState successor : successors) {
            if
(!visitedBoards.contains(successor.getBoard())) {
                int hCostSuccessor =
heuristic.calculate(successor);
                if (hCostSuccessor ==
Integer.MAX_VALUE) {
                    continue;
                }

successor.setHeuristicCosts(hCostSuccessor);

                candidates.add(successor);

visitedBoards.add(successor.getBoard());
            }
        }

        currentBeam.clear();
        int count = 0;
        while (!candidates.isEmpty() && count <
this.beamWidth) {
            currentBeam.add(candidates.poll());
            count++;
        }

        if (currentBeam.isEmpty() &&
!candidates.isEmpty()) {
        }

        this.executionTime = System.nanoTime() - startTime;
        publishFinalProgress();
        return new SolutionResult(false, new ArrayList<>(),
this.visitedNodesCount, this.executionTime, new
ArrayList<>());
    }

    private void publishFinalProgress() {
        if (nodesProcessedSinceLastPublish > 0) {
            publishVisitedNodesCount();
            nodesProcessedSinceLastPublish = 0;
        }
    }

    @Override
    public String getName() {

```

```

        return "Beam Search Solver (w=" + this.beamWidth +
") with " + (heuristic != null ? heuristic.getName() : "No
Heuristic");
    }
}

```

AStarSolver.java

```

package controller.solver;

import controller.SolutionResult;
import controller.heuristic.Heuristic;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;
import model.GameState;
import model.core.Board;

public class AStarSolver extends RushHourSolver {
    private final Heuristic heuristic;

    public AStarSolver(GameState initialState, Heuristic
heuristic) {
        super(initialState);
        this.heuristic = heuristic;
    }

    @Override
    public SolutionResult solve() {
        long startTime = System.nanoTime();
        this.visitedNodesCount = 0;

        PriorityQueue<GameState> frontier = new
PriorityQueue<>(Comparator.comparingInt (GameState::getfCost))
;
        Set<Board> visitedBoards = new HashSet<>();

        if (this.initialState == null ||
this.initialState.getBoard() == null) {
            System.err.println("GBFSolver Error: Initial
state or initial board is null.");
            this.executionTime = System.nanoTime() -
startTime;
            return new SolutionResult(false, new
ArrayList<>(), 0, this.executionTime, new ArrayList<>());

```

```

    }

    frontier.add(this.initialState);
    visitedBoards.add(this.initialState.getBoard());

    int nodesProcessedSinceLastPublish = 0;
    final int PUBLISH_INTERVAL_NODES = 500;

    while(!frontier.isEmpty()) {
        GameState currentGameState = frontier.poll();
        this.visitedNodesCount++;
        nodesProcessedSinceLastPublish++;

        if (nodesProcessedSinceLastPublish >=
PUBLISH_INTERVAL_NODES) {
            publishVisitedNodesCount();
            nodesProcessedSinceLastPublish = 0;
        }

        if (currentGameState.isGoalState()) {
            this.executionTime = System.nanoTime() -
startTime;
            if (nodesProcessedSinceLastPublish > 0) {
                publishVisitedNodesCount();
            }
            List<Board> boardStatesPath =
reconstructBoardStates(currentGameState);
            return new SolutionResult(true,
currentGameState.getMoves(), this.visitedNodesCount,
this.executionTime, boardStatesPath);
        }

        List<GameState> successors =
currentGameState.getSuccessors();
        for (GameState successor : successors) {
            if
(!visitedBoards.contains(successor.getBoard())) {
                int hCostSuccessor =
heuristic.calculate(successor);
                if (hCostSuccessor == Integer.MAX_VALUE)
{
                    continue;
                }

                successor.setHeuristicCosts(hCostSuccessor);
                visitedBoards.add(successor.getBoard());
                frontier.add(successor);
            }
        }
    }

```

```

        }

        this.executionTime = System.nanoTime() - startTime;
        if (nodesProcessedSinceLastPublish > 0) {
            publishVisitedNodesCount();
        }
        return new SolutionResult(false, new ArrayList<>(),
this.visitedNodesCount, this.executionTime, new
ArrayList<>());
    }

    @Override
    public String getName() {
        return "A* Solver with " + (heuristic != null ?
heuristic.getName() : "No Heuristic");
    }
}

```

Heuristic.java

```

package controller.heuristic;

import model.GameState;

public interface Heuristic {
    public int calculate(GameState gameState);
    public String getName();
}

```

ManhattanDistance.java

```

package controller.heuristic;

import java.util.ArrayList;

import model.GameState;
import model.core.*;

public class ManhattanDistance implements Heuristic{
    // Fields

    // Constructor
    public ManhattanDistance() { }

    // Methods
}

```

```

@Override
public int calculate(GameState gameState){
    Board board = gameState.getBoard();
    ArrayList<Position> pPos = new ArrayList<>();

    for (Piece piece : gameState.getBoard().getPieces())
    {
        if (piece.getId() == 'P') {
            pPos.addAll(piece.getPositions());
        }
    }

    if (pPos.isEmpty()) return 0;

    int result = Integer.MAX_VALUE;
    for (Position p : pPos) {
        int distance = Math.abs(p.getX() -
board.getExitPosition().getX()) + Math.abs(p.getY() -
board.getExitPosition().getY());
        result = Math.min(result, distance);
    }
    return result;
}

@Override
public String getName() {
    return "Manhattan Distance Heuristic";
}
}

```

BlockingPieceHeuristic.java

```

package controller.heuristic;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import model.GameState;
import model.core.Board;
import model.core.Orientation;
import model.core.Piece;
import model.core.Position;

public class BlockingPiecesHeuristic implements Heuristic {

    public BlockingPiecesHeuristic() {

    }
}

```

```

@Override
public int calculate(GameState gameState) {
    Board board = gameState.getBoard();
    if (board == null) {
        return Integer.MAX_VALUE;
    }

    Piece primaryPiece = board.getPrimaryPiece();
    if (primaryPiece == null) {
        return Integer.MAX_VALUE;
    }

    Position exitPos = board.getExitPosition();
    if (exitPos == null) {
        return Integer.MAX_VALUE;
    }

    Set<Character> blockingPieceIds = new HashSet<>();
    List<Piece> allPieces = board.getPieces();

    if (primaryPiece.getOrientation() ==
Orientation.HORIZONTAL) {
        int pieceRow =
primaryPiece.getPositions().get(0).getY();
        int primaryPieceMinX = Integer.MAX_VALUE;
        int primaryPieceMaxX = Integer.MIN_VALUE;

        for (Position pos :
primaryPiece.getPositions()) {
            if (pos.getX() < primaryPieceMinX) {
                primaryPieceMinX = pos.getX();
            }
            if (pos.getX() > primaryPieceMaxX) {
                primaryPieceMaxX = pos.getX();
            }
        }

        int pathStartX, pathEndX;
        if (exitPos.getX() > primaryPieceMaxX) {
            pathStartX = primaryPieceMaxX + 1;
            pathEndX = board.getWidth() - 1;
        } else if (exitPos.getX() < primaryPieceMinX)
{
            pathStartX = primaryPieceMinX - 1;
            pathEndX = 0;
        } else {
            return 0;
        }
    }
}

```

```

        for (int x = Math.min(pathStartX, pathEndX); x
<= Math.max(pathStartX, pathEndX); x++) {
            Position currentPathPos = new Position(x,
pieceRow);

            for (Piece otherPiece : allPieces) {
                if (otherPiece.isPrimary()) {
                    continue;
                }
                if
(otherPiece.getPositions().contains(currentPathPos)) {
blockingPieceIds.add(otherPiece.getId());
                }
            }
        }

        } else {
            int pieceCol =
primaryPiece.getPositions().get(0).getX();
            int primaryPieceMinY = Integer.MAX_VALUE;
            int primaryPieceMaxY = Integer.MIN_VALUE;

            for (Position pos :
primaryPiece.getPositions()) {
                if (pos.getY() < primaryPieceMinY) {
                    primaryPieceMinY = pos.getY();
                }
                if (pos.getY() > primaryPieceMaxY) {
                    primaryPieceMaxY = pos.getY();
                }
            }

            int pathStartY, pathEndY;
            if (exitPos.getY() > primaryPieceMaxY) {
                pathStartY = primaryPieceMaxY + 1;
                pathEndY = board.getHeight() - 1;
            } else if (exitPos.getY() < primaryPieceMinY)
{
                pathStartY = primaryPieceMinY - 1;
                pathEndY = 0;
            } else {
                return 0;
            }

            for (int y = Math.min(pathStartY, pathEndY); y
<= Math.max(pathStartY, pathEndY); y++) {
                Position currentPathPos = new
Position(pieceCol, y);
                for (Piece otherPiece : allPieces) {
                    if (otherPiece.isPrimary()) {

```

```

        continue;
    }
    if
(otherPiece.getPositions().contains(currentPathPos)) {
blockingPieceIds.add(otherPiece.getId());
    }
    }
    }
    return blockingPieceIds.size();
}

@Override
public String getName() {
    return "Blocking Pieces Heuristic";
}
}

```


Input dan Output

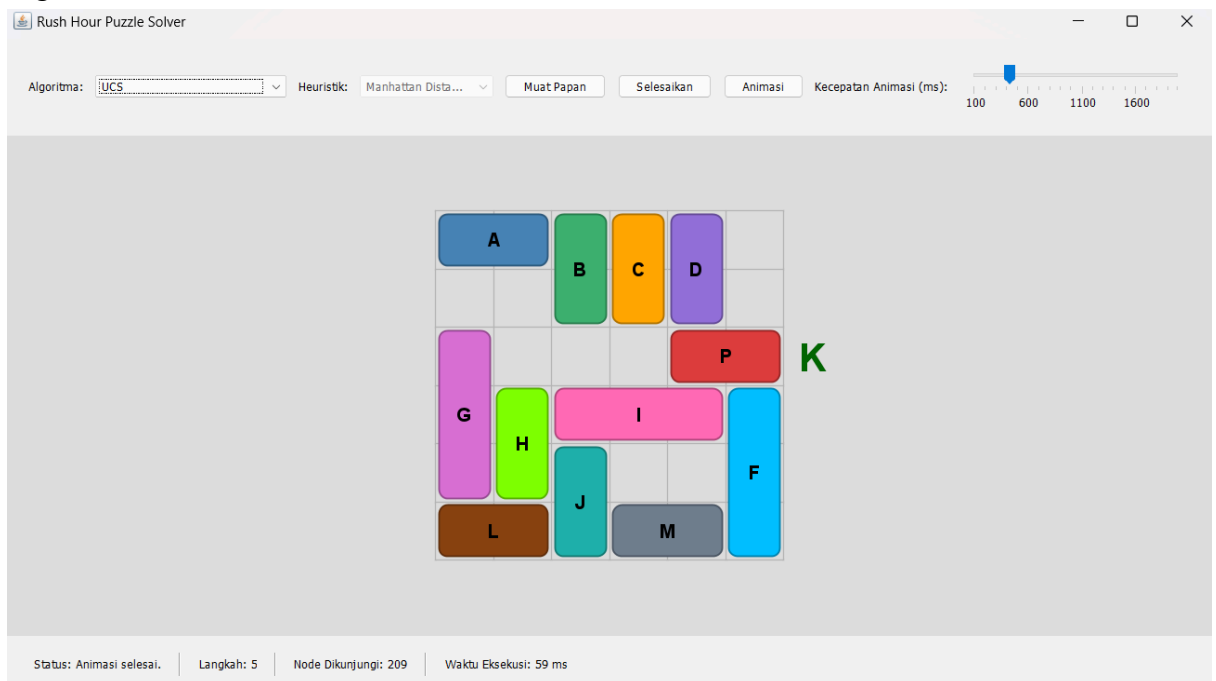
1. Input

```
6 6
11
AAB...F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

Output

Algoritma: UCS

Heuristic: -

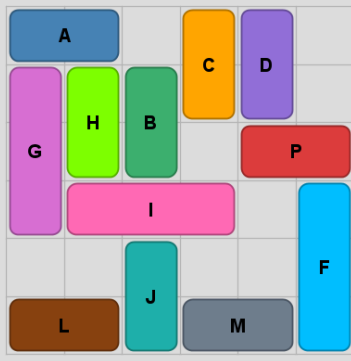


Algoritma: Greedy Best First Search

Heuristic: Manhattan Distance

Rush Hour Puzzle Solver

Algoritma: Heuristik: Muat Papan Animasi Kecepatan Animasi (ms):



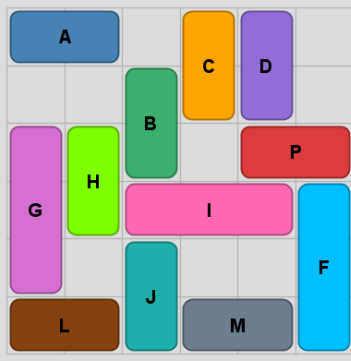
Status: Animasi selesai. Langkah: 13 Node Dikunjungi: 31 Waktu Eksekusi: 7 ms

Algoritma: Greedy Best First Search

Heuristic: Blocking Pieces

Rush Hour Puzzle Solver

Algoritma: Heuristik: Muat Papan Animasi Kecepatan Animasi (ms):



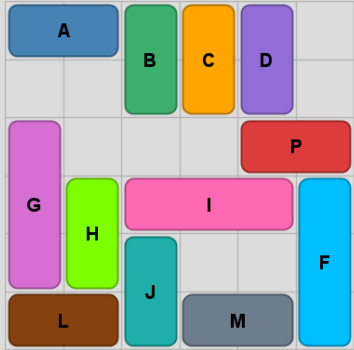
Status: Animasi selesai. Langkah: 15 Node Dikunjungi: 75 Waktu Eksekusi: 11 ms

Algoritma: A*

Heuristic: Manhattan Distance

Rush Hour Puzzle Solver

Algoritma: Heuristik: Kecepatan Animasi (ms):



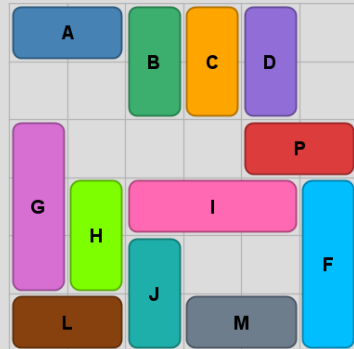
Status: Animasi selesai. | Langkah: 6 | Node Dikunjungi: 130 | Waktu Eksekusi: 29 ms

Algoritma: A*

Heuristic: Blocking Pieces

Rush Hour Puzzle Solver

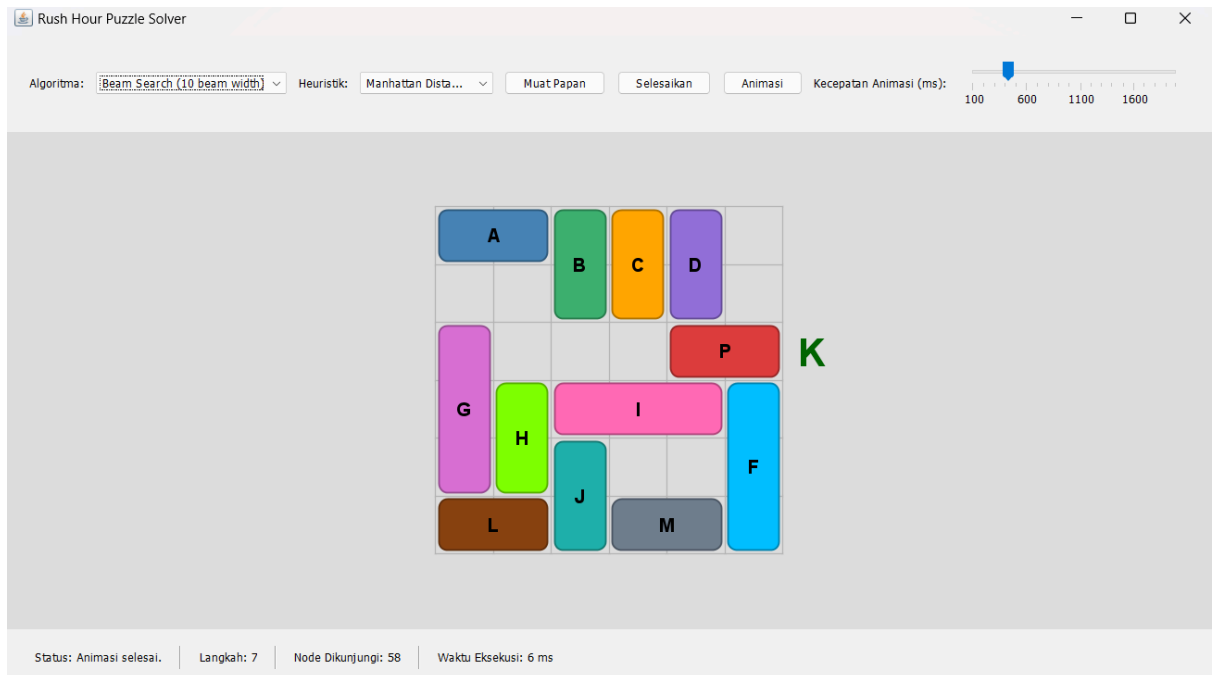
Algoritma: Heuristik: Kecepatan Animasi (ms):



Status: Animasi selesai. | Langkah: 5 | Node Dikunjungi: 76 | Waktu Eksekusi: 15 ms

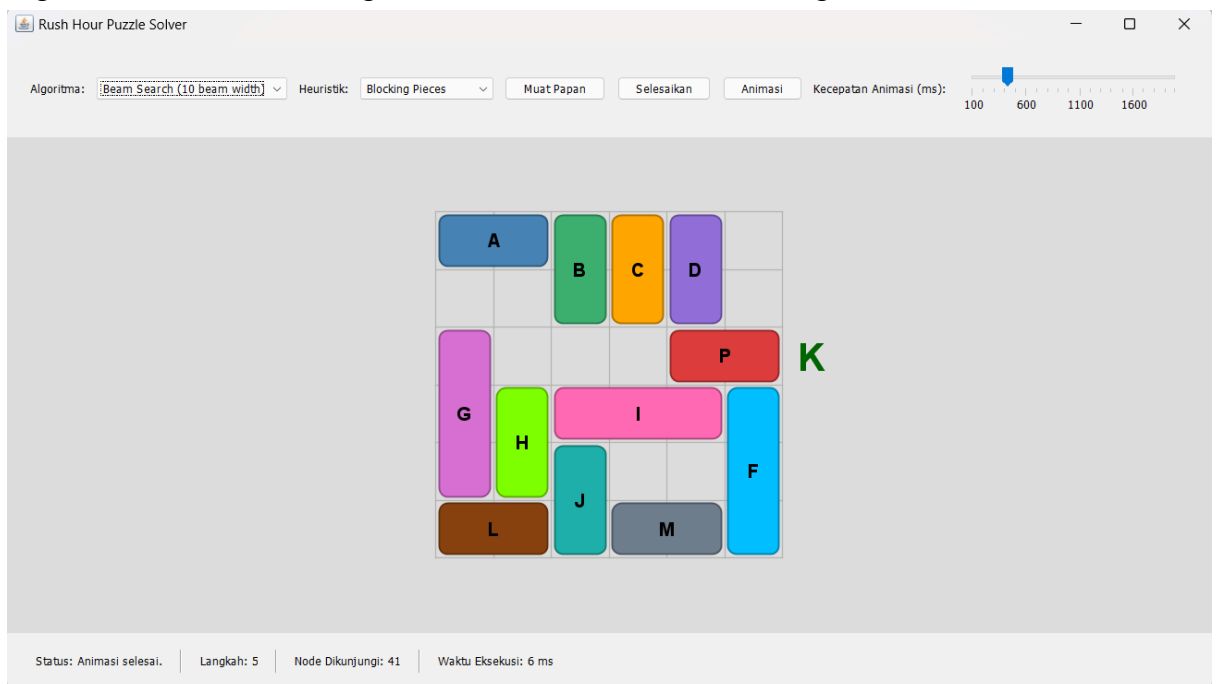
Algoritma: Beam Search Algorithm

Heuristic: Manhattan Distance



Algoritma: Beam Search Algorithm

Heuristic: Blocking Pieces



2. Input

```

7 7
12
    K
CAAA.F.
CM.BBF.
    
```

```

CMD..F.
HMDPEEE
H.DP..G
..JJ..G
II..LLL

```

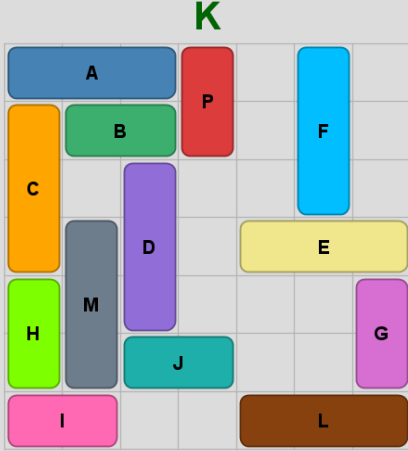
Output

Algoritma: UCS

Heuristic: -

Rush Hour Puzzle Solver

Algoritma: Heuristik: Muat Papan Selesaikan Animasi Kecepatan Animasi (ms):



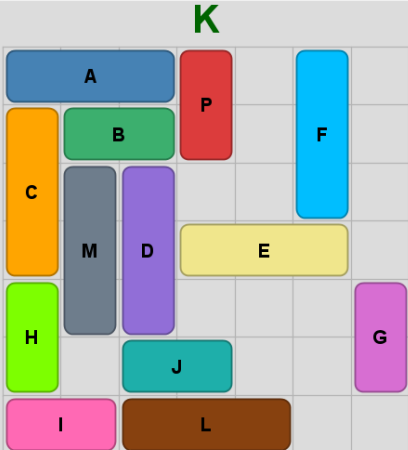
Status: Animasi selesai. Langkah: 6 Node Dikunjungi: 6406 Waktu Eksekusi: 3510 ms

Algoritma: Greedy Best First Search

Heuristic: Manhattan Distance

Rush Hour Puzzle Solver

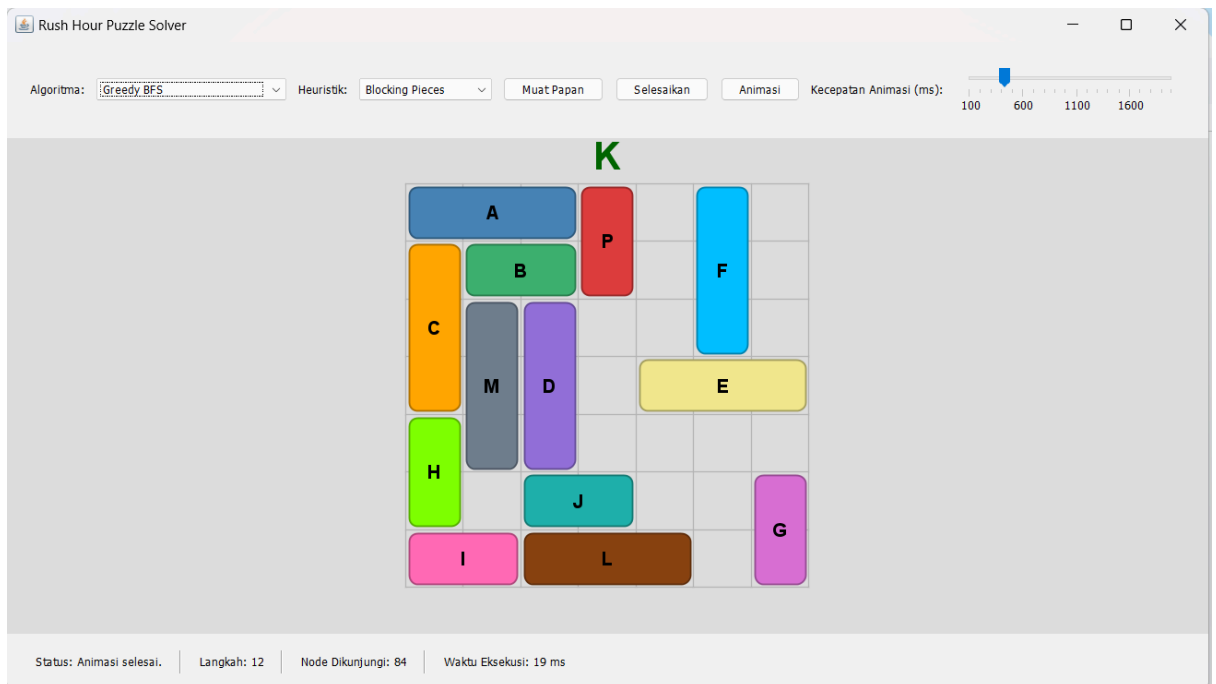
Algoritma: Heuristik: Muat Papan Selesaikan Animasi Kecepatan Animasi (ms):



Status: Animasi selesai. Langkah: 17 Node Dikunjungi: 344 Waktu Eksekusi: 121 ms

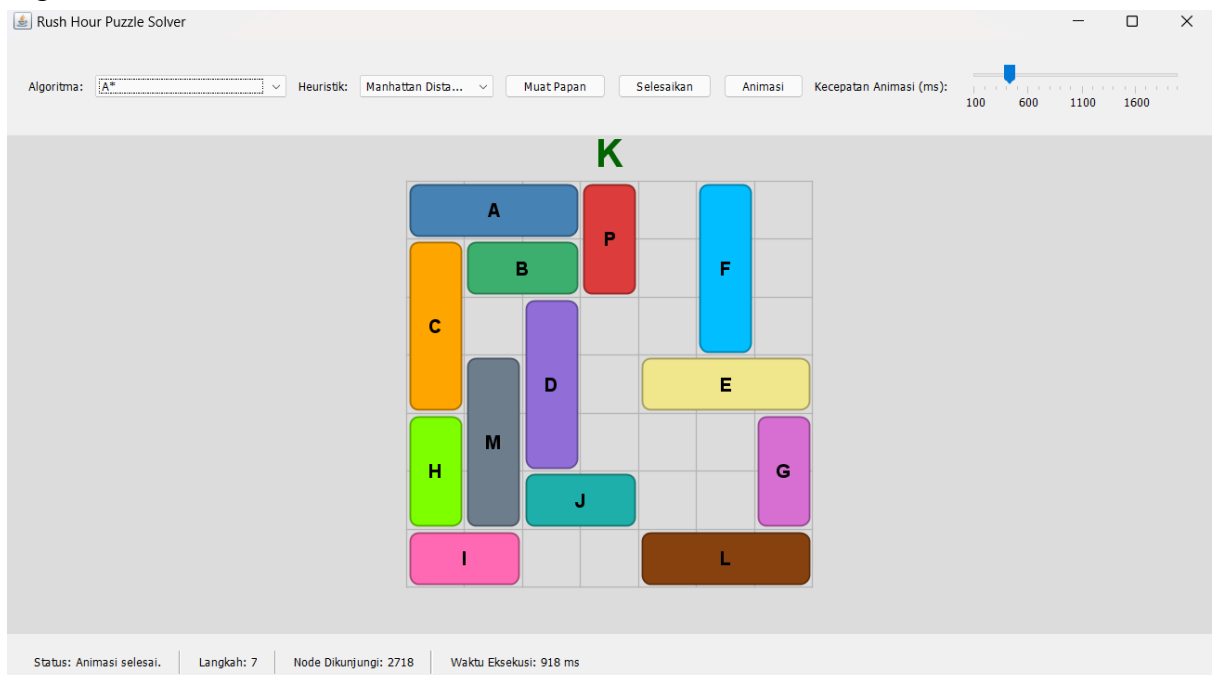
Algoritma: Greedy Best First Search

Heuristic: Blocking Pieces



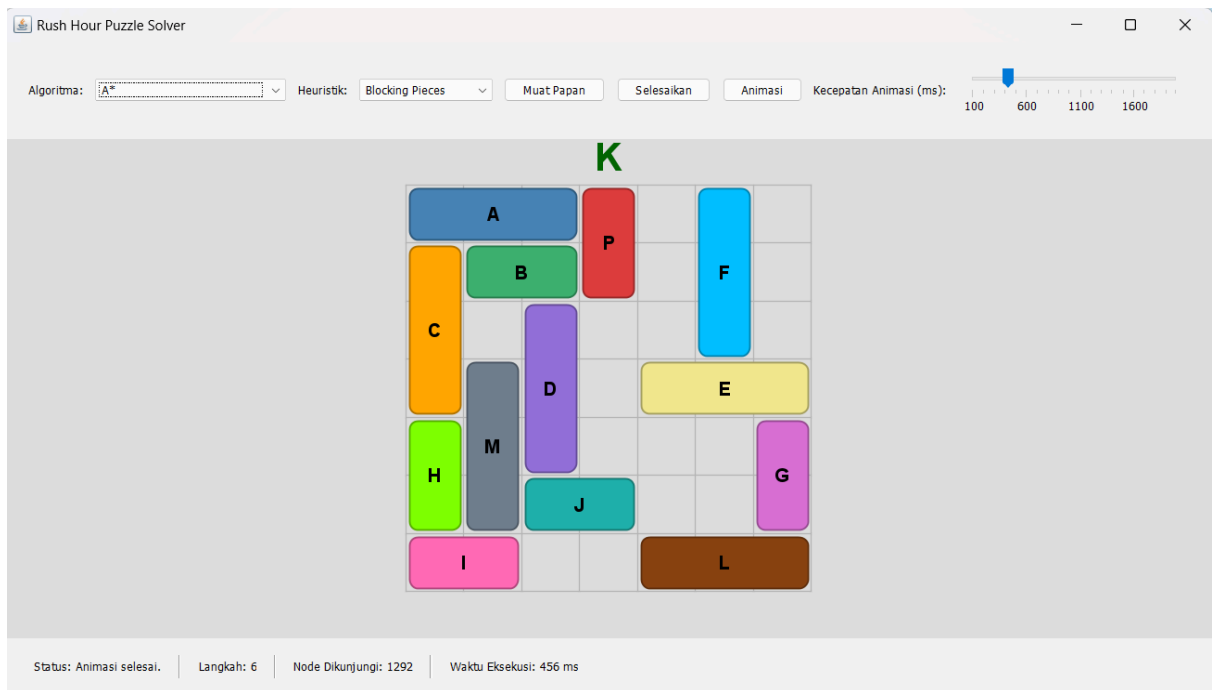
Algoritma: A*

Heuristic: Manhattan Distance



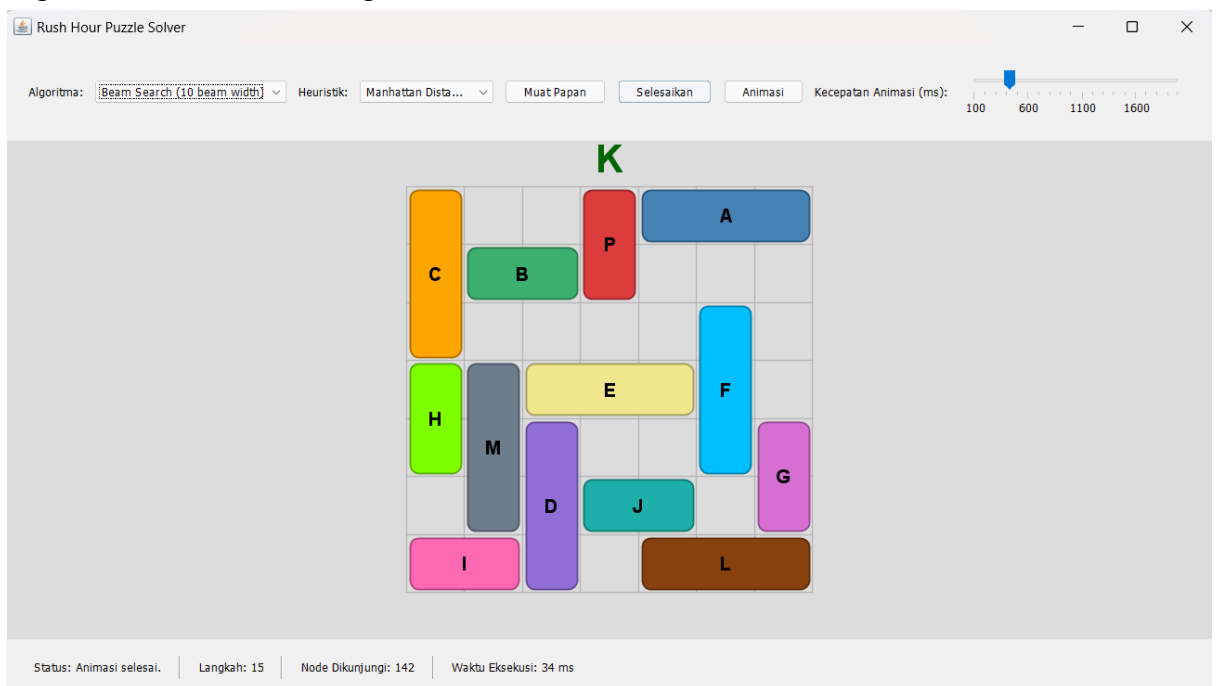
Algoritma: A*

Heuristic: Blocking Pieces



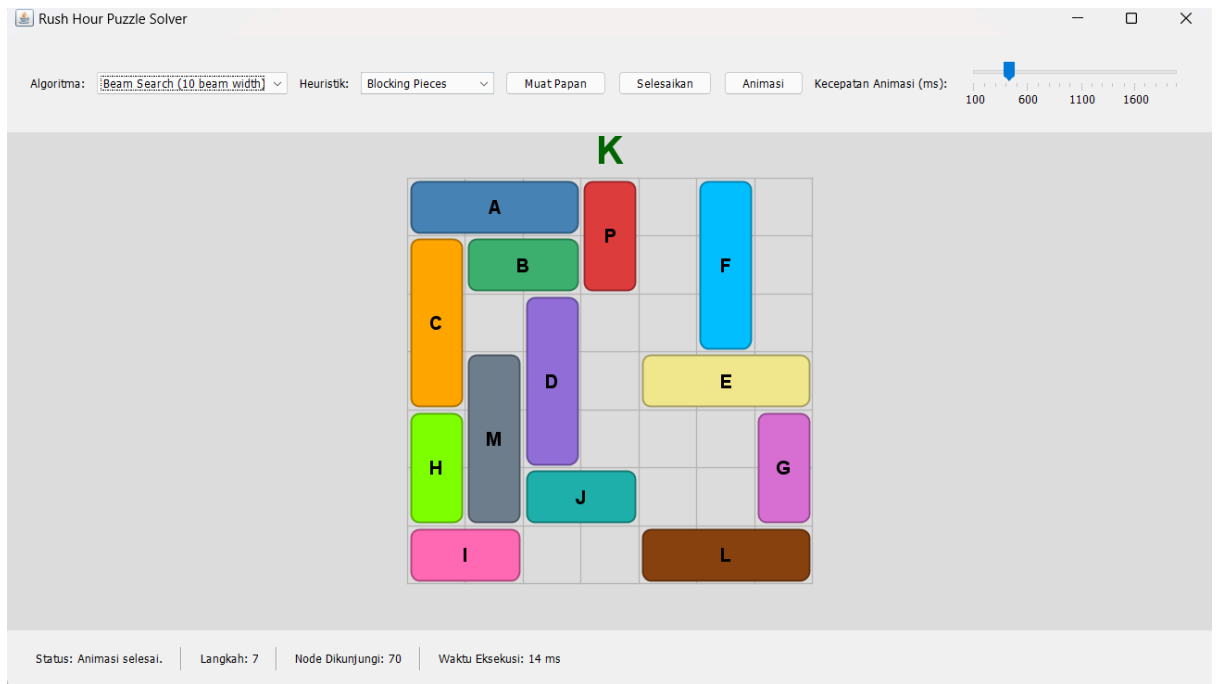
Algoritma: Beam Search Algorithm

Heuristic: Manhattan Distance



Algoritma: Beam Search Algorithm

Heuristic: Blocking Pieces



3. Input

```
6 6
12
GBB.L.
GHI.LM
KGIIPPM
CCCA.M
..JADD
EEJFF.
```


Output

Algoritma: UCS

Heuristic: -

Rush Hour Puzzle Solver

Algoritma: Heuristik: Muat Papan Selesaikan Animasi Kecepatan Animasi (ms):

Status: Animasi selesai. Langkah: 51 Node Dikunjungi: 3029 Waktu Eksekusi: 831 ms

Algoritma: Greedy Best First Search

Heuristic: Manhattan Distance

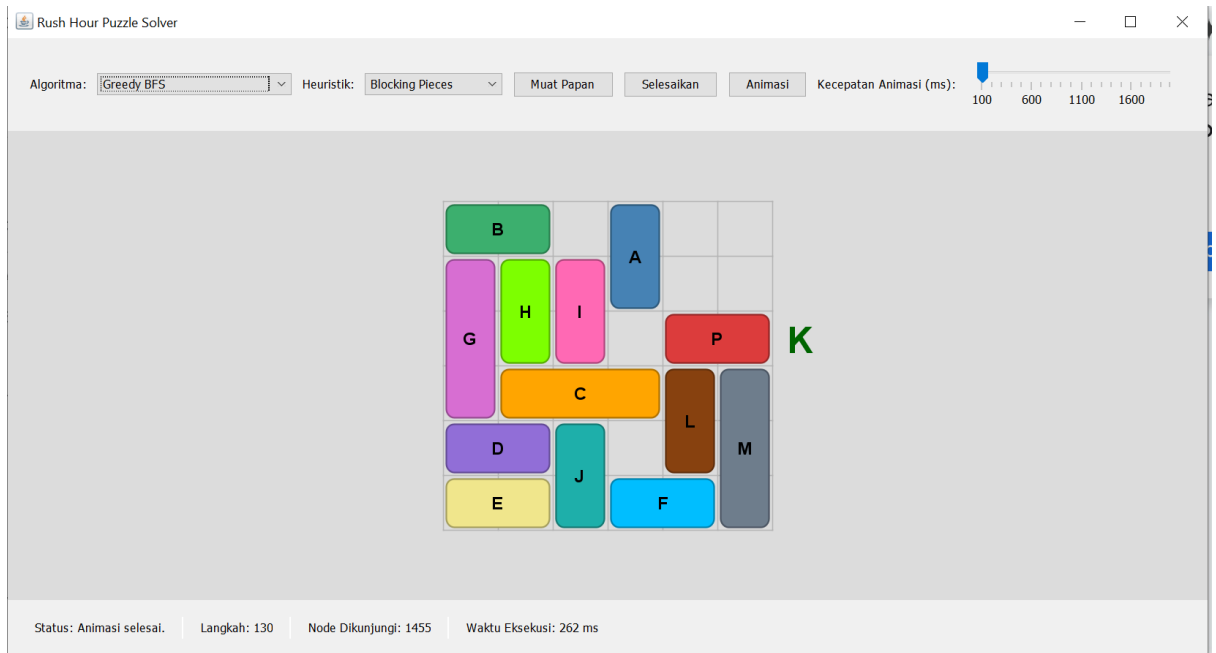
Rush Hour Puzzle Solver

Algoritma: Heuristik: Muat Papan Selesaikan Animasi Kecepatan Animasi (ms):

Status: Animasi selesai. Langkah: 128 Node Dikunjungi: 1552 Waktu Eksekusi: 303 ms

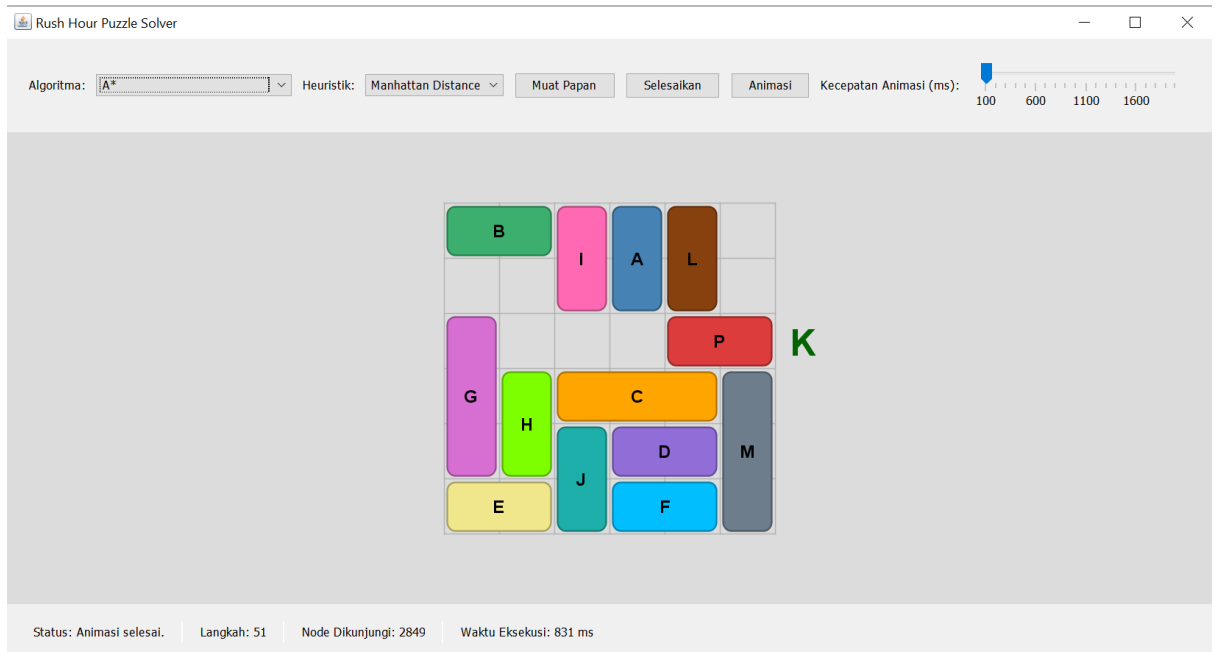
Algoritma: Greedy Best First Search

Heuristic: Blocking Pieces



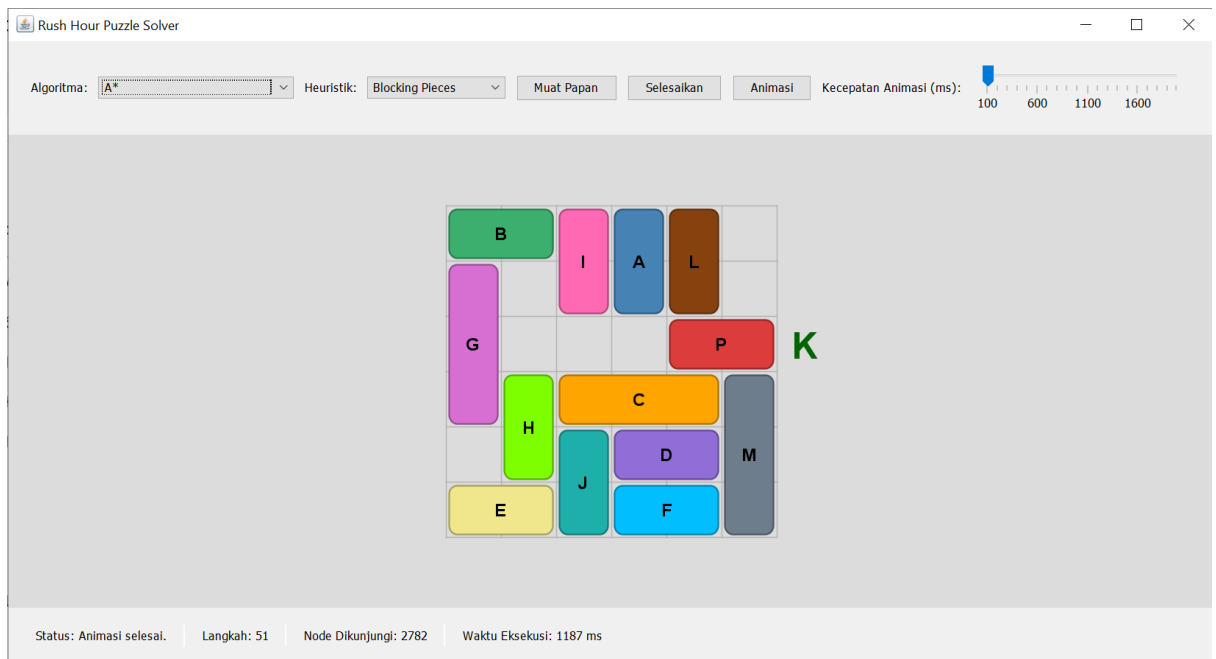
Algoritma: A*

Heuristic: Manhattan Distance



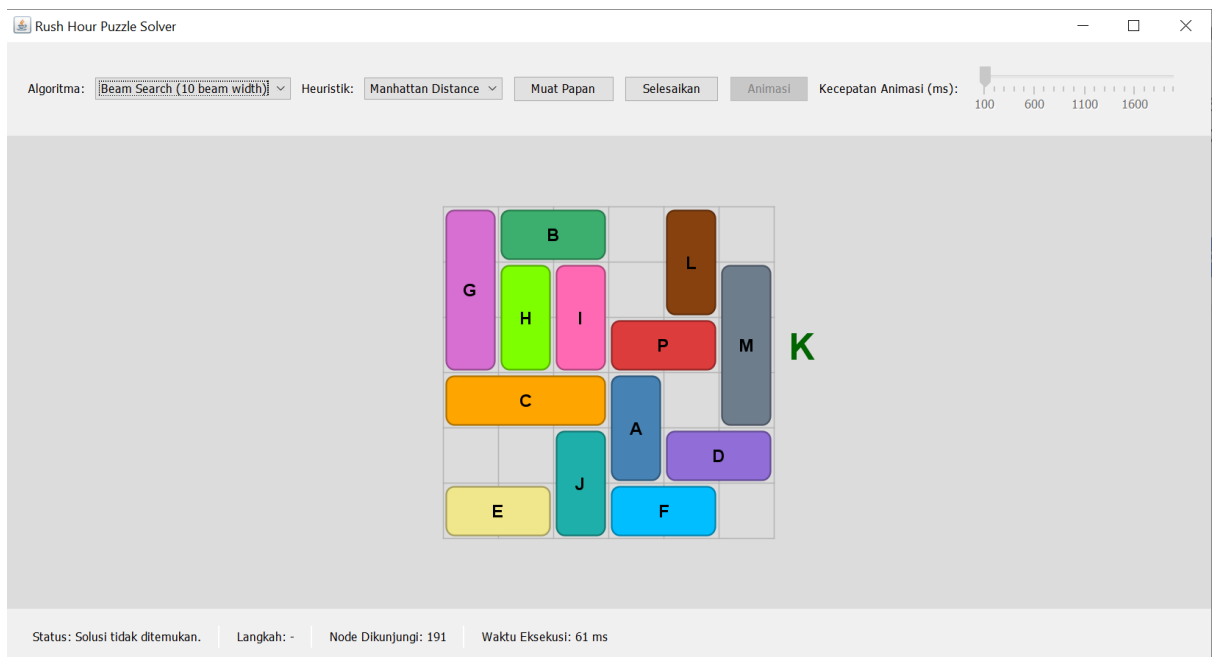
Algoritma: A*

Heuristic: Blocking Pieces



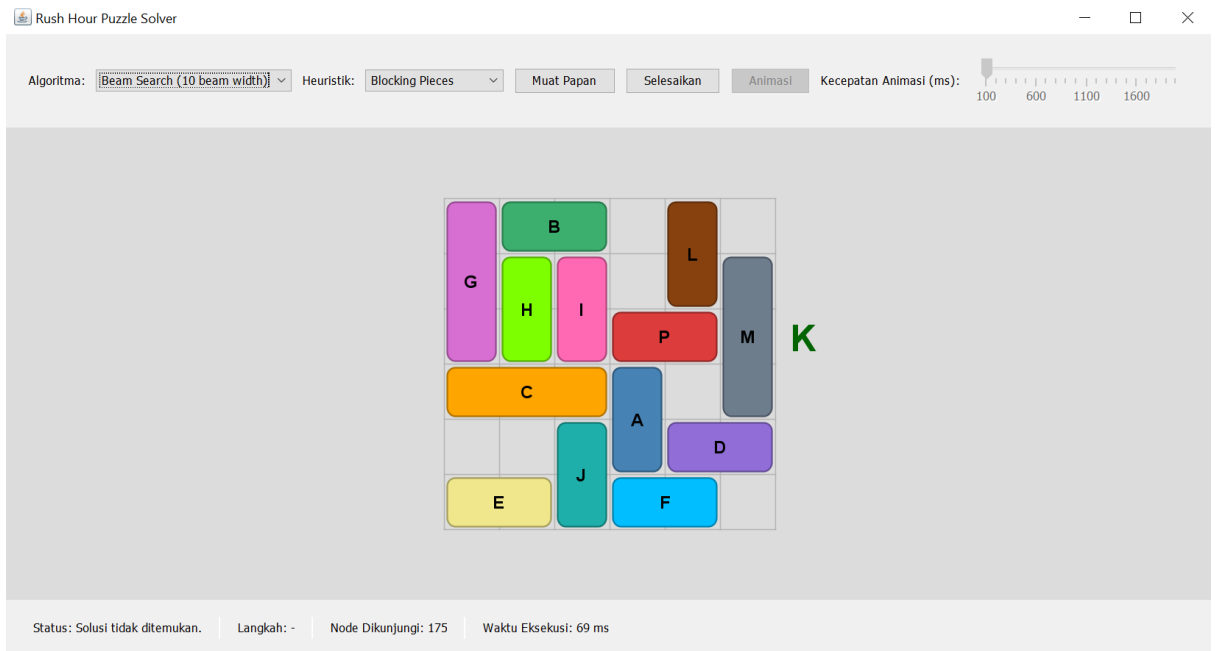
Algoritma: Beam Search Algorithm

Heuristic: Manhattan Distance



Algoritma: Beam Search Algorithm

Heuristic: Blocking Pieces



4. Input

```

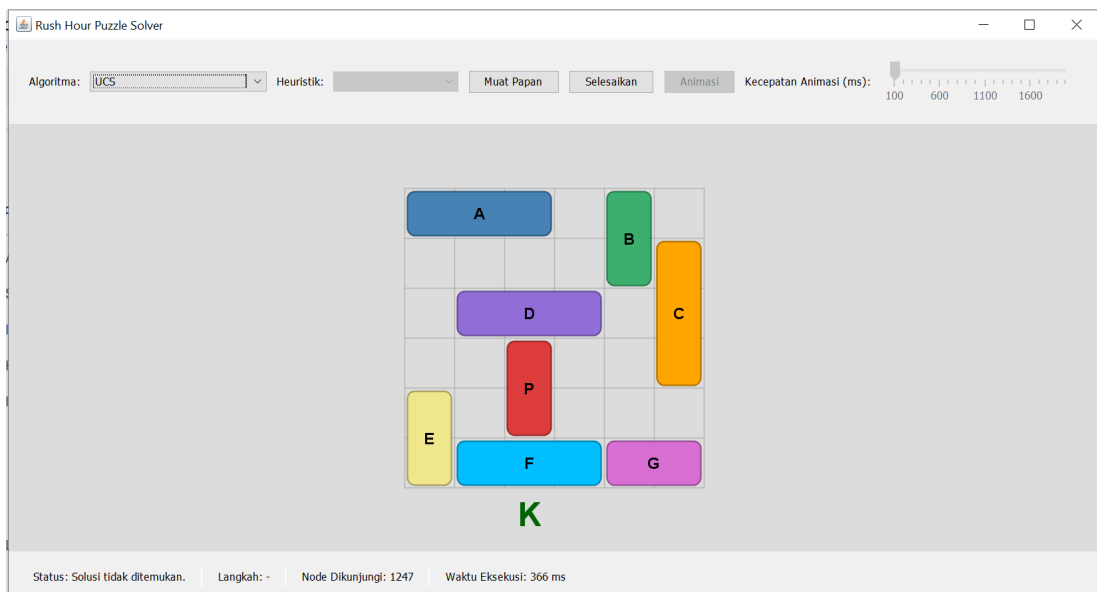
6 6
7
AAA.B.
....BC
.DDD.C
..P..C
E.P...
EFFFGG
  K

```

Output

Algoritma: UCS

Heuristic: -



Hasil Analisis

- UCS mengeksplorasi semua node dengan cost kumulatif dari awal hingga node tersebut. Kompleksitas waktu untuk kasus umum adalah $O(b^{C^*})$ dengan b adalah branching factor dan C^* adalah biaya solusi optimal.
- Greedy Best First Search hanya menggunakan cost heuristic $h(n)$ sebagai pertimbangan untuk mengeksplorasi nodenya sehingga $f(n) = h(n)$. Algoritma ini selalu mencari node yang paling dekat dengan goal. Kompleksitas waktu dari algoritma ini adalah $O(b^m)$ dengan b sebagai branching factor dan m adalah kedalaman maksimum eksplorasi.
- A* menggunakan fungsi $f(n) = g(n) + h(n)$ dengan $g(n)$ adalah biaya dari start ke node n dan $h(n)$ adalah biaya dari n ke goal dalam heuristic. Kompleksitas waktu A* adalah $O(b^d)$ dengan b adalah branching factor dan d adalah kedalaman solusi optimal.
- Beam Search Algorithm adalah pengembangan dari algoritma BFS. Beam search hanya menjaga w node terbaik berdasarkan heuristicnya di setiap level dan tidak menyimpan semua kemungkinan successor. Kompleksitas waktu beam search adalah $O(w * b * d)$ dengan w adalah lebar beam, b adalah branching factor, dan d adalah kedalaman solusi optimal

Implementasi Bonus

Implementasi algoritma pathfinding alternatif

- Mengimplementasikan algoritma Beam Search
- Beam search adalah algoritma pencarian dengan heuristic yang menelusuri graf dengan memperluas node yang terbaik secara sistematis dalam suatu set. Algoritma ini menggabungkan BFS untuk membangun pohon pencariannya dengan menghasilkan successor di setiap level. Algoritma ini hanya mengevaluasi dan memperluas sejumlah set dari node terbaik pada setiap level berdasarkan cost heuristicnya.

Implementasi 2 atau lebih heuristic alternatif

- Menerapkan heuristic Manhattan Distance dan Blocking Pieces untuk pengguna bisa memilih heuristic yang dapat dipilih
 - Manhattan Distance: Menghitung jarak absolut, jumlah sel secara horizontal dan vertikal, dari setiap posisi primary piece ('P') ke posisi keluar ('K') dan mengambil nilai minimumnya.
 - Blocking Pieces Heuristic: Menghitung jumlah piece yang menghalangi jalur primary piece menuju ke posisi keluar.

Program memiliki GUI

- Terletak pada *package* view.gui (src/view/gui)
- RushHourGUI.java: Kelas utama yang menginisialisasi frame aplikasi dan mengatur komponen-komponen GUI lainnya.
- ControlPanel.java: Menyediakan elemen kontrol seperti tombol untuk "Muat Papan", "Selesaikan", "Animasi", serta JComboBox untuk memilih algoritma (UCS, A*, Greedy BFS, Beam Search) dan heuristic (Manhattan Distance, Blocking Pieces). Terdapat slider untuk mengatur kecepatan animasi.
- BoardPanel.java: visualisasikan *board* permainan. Kelas ini menggambar grid, mobil-mobil (dengan warna berbeda, mobil utama 'P' berwarna merah), dan posisi keluar 'K'. BoardPanel juga mampu menyorot mobil yang bergerak selama animasi.
- StatusPanel.java: Menampilkan informasi status seperti status proses (misalnya, "Idle", "Solving..."), waktu eksekusi, jumlah node yang dikunjungi, dan jumlah langkah solusi yang ditemukan.

Lampiran

Pranala Repository Github

https://github.com/drianto/Tucil3_13523106_13523145

Tabel Ketercapaian

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	✓	