LAPORAN TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Oleh :

Muhammad Zakkiy (10122074)

Ghaisan Zaki Pratama (10122078)

Semester II Tahun 2024/2025

INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2025

Daftar Isi

Penjelasan Algoritma *UCS, Greedy Best First Search, dan A**

Uniform Cost Search (UCS) ialah algoritma pencarian rute dengan menelusuri graf dengan biaya sisi yang seragam. Mulai dari simpul awal (*root node*), kami memasukkan semua successor ke *frontier* (antrian prioritas) dengan prioritas $g(n)$ merupakan jumlah langkah yang telah diambil dari simpul awal hingga simul n. Pada setiap iterasi, simpul dengan nilai $g(n)$ terkecil di-pop dan dikembangkan. Semua tetangga yang belum pernah dikunjungi dimasukkan dengan cost yang diperbarui.

Greedy Best First Search (*GBFS*) ialah algoritma pencarian rute dengan menggunakan *heuristic* $h(n)$. *Frontier* disusun menurut simpul yang tampak paling dekat ke tujuan menurut estimasi, tanpa memperhatikan jumlah langkah yang sudah ditempuh. Algoritma ini sering kali memberikan hasil pencarian cepat sampai pada solusi karena bersifat *greedy* mengejar *heuristic,* tetapi tidak menjamin optimalitas.

A* ialah algoritma pencarian rute dengan memilih simpul menurut $f(n) = g(n) + h(n)$ dengan $g(n)$: cost real dari simpul awal ke simpul n dan $h(n)$: *heuristic* yang digunakan untuk mengestimasi cost dari simpul n ke simpul tujuan. *Frontier* selalu mengekspansi simpul dengan $f(n)$ paling kecil. Dengan syarat bahwa $h(n)$ *admissible,* A* menjamin menemukan jalur optimal dan jauh lebih fokus daripada UCS, sehingga mengurangi jumlah simpul yang diperiksa.

Branch and Bound (BnB) merupakan algoritma yang melakukan pruning berdasarkan *Upper Bound* (UB) pada solusi terbaik yang telah ditemukan. Kami tetap menggunakan antrian prioritas seperti UCS (berdasarkan $g(n)$). Saat simpul tujuan pertama kali ditemukan, nilai $g(n)$ kami simpan sebagai UB. Setiap calon successor dengan $g(n) \geq UB$ langsung dibuang, karena tidak mungkin menurunkan cost solusi. Dengan demikian BnB memotong banyak cabang yang jelas memiliki cost yang lebih besar tanpa perlu *heuristic*.

Analisis Algoritma *UCS, Greedy Best First Search, dan A**

Dalam algoritma pencarian rute biasa digunakan tiga fungsi pembanding untuk mencari cost terendah dalam pemilihan rute. Fungsi-fungsi tersebut biasa disimbolkan dengan $f(n)$, $g(n)$, $h(n)$. $g(n)$ adalah cost sesungguhnya (jumlah langkah) dan konfigurasi awal (*root*) hingga simpul n. $h(n)$ adalah estimasi biaya tersisa dari simpul n ke simpul tujuan dengan tiga *heuristic* yang kami gunakan. *Heuristic* yang pertama ialah *Manhattan distance* yaitu jarak *primary piece* ke exit di grid (horizontal + vertikal). *Heuristic* yang kedua ialah *Blocking count* yaitu jumlah *piece* yang berada di antara *primary piece* dan exit pada baris atau kolom yang sama (menghalangi jalan *primary piece* ke exit). *Heuristic* yang ketiga merupakan *Combined* yaitu penjumlahan kedua *heuristics* sebelumnya. $f(n)$ adalah fungsi evaluasi yang digunakan *frontier:*

- UCS: $f(n) = g(n)$
- GBFS: $f(n) = h(n)$
- A*: $f(n) = g(n) + h(n)$
- BnB: $f(n) = g(n)$, tetapi dengan pruning jika $g(n) \geq UB$

Suatu *heuristic* $h(n)$ dikatakan *admissible* jika untuk setiap simpul n, $h(n) \leq h^*(n)$, dengan $h^*(n)$ ialah cost sebenarnya untuk mencapai state tujuan dari state n. *Heuristic* yang *admissible* tidak akan pernah memiliki cost yang lebih kecil dibandingkan banyak langkah sebenarnya untuk mencapai state tujuan. Berikut merupakan penjelasan mengenai *admissible* dari *heuristic* yang kami buat:

- *Manhattan distance* tidak pernah melebihkan jumlah gerakan minimal karena *primary piece* hanya bergerak lurus, maka jelas *admissible*.
- *Blocking count* juga *admissible* karena setiap penghalang memang harus digerakan setidaknya sekali.
- *Combined* tetap *admissible* karena penjumlahan keduanya tidak melebihkan cost sesungguhnya (banyak *piece* yang menghalangi perlu setidaknya digerakkan sekali dan juga jarak dari *primary piece* menggunakan *manhattan* memberikan jumlah yang tidak melebihkan cost sesungguhnya).

Untuk algoritma UCS, ekspansi dilakukan menurut level cost yang sama dengan *Breadth First Search* (BFS). Oleh karena itu, urutan simpul yang diperiksa dan jalur yang dihasilkan oleh UCS dan BFS identik.

Secara teoritis, A* mengeksplorasi hanya simpul yang memiliki $f(n) \leq C^*$ dengan C^* ialah cos solusi optimal, sementara UCS mengeksplorasi semua simpul dengan $g(n) \leq C^*$. Karena $f(n) = g(n) + h(n) \geq g(n)$, A* umumnya memeriksa lebih sedikit simpul. Oleh karena itu A* lebih efisien daripada UCS.

GBFS merupakan algoritma berbasis *greedy*. GBFS dapat memilih jalur yang secara *heuristic* "tampak" cepat ke exit, tetapi mengorbankan panjang jalur yang sudah ditempuh. GBFS tidak menjamin menemukan jalur paling pendek (optimal).

Pruning berbasis UB pada BnB memungkinkan pengurangan eksplorasi cabang yang sudah pasti memiliki cost yang tinggi. Karena tidak bergantung *heuristic,* efektivitas pruning tergantung pada seberapa cepat UB (cost solusi pertama) ditemukan. Secara teori BnB menjamin optimalitas sama seperti UCS/A*, tetapi efisiensinya dapat bervariasi. Pada kasus solusi dengan cost rendah ditemukan awal, BnB jauh lebih hemat dibandingkan UCS, jika tidak bisa mendekati performa UCS.

Source Program

RushHour.java

```
import java.io.*;
import java.nio.file.*;
import solver.Solver;
import util.Parser;

public class RushHour {
    public static void main(String[] args) throws Exception {
        // Validasi jumlah argumen
        if (args.length < 3) {
            // Jika argumen kurang dari 2, tampilkan petunjuk penggunaan
            if (args.length < 2) {
                System.out.println("Usage: java RushHour <filename> <ucs|gbfs|astar|bnb>
<manhattan|blocking|combined>");
                return;
            }
            // Jika algoritma bukan ucs/bnb, tapi heuristik belum diberikan
            if (!args[1].equals("ucs") && !args[1].equals("bnb")) {
                System.out.println("Usage: java RushHour <filename> <ucs|gbfs|astar|bnb>
<manhattan|blocking|combined>");
                return;
            }
        }

        // Siapkan nama output berdasarkan input dan parameter
        String inputName = Paths.get(args[0]).getFileName().toString();
        Path outputPath;

        if (args.length < 3) {
            // Untuk algoritma yang tidak butuh heuristik (UCS, BnB)
            outputPath = Paths.get("test", "solusi_" + args[1] + "_" + inputName);
        } else {
            // Untuk algoritma yang menggunakan heuristik
            outputPath = Paths.get("test", "solusi_" + args[1] + "_" + args[2] + "_" +
inputName);
        }

        // Pastikan direktori output ada
        Files.createDirectories(outputPath.getParent());

        // Siapkan file output
        PrintStream fileOut = new PrintStream(new FileOutputStream(outputPath.toFile()), true);
        PrintStream console = System.out;

        // Ganti System.out agar output dikirim ke file dan terminal
        System.setOut(new PrintStream(new OutputStream() {
            private final byte ESC = 0x1B;
            private boolean inAnsi = false;

            @Override
            public void write(int b) throws IOException {
                console.write(b); // tetap tampilkan di console
                if (b == ESC) {
                    inAnsi = true;
                    return;
                }
                if (inAnsi) {
                    if ((char) b == 'm') {
                        inAnsi = false;
                    }
                    return;
                }
                fileOut.write(b); // juga tulis ke file
            }
```

```
            @Override public void flush() throws IOException {
                console.flush();
                fileOut.flush();
            }

            @Override public void close() throws IOException {
                console.close();
                fileOut.close();
            }
        }, true));

        // Atur heuristik jika diperlukan
        if (!args[1].equals("ucs") && !args[1].equals("bnb")) {
            String heu = args[2].toLowerCase();
            switch (heu) {
                case "manhattan":
                    Solver.heuristicMode = Solver.HeuristicMode.MANHATTAN;
                    break;
                case "blocking":
                    Solver.heuristicMode = Solver.HeuristicMode.BLOCKING;
                    break;
                case "combined":
                    Solver.heuristicMode = Solver.HeuristicMode.COMBINED;
                    break;
                default:
                    throw new IllegalArgumentException("Unknown heuristic: " + heu);
            }
        }

        // Atur algoritma pencarian
        String alg = args[1].toLowerCase();
        switch (alg) {
            case "ucs":
                Solver.useAStar = false;
                Solver.useGreedy = false;
                Solver.useBnB = false;
                System.out.println("Running Uniform Cost Search...");
                break;
            case "astar":
                Solver.useAStar = true;
                Solver.useGreedy = false;
                Solver.useBnB = false;
                System.out.println("Running A* Search...");
                break;
            case "gbfs":
                Solver.useAStar = false;
                Solver.useGreedy = true;
                Solver.useBnB = false;
                System.out.println("Running Greedy Best First Search...");
                break;
            case "bnb":
                Solver.useAStar = false;
                Solver.useGreedy = false;
                Solver.useBnB = true;
                System.out.println("Running Branch and Bound Search...");
                break;
            default:
                Solver.useAStar = true;
                Solver.useGreedy = false;
                Solver.useBnB = false;
                System.out.println("Running A* Search...");
                break;
        }

        // Parse input board
        int[] dimensions = new int[2]; // [rows, cols]
        char[][] board = Parser.parseInput(args[0], dimensions);

        // Jalankan solver
```

```
        Solver.solve(board, Parser.rows, Parser.cols, Parser.exit);

        // Tutup file output
        fileOut.close();
    }
}
```

RusHourGUI.java

```
import java.awt.*;
import java.io.File;
import java.io.IOException;
import java.util.List;
import javax.swing.*;
import javax.swing.border.LineBorder;
import solver.Solver;
import util.Parser;

public class RushHourGUI extends JFrame {
    private JLabel statsLabel;
    private JComboBox<String> algoBox, heuBox;
    private JButton chooseFileBtn, runBtn;
    private JPanel boardPanel;
    private File inputFile;
    private List<char[][]> solutionPath; // hasil solve: list konfigurasi board

    public RushHourGUI() {
        super("Rush Hour Solver");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout(8,8));

        // --- Top controls ---
        JPanel control = new JPanel(new FlowLayout(FlowLayout.LEFT, 6,6));
        chooseFileBtn = new JButton("Pilih File");
        algoBox = new JComboBox<>(new String[]{"UCS","GBFS","A*","BnB"});
        heuBox  = new JComboBox<>(new String[]{"Manhattan","Blocking","Combined"});
        runBtn   = new JButton("Jalankan");
        control.add(chooseFileBtn);
        control.add(new JLabel("Algoritma:")); control.add(algoBox);
        control.add(new JLabel("Heuristic:"));  control.add(heuBox);
        control.add(runBtn);
        add(control, BorderLayout.NORTH);

        // --- Center: tempat papan ---
        boardPanel = new JPanel();
        add(boardPanel, BorderLayout.CENTER);
        statsLabel = new JLabel(" ");
        statsLabel.setBorder(BorderFactory.createEmptyBorder(4, 8, 4, 8));
        add(statsLabel, BorderLayout.SOUTH);

        // --- Listeners ---
        chooseFileBtn.addActionListener(e -> {
            JFileChooser fc = new JFileChooser(".");
            if (fc.showOpenDialog(this)==JFileChooser.APPROVE_OPTION) {
                inputFile = fc.getSelectedFile();
                chooseFileBtn.setText(inputFile.getName());
            }
        });

        runBtn.addActionListener(e -> {
            if (inputFile == null) {
                JOptionPane.showMessageDialog(this, "Silakan pilih file puzzle terlebih
dahulu.");
                return;
            }
            runSolverAndAnimate();
        });
```

```
        setSize(600, 650);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    private void runSolverAndAnimate() {
        try {
            // Atur pilihan algoritma & heuristic ke Solver
            String alg = (String)algoBox.getSelectedItem();
            Solver.useAStar  = "A*".equals(alg);
            Solver.useGreedy = "GBFS".equals(alg);
            Solver.useBnB    = "BnB".equals(alg);
            // (UCS: kedua flag false)

            String heu = ((String)heuBox.getSelectedItem()).toLowerCase();
            switch (heu) {
                case "manhattan": Solver.heuristicMode = Solver.HeuristicMode.MANHATTAN;
break;
                case "blocking":  Solver.heuristicMode = Solver.HeuristicMode.BLOCKING;
break;
                default:          Solver.heuristicMode = Solver.HeuristicMode.COMBINED;
break;
            }

            // Parse dan solve
            int[] dim = new int[2];
            char[][] startBoard = Parser.parseInput(inputFile.getAbsolutePath(), dim);

            // 1) Tampilkan board awal apa adanya
            boardPanel.removeAll();
            displayBoard(startBoard, '\0');

            // 2) Jalankan solver & dapatkan path
            solutionPath = Solver.solveAndReturnPath(
                startBoard, Parser.rows, Parser.cols, Parser.exit);
            long nodes = Solver.getNodesExpanded();
            double secs = Solver.getElapsedSeconds();

            // update statsLabel
            statsLabel.setText(
                String.format("Nodes expanded: %,d    Waktu eksekusi: %.3f s", nodes, secs)
            );
            if (solutionPath != null && !solutionPath.isEmpty()) {
               char[][] last = solutionPath.get(solutionPath.size() - 1);
               for (int r = 0; r < last.length; r++) {
                   for (int c = 0; c < last[0].length; c++) {
                       if (last[r][c] == 'P') {
                           last[r][c] = '.';
                       }
                   }
               }
           }
            // 3) Tangani no-solution
            if (solutionPath == null || solutionPath.isEmpty()) {
                JOptionPane.showMessageDialog(this,
                    "No solution found", "Hasil", JOptionPane.INFORMATION_MESSAGE);
                return;
            }

            // 4) Animasi pergerakan
            Timer timer = new Timer(700, null);
            final int[] idx = {1};
            timer.addActionListener(evt -> {
                if (idx[0] < solutionPath.size()) {
                    char[][] prev = solutionPath.get(idx[0]-1);
                    char[][] curr = solutionPath.get(idx[0]);
                    char moved = detectMovedPiece(prev, curr);
                    displayBoard(curr, moved);
                    idx[0]++;
```

```
                } else {
                    ((Timer)evt.getSource()).stop();
                }
            });
            timer.start();

        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this,
                "Gagal membaca file: " + ex.getMessage(),
                "Error", JOptionPane.ERROR_MESSAGE);
        } catch (Exception ex) {
            ex.printStackTrace();
            JOptionPane.showMessageDialog(this,
                "Error: " + ex.getMessage(),
                "Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(RushHourGUI::new);
    }
    private void displayBoard(char[][] board, char activePiece) {
        boardPanel.removeAll();
        int R = board.length, C = board[0].length;
        boardPanel.setLayout(new GridLayout(R, C, 1,1));
        for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++) {
                char c = board[i][j];
                JLabel cell = new JLabel(String.valueOf(c), SwingConstants.CENTER);
                cell.setOpaque(true);
                cell.setBorder(new LineBorder(Color.DARK_GRAY));
                if (c == 'P') {
                    cell.setBackground(Color.RED);
                } else if (c == 'K') {
                    cell.setBackground(Color.GREEN);
                } else if (c == activePiece) {
                    cell.setBackground(Color.YELLOW);
                } else if (c == '.') {
                    cell.setBackground(Color.WHITE); // jalan kosong jadi putih
                } else {
                    cell.setBackground(Color.LIGHT_GRAY); // mobil jadi abu-abu
                }
                boardPanel.add(cell);
            }
        }
        boardPanel.revalidate();
        boardPanel.repaint();
    }

        /**
     * Cari karakter piece yang pindah antara board a dan b.
     */
    private char detectMovedPiece(char[][] a, char[][] b) {
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a[0].length; j++) {
                if (a[i][j] != b[i][j] && b[i][j] != '.' && b[i][j] != 'K') {
                    return b[i][j];
                }
            }
        }
        return '\0';
    }


}
```

Solver.java

```java
package solver;

import java.util.*;
import model.*;

/**
 * Solver class untuk menyelesaikan puzzle Rush Hour menggunakan algoritma pencarian:
 * A*, Greedy, dan Branch and Bound (BnB).
 */
public class Solver {
    // Pengaturan algoritma yang digunakan
    public static boolean useAStar = false;
    public static boolean useGreedy = false;
    public static boolean useBnB    = false;

    // Ukuran papan
    public static int rows, cols;

    // Pengaturan warna terminal
    public static boolean enableColor = true;

    // Mode heuristik
    public enum HeuristicMode { MANHATTAN, BLOCKING, COMBINED }
    public static HeuristicMode heuristicMode = HeuristicMode.COMBINED;

    // Kode warna ANSI
    private static final String RESET  = "\u001B[0m";
    private static final String RED    = "\u001B[31m";  // P (primary piece)
    private static final String GREEN  = "\u001B[32m";  // K (exit)
    private static final String YELLOW = "\u001B[33m";  // Piece yang aktif bergerak

    // Variabel internal
    private static Position exitPosition;
    private static long nodesExpanded;
    private static long startTime, endTime;
    private static long elapsedNanos;

    /**
     * Fungsi utama untuk menjalankan algoritma pencarian dari kondisi awal.
     */
    public static void solve(char[][] initialBoard, int n_rows, int n_cols, Position exit) {
        rows = n_rows;
        cols = n_cols;
        exitPosition = exit;
        nodesExpanded = 0;
        startTime = System.nanoTime();

        PriorityQueue<State> pq = new PriorityQueue<>();
        Set<String> visited = new HashSet<>();
        int bestCost = Integer.MAX_VALUE;
        State bestGoal = null;

        State start = new State(initialBoard, rows, cols, 0, null, exitPosition);
        pq.add(start);

        while (!pq.isEmpty()) {
            State current = pq.poll();
            String boardStr = boardToString(current.board);

            if (visited.contains(boardStr)) continue;
            visited.add(boardStr);
            nodesExpanded++;

            if (isGoal(current.board, exitPosition)) {
                // Modifikasi papan jika goal tercapai (untuk mencetak solusi)
                modifyBoardForGoal(current);

                if (useBnB) {
                    bestCost = current.cost;
                    bestGoal = current;
```

```
                    continue;
                } else {
                    endTime = System.nanoTime();
                    printSolution(current);
                    printStats(nodesExpanded, startTime, endTime);
                    return;
                }
            }

            // Tambahkan semua successor ke antrian
            for (State next : generateSuccessors(current, exitPosition)) {
                if (useBnB && next.cost >= bestCost) continue;
                pq.add(next);
            }
        }

        // Jika BnB dan goal terbaik ditemukan
        if (useBnB && bestGoal != null) {
            endTime = System.nanoTime();
            printSolution(bestGoal);
            printStats(nodesExpanded, startTime, endTime);
            return;
        }

        System.out.println("No solution found.");
    }

    /** Mengecek apakah papan saat ini adalah goal state */
    static boolean isGoal(char[][] board, Position exit) {
        return board[exit.row][exit.col] == 'P';
    }

    /** Statistik proses pencarian */
    private static void printStats(long nodes, long t0, long t1) {
        double secs = (t1 - t0) / 1_000_000_000.0;
        System.out.println("\n--- Search stats ---");
        System.out.printf("Nodes expanded : %d\n", nodes);
        System.out.printf("Elapsed time   : %.3f s\n", secs);
        System.out.println("--------------------\n");
    }

    /** Generate semua successor yang mungkin dari sebuah state */
    static List<State> generateSuccessors(State current, Position exit) {
        char[][] board = current.board;
        Set<Character> pieces = new HashSet<>();

        // Identifikasi semua pieces di papan
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++) {
                char c = board[i][j];
                if (c != '.' && c != 'K' && c != ' ') pieces.add(c);
            }

        List<State> successors = new ArrayList<>();
        for (char piece : pieces) {
            List<Position> positions = new ArrayList<>();

            // Posisi semua bagian dari piece
            for (int i = 0; i < rows; i++)
                for (int j = 0; j < cols; j++)
                    if (board[i][j] == piece) positions.add(new Position(i, j));

            boolean horizontal = positions.stream().allMatch(p -> p.row ==
positions.get(0).row);

            // Gerakan horizontal
            if (horizontal) {
                int r = positions.get(0).row;
                int minCol = positions.stream().mapToInt(p -> p.col).min().getAsInt();
                int maxCol = positions.stream().mapToInt(p -> p.col).max().getAsInt();
```

```
                int i = 0;

                // Geser ke kiri
                while (minCol - i > 0 && (board[r][minCol - 1 - i] == '.' || board[r][minCol
- 1 - i] == 'K')) {
                    char[][] newBoard = copyBoard(board);
                    for (int j = 0; j <= i; j++) {
                        newBoard[r][maxCol - j] = '.';
                        newBoard[r][minCol - 1 - j] = piece;
                    }
                    successors.add(new State(newBoard, rows, cols, current.cost + 1,
current, exit));
                    i++;
                }

                // Geser ke kanan
                i = 0;
                while (maxCol + i < board[r].length - 1 && (board[r][maxCol + 1 + i] == '.'
|| board[r][maxCol + 1 + i] == 'K')) {
                    char[][] newBoard = copyBoard(board);
                    for (int j = 0; j <= i; j++) {
                        newBoard[r][minCol + j] = '.';
                        newBoard[r][maxCol + 1 + j] = piece;
                    }
                    successors.add(new State(newBoard, rows, cols, current.cost + 1,
current, exit));
                    i++;
                }

            } else {
                // Gerakan vertikal
                int c = positions.get(0).col;
                int minRow = positions.stream().mapToInt(p -> p.row).min().getAsInt();
                int maxRow = positions.stream().mapToInt(p -> p.row).max().getAsInt();
                int length = rows;

                // Geser ke atas
                int i = 0;
                while (minRow - i > 0 && (board[minRow - 1 - i][c] == '.' || board[minRow - 1
- i][c] == 'K')) {
                    char[][] newBoard = copyBoard(board);
                    for (int j = 0; j <= i; j++) {
                        newBoard[maxRow - j][c] = '.';
                        newBoard[minRow - 1 - j][c] = piece;
                    }
                    successors.add(new State(newBoard, rows, cols, current.cost + 1,
current, exit));
                    i++;
                }

                // Geser ke bawah
                i = 0;
                while (maxRow + i < length - 1 && (board[maxRow + 1 + i][c] == '.' ||
board[maxRow + 1 + i][c] == 'K')) {
                    char[][] newBoard = copyBoard(board);
                    for (int j = 0; j <= i; j++) {
                        newBoard[minRow + j][c] = '.';
                        newBoard[maxRow + 1 + j][c] = piece;
                    }
                    successors.add(new State(newBoard, rows, cols, current.cost + 1,
current, exit));
                    i++;
                }
            }
        }

        return successors;
    }

    /** Menyalin papan */
```

```
    static char[][] copyBoard(char[][] board) {
        char[][] newBoard = new char[rows][cols];
        for (int i = 0; i < rows; i++) newBoard[i] = board[i].clone();
        return newBoard;
    }

    /** Konversi papan ke string untuk keperluan hash */
    private static String boardToString(char[][] board) {
        StringBuilder sb = new StringBuilder();
        for (char[] row : board) sb.append(row).append("\n");
        return sb.toString();
    }

    /** Modifikasi papan untuk menampilkan tujuan */
    private static void modifyBoardForGoal(State current) {
        List<Position> positions = new ArrayList<>();
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                if (current.board[i][j] == 'P') positions.add(new Position(i, j));

        boolean horizontal = positions.stream().allMatch(p -> p.row == positions.get(0).row);
        if (horizontal) {
            if (exitPosition.col == 0) current.board[exitPosition.row][exitPosition.col + 1]
= '.';
            else current.board[exitPosition.row][exitPosition.col - 1] = '.';
        } else {
            if (exitPosition.row == 0) current.board[exitPosition.row + 1][exitPosition.col]
= '.';
            else current.board[exitPosition.row - 1][exitPosition.col] = '.';
        }
        current.board[exitPosition.row][exitPosition.col] = 'K';
    }

    /** Cetak solusi langkah per langkah */
    private static void printSolution(State goal) {
        List<State> path = new ArrayList<>();
        for (State s = goal; s != null; s = s.parent) path.add(s);
        Collections.reverse(path);

        System.out.println("Papan Awal");
        printBoard(path.get(0).board, '\0');

        for (int i = 1; i < path.size(); i++) {
            char[][] prev = path.get(i - 1).board;
            char[][] curr = path.get(i).board;
            boolean lastMove = isGoal(curr, exitPosition);
            char moved = detectMovedPiece(prev, curr);
            if (moved == '?' && lastMove) moved = 'P';

            String dir = (moved == 'P' && lastMove)
                       ? lastMoveDirection(prev, moved)
                       : detectDirection(prev, curr, moved);

            System.out.printf("\nGerakan %d: %c-%s%n", i, moved, dir);
            printBoard(curr, moved);
        }
    }

    /** Deteksi arah gerakan terakhir menuju goal */
    private static String lastMoveDirection(char[][] prev, char piece) {
        int minR = rows, maxR = -1, minC = cols, maxC = -1;
        for (int r = 0; r < rows; r++)
            for (int c = 0; c < cols; c++)
                if (prev[r][c] == piece) {
                    minR = Math.min(minR, r);
                    maxR = Math.max(maxR, r);
                    minC = Math.min(minC, c);
                    maxC = Math.max(maxC, c);
                }
        if (minR == maxR) return exitPosition.col > maxC ? "kanan" : "kiri";
```

```
        else return exitPosition.row > maxR ? "bawah" : "atas";
    }

    /** Deteksi piece yang bergerak antara dua papan */
    private static char detectMovedPiece(char[][] a, char[][] b) {
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                if (a[i][j] != b[i][j] && b[i][j] != '.' && b[i][j] != 'K')
                    return b[i][j];
        return 'P';
    }

    /** Deteksi arah gerakan sebuah piece */
    private static String detectDirection(char[][] a, char[][] b, char piece) {
        Position before = findPosition(a, piece);
        Position after  = findPosition(b, piece);
        if (before == null || after == null) return piece == 'P' ? lastMoveDirection(a,
piece) : "?";
        return before.row == after.row
               ? (after.col > before.col ? "kanan" : "kiri")
               : (after.row > before.row ? "bawah" : "atas");
    }

    /** Temukan posisi awal dari piece */
    private static Position findPosition(char[][] board, char piece) {
        for (int i = 0; i < board.length; i++)
            for (int j = 0; j < board[i].length; j++)
                if (board[i][j] == piece) return new Position(i, j);
        return null;
    }

    /** Cetak papan ke layar */
    public static void printBoard(char[][] board, char activePiece) {
        for (char[] row : board) {
            for (char c : row) {
                if      (c == 'P' && enableColor) System.out.print(RED + c + RESET);
                else if (c == 'K' && enableColor) System.out.print(GREEN + c + RESET);
                else if (c == activePiece && enableColor) System.out.print(YELLOW + c +
RESET);
                else System.out.print(c);
            }
            System.out.println();
        }
    }

    public static List<char[][]> solveAndReturnPath(
            char[][] initialBoard, int n_rows, int n_cols, Position exit) {
        rows = n_rows;
        cols = n_cols;
        exitPosition = exit;
        nodesExpanded = 0;
        long start = System.nanoTime();

        PriorityQueue<State> pq = new PriorityQueue<>();
        Set<String> visited = new HashSet<>();
        State goalState = null;

        pq.add(new State(initialBoard, rows, cols, 0, null, exit));

        while (!pq.isEmpty()) {
            State current = pq.poll();
            String key = boardToString(current.board);
            if (visited.contains(key)) continue;
            visited.add(key);
            nodesExpanded++;

            if (isGoal(current.board, exitPosition)) {
                goalState = current;
                break;
            }
```

```
            for (State next : generateSuccessors(current, exitPosition)) {
                pq.add(next);
            }
        }

        elapsedNanos = System.nanoTime() - start;

        if (goalState == null) return Collections.emptyList();

        LinkedList<char[][]> path = new LinkedList<>();
        for (State s = goalState; s != null; s = s.parent) {
            char[][] copy = new char[rows][cols];
            for (int r = 0; r < rows; r++) copy[r] = s.board[r].clone();
            path.addFirst(copy);
        }
        return path;
    }

    /** GUI akan memanggil ini untuk menampilkan "banyak node" */
    public static long getNodesExpanded() {
        return nodesExpanded;
    }
    /** GUI akan memanggil ini untuk menampilkan "waktu" dalam detik */
    public static double getElapsedSeconds() {
        return elapsedNanos / 1_000_000_000.0;
    }
}
```

Heuristics.java

```
package util;

import model.Position;

/**
 * Kelas Heuristics menyediakan fungsi-fungsi heuristik yang digunakan untuk
 * algoritma pencarian seperti A*, Greedy Best First Search, dan Combined heuristic.
 *
 * Terdapat tiga jenis heuristik:
 * 1. Manhattan Distance
 * 2. Jumlah kendaraan yang menghalangi
 * 3. Kombinasi dari keduanya
 */
public class Heuristics {

    /**
     * Menghitung jarak Manhattan antara kendaraan utama ('P') dan posisi keluar ('K').
     * Jarak ini merupakan jumlah langkah horizontal dan vertikal minimal untuk mencapai pintu
keluar.
     *
     * @param board papan permainan
     * @param exit  posisi keluar
     * @param rows  jumlah baris papan
     * @param cols  jumlah kolom papan
     * @return jarak Manhattan dari 'P' ke 'K'
     */
    public static int manhattan(char[][] board, Position exit, int rows, int cols) {
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                if (board[i][j] == 'P')
                    return Math.abs(i - exit.row) + Math.abs(j - exit.col);
        return 0; // fallback jika 'P' tidak ditemukan (seharusnya tidak terjadi)
    }

    /**
     * Menghitung jumlah kendaraan yang menghalangi jalur kendaraan utama ('P') menuju keluar
```

```
('K').
     * Heuristik ini memperkirakan hambatan aktual tanpa memperhitungkan jarak.
     *
     * @param board papan permainan
     * @param exit  posisi keluar
     * @param rows  jumlah baris papan
     * @param cols  jumlah kolom papan
     * @return jumlah kendaraan yang menghalangi jalur
     */
    public static int blockingCount(char[][] board, Position exit, int rows, int cols) {
        int blockers = 0;
        int minR = rows, maxR = -1, minC = cols, maxC = -1;

        // Cari rentang posisi kendaraan utama ('P') di papan
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                if (board[r][c] == 'P') {
                    minR = Math.min(minR, r);
                    maxR = Math.max(maxR, r);
                    minC = Math.min(minC, c);
                    maxC = Math.max(maxC, c);
                }
            }
        }

        if (minR == maxR) {
            // Kendaraan utama bergerak horizontal
            for (int c = Math.min(exit.col, minC); c <= Math.max(exit.col, maxC); c++) {
                char cell = board[minR][c];
                if (cell != 'P' && cell != '.' && cell != 'K') {
                    blockers++;
                }
            }
        } else {
            // Kendaraan utama bergerak vertikal
            for (int r = Math.min(exit.row, minR); r <= Math.max(exit.row, maxR); r++) {
                char cell = board[r][minC];
                if (cell != 'P' && cell != '.' && cell != 'K') {
                    blockers++;
                }
            }
        }

        return blockers;
    }

    /**
     * Kombinasi dari heuristik Manhattan dan blocking count.
     * Biasanya digunakan untuk mendapatkan estimasi yang lebih informatif.
     *
     * @param board papan permainan
     * @param exit  posisi keluar
     * @param rows  jumlah baris papan
     * @param cols  jumlah kolom papan
     * @return kombinasi heuristik: jarak + jumlah blocker
     */
    public static int combined(char[][] board, Position exit, int rows, int cols) {
        return manhattan(board, exit, rows, cols)
             + blockingCount(board, exit, rows, cols);
    }
}
```

Parser.java

```
package util;

import java.io.*;
```

```java
import java.nio.file.*;
import java.util.*;
import model.Position;

public class Parser {
    public static Position exit;
    public static int rows, cols;

    /**
     * Membaca file, menambah baris/kolom jika K berada di luar grid,
     * dan mengembalikan board final beserta exit position.
     */
    public static char[][] parseInput(String filename, int[] dimensions) throws IOException {
        List<String> lines = Files.readAllLines(Paths.get(filename));

        // baris 0: "R C"
        String[] size = lines.get(0).trim().split("\\s+");
        int origR = Integer.parseInt(size[0]);
        int origC = Integer.parseInt(size[1]);

        // baris 1: jumlah pieces (abaikan)
        List<String> gridLines = lines.subList(2, lines.size());

        // cari posisi K dalam gridLines (raw index)
        int rawR = -1, rawC = -1;
        for (int i = 0; i < gridLines.size(); i++) {
            int idx = gridLines.get(i).indexOf('K');
            if (idx != -1) {
                rawR = i;
                rawC = idx;
                break;
            }
        }
        if (rawR == -1) {
            throw new IllegalArgumentException("Tidak ditemukan huruf K di input.");
        }

        // tentukan extra top/bottom/left/right
        boolean extraTop    = rawR < 0;
        boolean extraBottom = rawR >= origR;
        boolean extraLeft   = rawC < 0;
        boolean extraRight  = rawC >= origC;

        // ukuran akhir
        rows = origR + (extraTop ? 1 : 0) + (extraBottom ? 1 : 0);
        cols = origC + (extraLeft ? 1 : 0) + (extraRight ? 1 : 0);

        // simpan ke dimensions
        dimensions[0] = rows;
        dimensions[1] = cols;

        // buat board dengan spasi
        char[][] board = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            Arrays.fill(board[i], ' ');
        }

        // hitung offset kalau ada extra top/left
        int rowOff = extraTop ? 1 : 0;
        int colOff = extraLeft ? 1 : 0;

        // copy grid asli (tanpa K) ke board
        for (int i = 0; i < origR; i++) {
            String line = gridLines.get(i);
            for (int j = 0; j < origC; j++) {
                char c = (j < line.length() ? line.charAt(j) : ' ');
                if (c != 'K') {
                    board[i + rowOff][j + colOff] = c;
                }
            }
```

```
        }

        // hitung posisi exit di board akhir
        int exitR = rawR + rowOff;
        int exitC = rawC + colOff;
        board[exitR][exitC] = 'K';
        exit = new Position(exitR, exitC);

        return board;
    }


    /**
     * Menghitung jumlah baris dalam file input.
     *
     * @param filename Nama file input.
     * @return Jumlah baris dalam file.
     */
    public static int countLines(String filename) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        int lines = 0;
        while (reader.readLine() != null) {
            lines++;
        }
        reader.close();
        return lines;
    }
}
```

Position.java

```
package model;

/**
 * Kelas Position merepresentasikan posisi suatu objek
 * di dalam papan permainan sebagai koordinat (baris, kolom).
 */
public class Position {
    public int row; // Baris dari posisi
    public int col; // Kolom dari posisi

    /**
     * Konstruktor untuk membuat posisi baru dengan koordinat tertentu.
     *
     * @param row Baris (y) dari posisi.
     * @param col Kolom (x) dari posisi.
     */
    public Position(int row, int col) {
        this.row = row;
        this.col = col;
    }
}
```

State.java

```
package model;

import solver.Solver;
import util.Heuristics;

/**
 * Kelas State merepresentasikan sebuah kondisi/papan dalam pencarian solusi puzzle Rush Hour.
 * Setiap state menyimpan papan, cost, parent (untuk rekonstruksi solusi), dan estimasi biaya
(heuristik).
 */
```

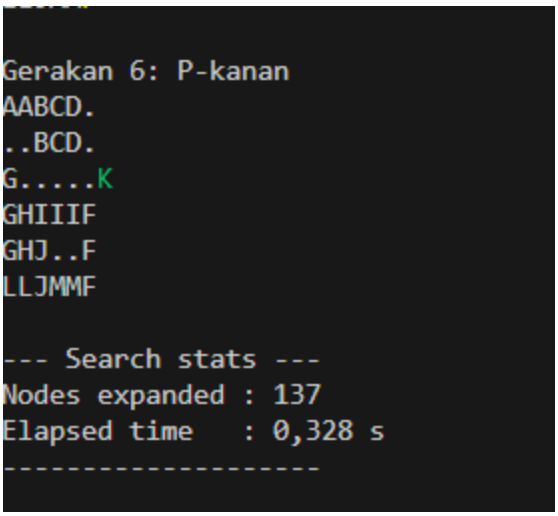
```java
public class State implements Comparable<State> {
    public char[][] board;        // Representasi papan permainan
    public int cost;              // Biaya aktual (g(n)) untuk mencapai state ini dari awal
    public int estimatedCost;     // Estimasi biaya ke goal (h(n)), tergantung algoritma
    public State parent;          // Referensi ke state sebelumnya (untuk pelacakan jalur)
    public int rows, cols;        // Ukuran papan

    /**
     * Konstruktor untuk membuat state baru.
     *
     * @param board  Papan permainan saat ini
     * @param n_rows Jumlah baris papan
     * @param n_cols Jumlah kolom papan
     * @param cost   Biaya aktual g(n)
     * @param parent State sebelumnya dalam jalur pencarian
     * @param exit   Posisi pintu keluar
     */
    public State(char[][] board, int n_rows, int n_cols,
                 int cost, State parent, Position exit) {
        this.board  = board;
        this.cost   = cost;
        this.parent = parent;
        this.rows   = n_rows;
        this.cols   = n_cols;

        // Hitung estimasi biaya h(n) hanya jika menggunakan A* atau Greedy
        if (Solver.useAStar || Solver.useGreedy) {
            switch (Solver.heuristicMode) {
                case MANHATTAN:
                    this.estimatedCost = Heuristics.manhattan(board, exit, rows, cols);
                    break;
                case BLOCKING:
                    this.estimatedCost = Heuristics.blockingCount(board, exit, rows, cols);
                    break;
                default: // COMBINED
                    this.estimatedCost = Heuristics.combined(board, exit, rows, cols);
            }
        } else {
            this.estimatedCost = 0;
        }
    }

    /**
     * Perbandingan antar state untuk keperluan antrian prioritas.
     * Bergantung pada algoritma yang digunakan (UCS, GBFS, A*).
     */
    @Override
    public int compareTo(State other) {
        if (Solver.useAStar) {
            // f(n) = g(n) + h(n)
            return Integer.compare(this.cost + this.estimatedCost,
                                   other.cost + other.estimatedCost);
        }
        if (Solver.useGreedy) {
            // h(n) saja
            return Integer.compare(this.estimatedCost, other.estimatedCost);
        }
        // UCS dan BnB: g(n) saja
        return Integer.compare(this.cost, other.cost);
    }
}
```

Test Case

No	*Input*	*Output*

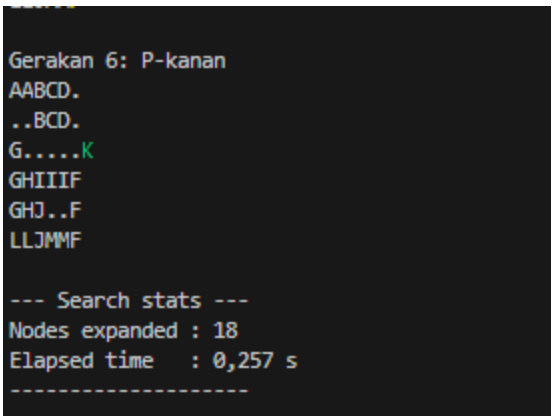
1	Algoritma : UCS Test case : 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Running Uniform Cost Search... Papan Awal AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. Gerakan 1: C-atas AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. Gerakan 2: D-atas AABCDF ..BCDF GPP..FK GH.III GHJ... LLJMM. Gerakan 3: I-kiri AABCDF ..BCDF GPP..FK GHIII. GHJ... LLJMM. Gerakan 4: F-bawah AABCD. ..BCD. GPP...K GHIIIF GHJ..F LLJMMF Gerakan 5: P-kanan AABCD. ..BCD. G.....K GHIIIF GHJ..F LLJMMF --- Search stats --- Nodes expanded : 189 Elapsed time : 0,148 s ----------

2	Algoritma : A* Heuristic : Manhatttan Test case : 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Running A* Search... Papan Awal AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. Gerakan 1: C-atas AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. Gerakan 2: D-atas AABCDF ..BCDF GPP..FK GH.III GHJ... LLJMM. Gerakan 3: P-kanan AABCDF ..BCDF G..PPFK GH.III GHJ... LLJMM. Gerakan 4: I-kiri AABCDF ..BCDF G..PPFK GHIII. GHJ... LLJMM. Gerakan 5: F-bawah AABCD. ..BCD. G..PP.K GHIIIF GHJ..F LLJMMF

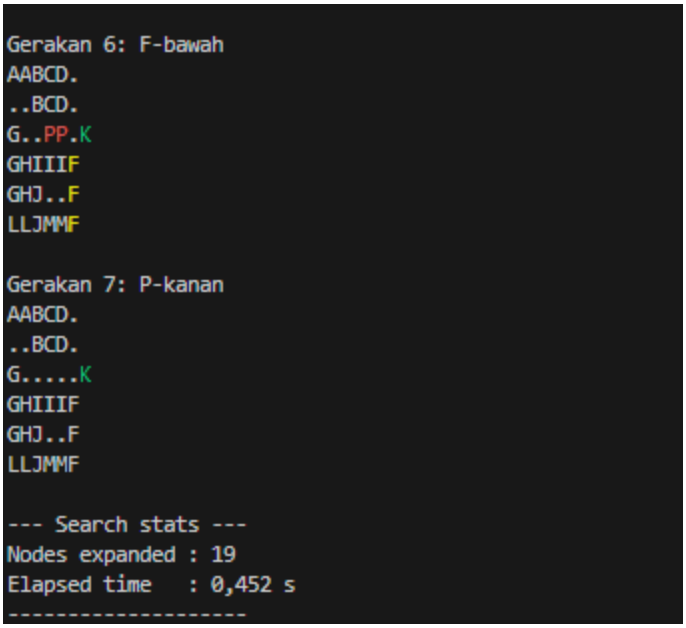
		 Gerakan 6: P-kanan AABCD. ..BCD. G.....K GHIIIF GHJ..F LLJMMF --- Search stats --- Nodes expanded : 137 Elapsed time : 0,328 s ----------

3	Algoritma : A* Heuristic : Blocking Test case : 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Running A* Search... Papan Awal AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. Gerakan 1: I-kiri AAB..F ..BCDF GPPCDFK GHIII. GHJ... LLJMM. Gerakan 2: D-atas AAB.DF ..BCDF GPPC.FK GHIII. GHJ... LLJMM. Gerakan 3: F-bawah AAB.D. ..BCD. GPPC..K GHIIIF GHJ..F LLJMMF Gerakan 4: C-atas AABCD. ..BCD. GPP...K GHIIIF GHJ..F LLJMMF Gerakan 5: P-kanan AABCD. ..BCD. G.....K GHIIIF GHJ..F LLJMMF --- Search stats --- Nodes expanded : 59 Elapsed time : 0,215 s ----------

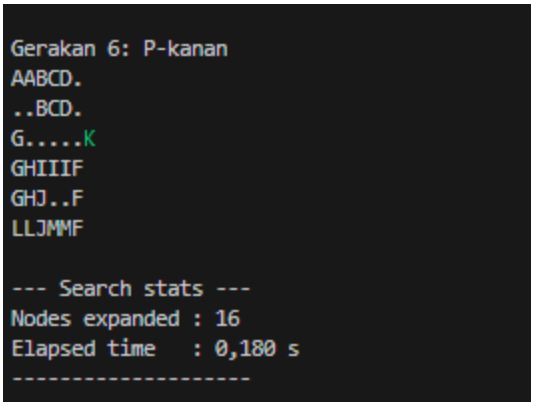
4	Algoritma : A* Heuristic : Combine Test case : 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Running A* Search... Papan Awal AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. Gerakan 1: C-atas AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. Gerakan 2: D-atas AABCDF ..BCDF GPP..FK GH.III GHJ... LLJMM. Gerakan 3: P-kanan AABCDF ..BCDF G..PPFK GH.III GHJ... LLJMM. Gerakan 4: I-kiri AABCDF ..BCDF G..PPFK GHIII. GHJ... LLJMM. Gerakan 5: F-bawah AABCD. ..BCD. G..PP.K GHIIIF GHJ..F LLJMMF

		 Gerakan 6: P-kanan AABCD. ..BCD. G.....K GHIIIF GHJ..F LLJMMF --- Search stats --- Nodes expanded : 18 Elapsed time : 0,257 s ----------

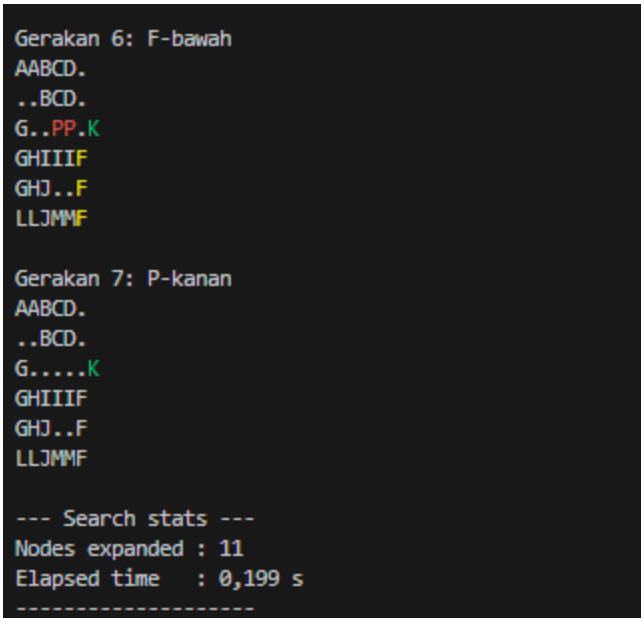
5	Algoritma : GBFS Heuristic : Manhattan Test case : 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Running Greedy Best First Search... Papan Awal AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. Gerakan 1: C-atas AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. Gerakan 2: P-kanan AABC.F ..BCDF G.PPDFK GH.III GHJ... LLJMM. Gerakan 3: D-atas AABCDF ..BCDF G.PP.FK GH.III GHJ... LLJMM. Gerakan 4: P-kanan AABCDF ..BCDF G..PPFK GH.III GHJ... LLJMM. Gerakan 5: I-kiri AABCDF ..BCDF G..PPFK GHIII. GHJ... LLJMM.

		 Gerakan 6: F-bawah AABCD. ..BCD. G..PP.K GHIIIF GHJ..F LLJMMF Gerakan 7: P-kanan AABCD. ..BCD. G.....K GHIIIF GHJ..F LLJMMF --- Search stats --- Nodes expanded : 19 Elapsed time : 0,452 s ----------

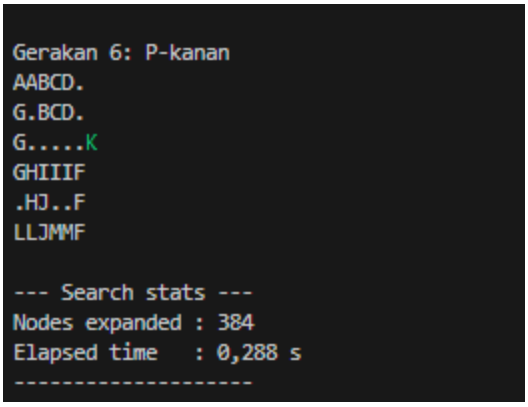
6	Algoritma : GBFS Heuristic : Blocking Test case : 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Running Greedy Best First Search... Papan Awal AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. Gerakan 1: C-atas AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. Gerakan 2: D-atas AABCDF ..BCDF GPP..FK GH.III GHJ... LLJMM. Gerakan 3: I-kiri AABCDF ..BCDF GPP..FK GHIII. GHJ... LLJMM. Gerakan 4: F-bawah AABCD. ..BCD. GPP...K GHIIIF GHJ..F LLJMMF Gerakan 5: P-kanan AABCD. ..BCD. G.PP..K GHIIIF GHJ..F LLJMMF

		 Gerakan 6: P-kanan AABCD. ..BCD. G.....K GHIIIF GHJ..F LLJMMF --- Search stats --- Nodes expanded : 16 Elapsed time : 0,180 s ----------

7	Algoritma : GBFS Heuristic : Combine Test case : 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Running Greedy Best First Search... Papan Awal AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. Gerakan 1: C-atas AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. Gerakan 2: P-kanan AABC.F ..BCDF G.PPDFK GH.III GHJ... LLJMM. Gerakan 3: D-atas AABCDF ..BCDF G.PP.FK GH.III GHJ... LLJMM. Gerakan 4: P-kanan AABCDF ..BCDF G..PPFK GH.III GHJ... LLJMM. Gerakan 5: I-kiri AABCDF ..BCDF G..PPFK GHIII. GHJ... LLJMM.

		 Gerakan 6: F-bawah AABCD. ..BCD. G..PP.K GHIIIF GHJ..F LLJMMF Gerakan 7: P-kanan AABCD. ..BCD. G.....K GHIIIF GHJ..F LLJMMF --- Search stats --- Nodes expanded : 11 Elapsed time : 0,199 s ----------

8	Algoritma : BnB Test case : 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Running Branch and Bound Search... Papan Awal AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. Gerakan 1: G-atas AAB..F G.BCDF GPPCDFK GH.III .HJ... LLJMM. Gerakan 2: D-atas AAB.DF G.BCDF GPPC.FK GH.III .HJ... LLJMM. Gerakan 3: I-kiri AAB.DF G.BCDF GPPC.FK GHIII. .HJ... LLJMM. Gerakan 4: F-bawah AAB.D. G.BCD. GPPC..K GHIIIF .HJ..F LLJMMF Gerakan 5: C-atas AABCD. G.BCD. GPP...K GHIIIF .HJ..F LLJMMF

Hasil Analisis

Algoritma Uniform Cost Search (UCS) merupakan algoritma pencarian yang menjamin solusi optimal karena selalu memilih state dengan cost terendah di setiap langkahnya. Dalam konteks penyelesaian puzzle Rush Hour, UCS terbukti selalu menghasilkan jalur atau solusi terpendek. Namun, kelemahan utama dari UCS adalah jumlah gerakan atau state yang diperiksa selama proses pencarian cenderung sangat banyak. Hal ini disebabkan karena UCS tidak mempertimbangkan arah tujuan sehingga mengeksplorasi banyak kemungkinan secara menyeluruh sebelum mencapai solusi yang mengakibatkan jumlah gerakan atau state yang diperiksa cenderung lebih banyak dibandingkan algoritma lain yang menggunakan pendekatan heuristik.

A* Search merupakan algoritma pencarian yang menggabungkan cost sejauh ini dan estimasi jarak ke tujuan (heuristik) sehingga lebih terarah dibanding UCS. Ketika menggunakan heuristik manhattan atau kombinasi antara heuristik manhattan dan blocking, A* dapat mengurangi jumlah gerakan yang diperiksa dibandingkan UCS. Namun, hasilnya tidak selalu menjamin solusi terpendek. Ketika A* menggunakan heuristik blocking, algoritma ini selalu menghasilkan jalur terpendek sekaligus memeriksa jumlah gerakan yang lebih sedikit daripada UCS. Ini menunjukkan bahwa pemilihan heuristik sangat mempengaruhi kinerja A* dan heuristik blocking terbukti lebih akurat dalam memperkirakan kedekatan terhadap solusi pada kasus Rush Hour.

Greedy Best First Search berfokus hanya pada nilai heuristik tanpa mempertimbangkan cost sejauh ini sehingga lebih cepat dalam menjelajahi state yang dianggap paling dekat ke tujuan. Dalam pengujian dengan heuristik manhattan, blocking, dan kombinasi keduanya, algoritma ini cenderung memeriksa jumlah gerakan yang lebih sedikit dibandingkan UCS dan A*. Namun, hasilnya tidak selalu menghasilkan solusi optimal atau jalur terpendek. Hal ini disebabkan oleh sifat greedy yang hanya fokus pada estimasi heuristik saat ini tanpa mempertimbangkan

langkah-langkah sebelumnya yang kadang membuatnya "terjebak" pada jalur yang tampak menjanjikan namun sebenarnya tidak optimal.

Algoritma Branch and Bound juga termasuk dalam algoritma pencarian berbasis cost, namun dengan pendekatan pemangkasan cabang berdasarkan batas tertentu. Dalam implementasinya untuk Rush Hour, algoritma ini tidak selalu menghasilkan jalur terpendek dan cenderung memeriksa lebih banyak gerakan dibandingkan UCS. Hal ini menunjukkan bahwa Branch and Bound kurang efektif jika tidak dipadukan dengan teknik pemangkasan atau heuristik yang kuat. Tanpa informasi tambahan untuk membatasi eksplorasi, algoritma ini kehilangan keunggulannya dan menjadi kurang efisien baik dari sisi optimalitas maupun eksplorasi.

Penjelasan Bonus

Implementasi *Branch and Bound*
Branch and Bound (BnB) merupakan salah satu teknik pemangkasan (pruning) dalam pencarian jalur optimal. Pada Rush Hour, BnB dibangun di atas kerangka Uniform Cost Search (UCS) sehingga tetap mempertahankan sifat optimalitas tapi mencoba memotong cabang yang sudah pasti lebih mahal daripada solusi terbaik yang pernah ditemui.
Mekanisme Utama

- **Frontier**: Menggunakan `PriorityQueue` yang mengurutkan simpul berdasarkan nilai g(n)g(n)g(n), yaitu total langkah yang telah diambil dari posisi awal hingga simpul tersebut.
- **Upper Bound**: Setelah pertama kali menemukan simpul goal (primary piece mencapai exit), kita rekam cost-nya sebagai `bestCost`.
- **Pruning**: Saat menghasilkan potensi langkah berikutnya (`successors`), jika `next.cost >= bestCost`, simpul itu langsung diabaikan—tidak dimasukkan ke frontier.
- **Terminasi**: Proses tetap berjalan sampai frontier kosong. Jika BnB, bukannya berhenti pada solusi pertama, kita teruskan hingga semua simpul yang tersisa memiliki cost lebih besar dari `bestCost`, lalu akhirnya mengembalikan solution path dengan cost terendah.

Heuristic tambahan
Untuk A* dan Greedy Best-First Search (GBFS), kami menambahkan tiga heuristic pilihan, yang semuanya bersifat *admissible* (tidak melebihkan cost minimum yang sebenarnya):

1. *Manhattan distance*
    - $h_{manhattan} = |kolom\ exit\ -\ max(kolom\ P)|,\ jika\ P\ horizontal.$
    - $h_{manhattan} = |baris\ exit\ -\ max(baris\ P)|,\ jika\ P\ vertikal.$
    - Mengukur jumlah langkah minimal tanpa memperhitungkan *blocker.*
2. *Blocking count*

- $h_{manhattan} = |\{unique\ piece\ antara\ P\ dan\ exit\}|$.
- Setiap *piece* yang menghalangi pasti perlu digerakan minimal sekali.

3. *Combined*
   - $h_{manhattan} = h_{manhattan}(n) + h_{blocking}(n)$.
   - Menambahkan jarak murni dan jumlah *blocker.*

Graphical User Interface (GUI)

Untuk memudahkan pengguna dalam menjalankan dan memvisualisasikan proses penyelesaian Rush Hour, kami menambahkan sebuah antarmuka grafis (GUI) berbasis Swing yang dikemas dalam satu jendela berjudul "Rush Hour Solver". Di bagian atas jendela terdapat panel kontrol dengan tombol **Pilih File…** untuk memuat puzzle dari sistem berkas, dropdown untuk memilih algoritma (UCS, GBFS, A*, atau BnB), dropdown untuk memilih jenis heuristic (Manhattan, Blocking, atau Combined), dan tombol **Jalankan** untuk memulai proses. Setelah file dipilih dan parameter diset, papan permainan asli ditampilkan pada panel tengah dalam bentuk grid dinamis: setiap sel direpresentasikan oleh sebuah `JLabel` yang diwarnai merah untuk primary piece (P), hijau untuk exit (K), abu-abu muda untuk ruang kosong, dan putih untuk blocker. Ketika solver menghasilkan urutan langkah, GUI menggunakan `javax.swing.Timer` untuk menganimasikan pergeseran setiap piece: sel tempat piece bergerak akan diwarnai kuning selama animasi, kemudian kembali ke warna normal di langkah berikutnya. Di akhir animasi ditampilkan statistik singkat—jumlah node yang dieksplorasi dan waktu eksekusi—di sebuah area status sehingga pengguna dapat langsung membandingkan performa berbagai kombinasi algoritma dan heuristic tanpa harus melihat output di console. Jika solver tidak menemukan solusi, GUI akan menampilkan dialog informasi "No solution found" untuk menjaga pengalaman pengguna yang mulus.

Lampiran

Pranala Repository : https://github.com/GhaisanZP/Tucil3_10122074_10122078

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	✔	
2	Program berhasil dijalankan	✔	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	✔	
4	Program dapat membaca masukan berkas .txt 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	✔	