LAPORAN TUGAS KECIL III

IF2211 STRATEGI ALGORITMA

Disusun oleh:

Ivant Samuel Silaban

13523129

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

DAFTAR ISI

BAB I
DESKRIPSI TUGAS

Gambar 1. Rush Hour Puzzle

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan. Setiap kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak dapat berputar. Tujuan utama dari permainan ini adalah memindahkan mobil merah ke pintu keluar dengan jumlah langkah seminimal mungkin.

Komponen penting dari permainan Rush Hour terdiri dari:

1. Papan – Papan merupakan tempat permainan dimainkan.
2. Papan terdiri atas cell, yaitu sebuah singular point dari papan. Sebuah piece akan menempati cell-cell pada papan. Ketika permainan dimulai, semua piece telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi piece dan orientasi, antara horizontal atau vertikal.
3. Hanya primary piece yang dapat digerakkan keluar papan melewati pintu keluar. Piece yang bukan primary piece tidak dapat digerakkan keluar papan. Papan memiliki satu pintu keluar yang pasti berada di dinding papan dan sejajar dengan orientasi primary piece.
4. Piece – Piece adalah sebuah kendaraan di dalam papan. Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece.

Secara standar, variasi ukuran sebuah piece adalah 2-piece (menempati 2 cell) atau 3-piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

5. Primary Piece – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.
6. Pintu Keluar – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan
7. Gerakan — Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

Tugas Kecil 3 IF2211 Strategi Algoritma bertujuan untuk mengimplementasikan algoritma pathfinding, yaitu Uniform Cost Search (UCS), Greedy Best First Search (GBFS), A*, dan algoritma alternatif (IDA* sebagai bonus), untuk menyelesaikan permainan puzzle Rush Hour. Rush Hour adalah permainan logika berbasis grid (biasanya 6x6) di mana pemain menggeser kendaraan untuk membuka jalan bagi mobil utama (primary piece, berwarna merah) menuju pintu keluar dengan langkah seminimal mungkin. Kendaraan hanya dapat bergerak lurus sesuai orientasinya (horizontal atau vertikal) tanpa menembus kendaraan lain.

Program harus membaca konfigurasi papan dari file teks, memungkinkan pengguna memilih algoritma dan heuristik (untuk GBFS, A*, dan IDA*), serta menampilkan jumlah node yang dikunjungi, waktu eksekusi, dan langkah-langkah solusi dengan warna untuk menyoroti primary piece, pintu keluar, dan kendaraan yang digerakkan. Implementasi bonus meliputi algoritma IDA*, heuristik tambahan (Manhattan, Euclidean, Chebyshev, dan kombinasi), serta GUI (tidak diimplementasikan dalam tugas ini). Laporan ini mencakup penjelasan algoritma, analisis, kode sumber, hasil percobaan, dan tautan ke repository

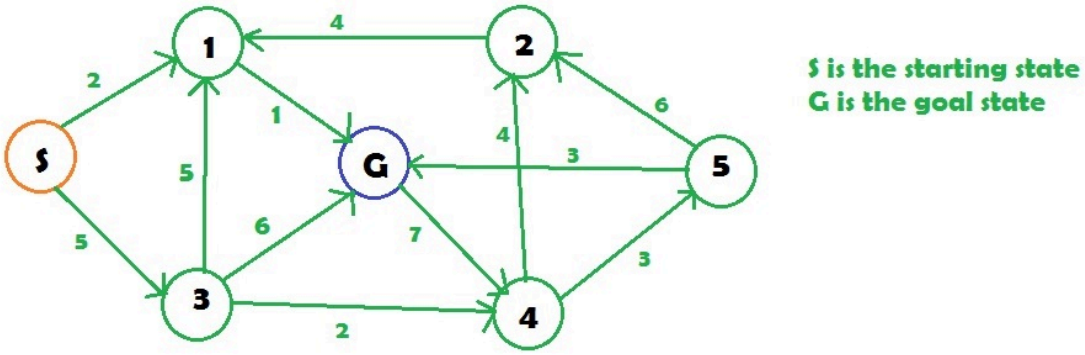
BAB II
LANDASAN TEORI

2.1 Algoritma Pathfinding

Algoritma pathfinding mencari jalur dari keadaan awal ke keadaan tujuan dalam ruang keadaan. Tugas ini menggunakan tiga algoritma utama dan satu algoritma tambahan: Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), dan A* untuk algoritma utama, dan Iterative Deepening A* (IDA*) sebagai algoritma tambahan.

2.1.1 Uniform Cost Search (UCS)

UCS merupakan salah satu algoritma pencarian rute yang merupakan variasi dari algoritma Dijkstra. UCS melakukan pencarian dengan menghitung cost untuk menuju semua simpul yang mungkin dicapai dan memiliki cost paling baik untuk mendapatkan hasil yang paling optimal. Varian Dijkstra ini berguna untuk infinite graph dan graph yang terlalu besar untuk direpresentasikan dalam memori Uniform-Cost Search terutama digunakan dalam AI.

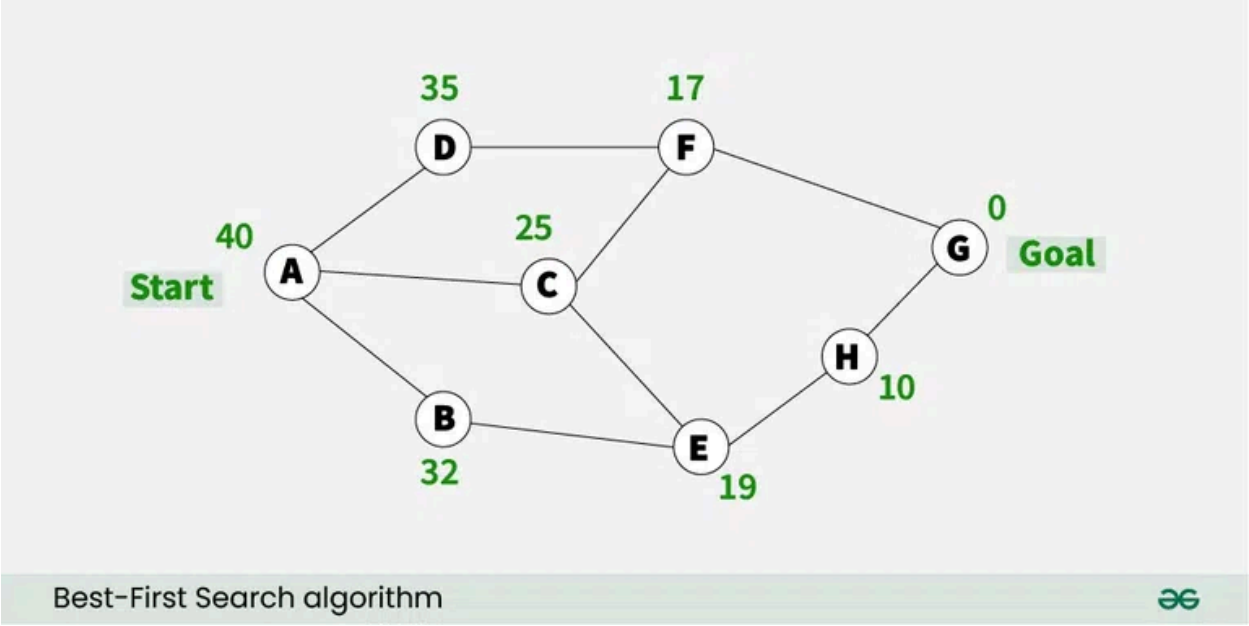


Gambar 2.1.1 Graf Uniform Cost Search

Algoritma UCS dimulai dengan mengunjungi simpul dan menambahkan simpul akar tersebut ke dalam priority queue. Kemudian, simpul akar akan dikunjungi dan dilakukan pengecekan apakah target sudah dicapai. Jika belum, UCS akan menambahkan simpul-simpul yang bertetangga dari simpul akar ke dalam priority queue dengan nilai cost yang dihitung. Priority Queue pada UCS disortir berdasarkan simpul yang memiliki cost yang paling kecil. UCS kemudian mengunjungi simpul dengan cost paling kecil dan mengulangi langkah-langkah seperti yang dilakukan pada simpul akar hingga target ditemukan atau seluruh graph sudah dikunjungi.

2.1.2 Algoritma Greedy Best First Search

Greedy Best First Search atau GBFS merupakan algoritma pencarian rute yang mencari solusi berdasarkan strategi Greedy yaitu menggunakan solusi lokal yang paling baik walaupun belum tentu solusi global yang dihasilkan merupakan solusi yang paling optimal

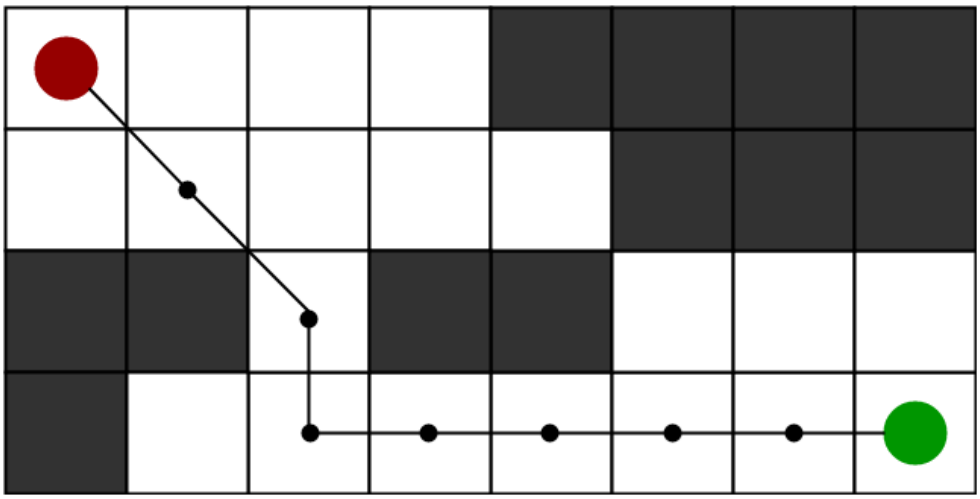


Gambar 2.1.2 Graf Greedy Best First Search

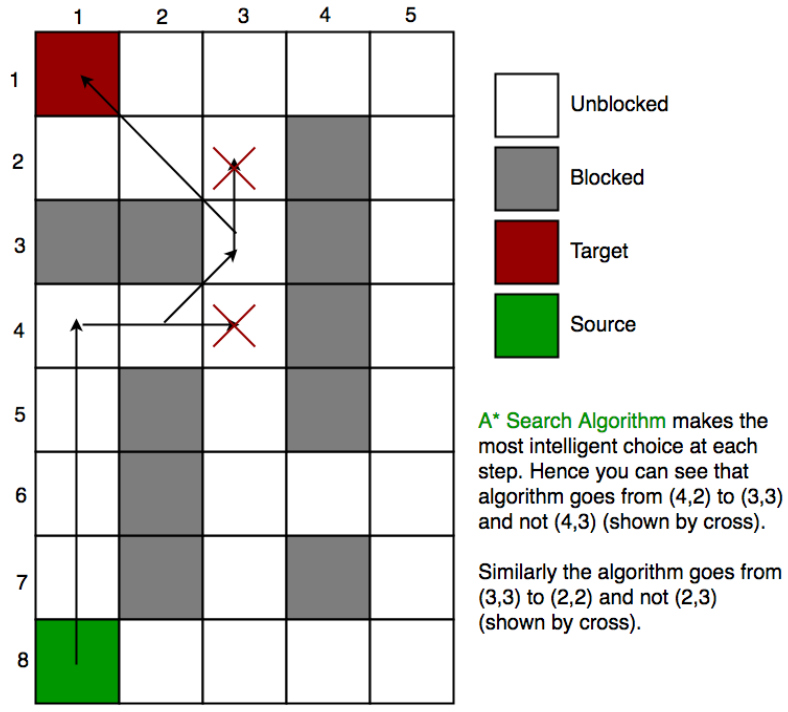
GBFS menentukan solusi lokal terbaik berdasarkan fungsi heuristik untuk menentukan seberapa baik suatu simpul untuk mencapai suatu target. Pada gambar 2.1.2, hasil pencarian rute ke simpul G menggunakan algoritma GBFS adalah A - C - F - G, dimana dari A ke C memiliki nilai heuristik 25, C ke F dengan nilai heuristik 17, dan F ke G memiliki nilai heuristik 0. Maka total nilai heuristik adalah 40 + 25 + 17 + 0 = 82. Rute tersebut belum tentu adalah solusi optimal global karena sifat GBFS yang memiliki rute berdasarkan solusi optimal lokal, tergantung heuristik yang kita gunakan.

2.1.3 Algoritma A*

A* (A Star) adalah salah satu algoritma pencarian terbaik dalam pencarian rute dan transversal graf. Algoritma A* memilih rute terbaik berdasarkan nilai f yang dihitung berdasarkan ketentuan tertentu. Nilai f setara dengan nilai heuristik pada algoritma GBFS, namun nilai f berasal dari dua parameter, yaitu g (cost untuk mencapai suatu simpul dari simpul akar) dan h (cost untuk mencapai simpul target). Mirip dengan algoritma Dijkstra A* menggunakan dua list berbeda, satu untuk menyimpan path saat ini, dan satu untuk menyimpan nilai f untuk setiap simpul yang dapat dikunjungi.



Gambar 2.1.3.1 Gambar Graf A*



Gambar 2.1.3.2 Algoritma A*

2.1.4 Algoritma Iterative Deepening A* (IDA*)

Iterative Deepening A* (IDA*) adalah varian dari Depth First Search (DFS) yang secara berulang memperdalam pencariannya dengan meningkatkan ambang biaya, yang mengendalikan kedalaman eksplorasi. IDA* menggunakan fungsi heuristik untuk mengevaluasi dan memprioritaskan simpul akar yang paling menjanjikan. Hal ini memungkinkannya untuk memangkas jalur yang kurang menjanjikan,

mengurangi penggunaan memori sekaligus memastikan bahwa pencarian difokuskan pada rute yang optimal. Algoritma ini kadang disebut sebagai memory-efficient version of A*.
Algoritma ini bekerja dengan meningkatkan threshold atau ambang batas secara bertahap berdasarkan nilai f setiap node, yang dihitung dengan formula:

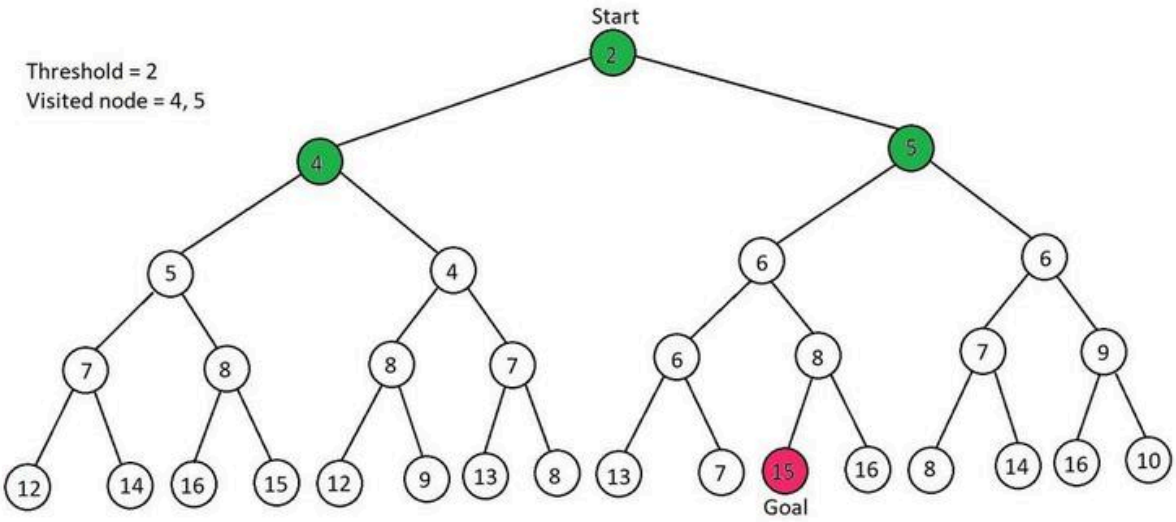
$f(n) = g(n) + h(n)$

$f(n) = Actual\ cost + Estimated\ cost$

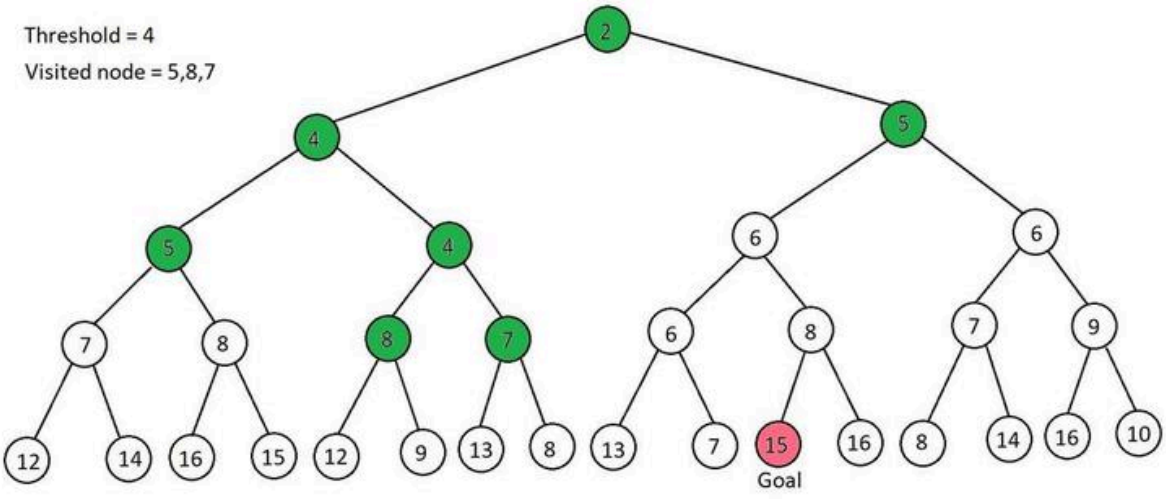
Dimana h adalah admissible atau dapat diterima, dimana:

- f(n) = Fungsi evaluasi biaya total
- g(n) = Biaya aktual dari simpul awal ke simpul saat ini
- h(n) = Estimasi biaya heuristik dari node saat ini ke dalam tujuan. Estimasi ini didasarkan pada perkiraan berdasarkan karakteristik masalah.

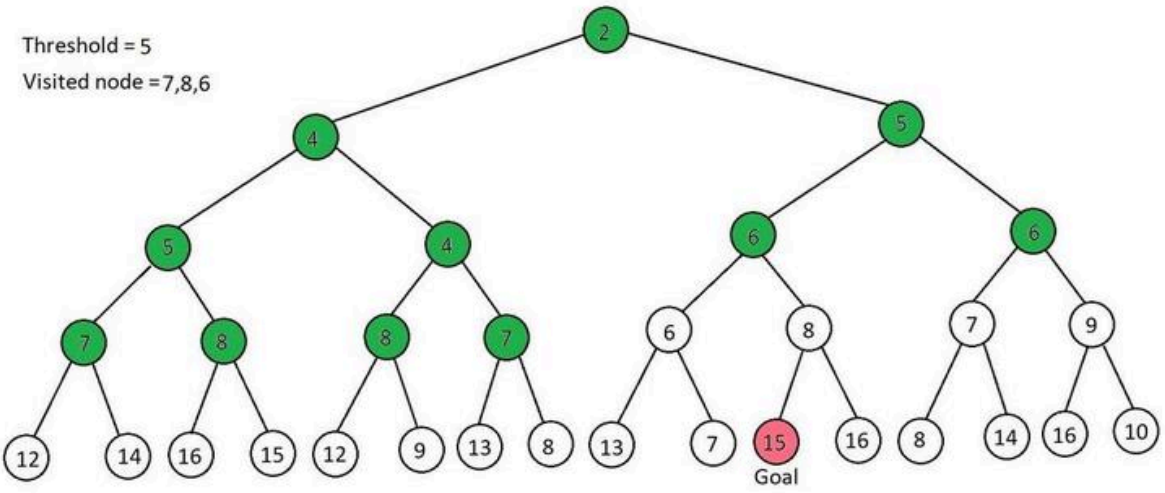
Ilustrasi IDA*:



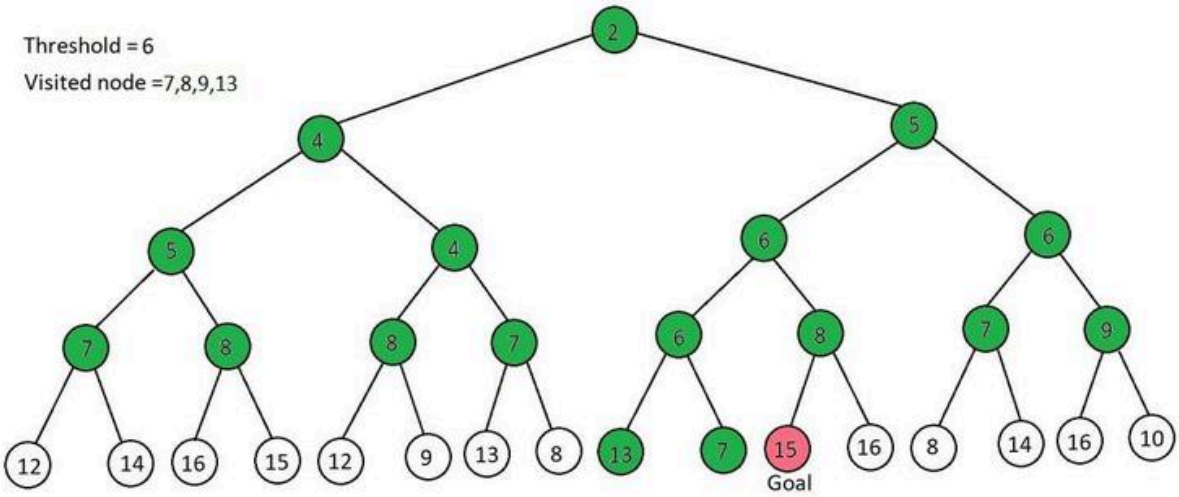
Gambar 2.1.4.1 Iterasi Pertama



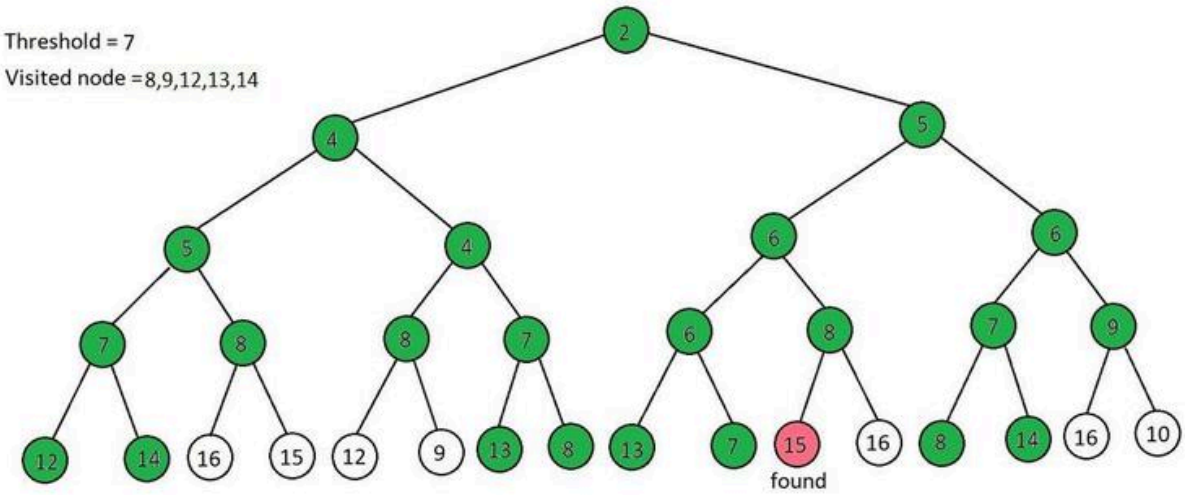
Gambar 2.1.4.2 Iterasi Kedua



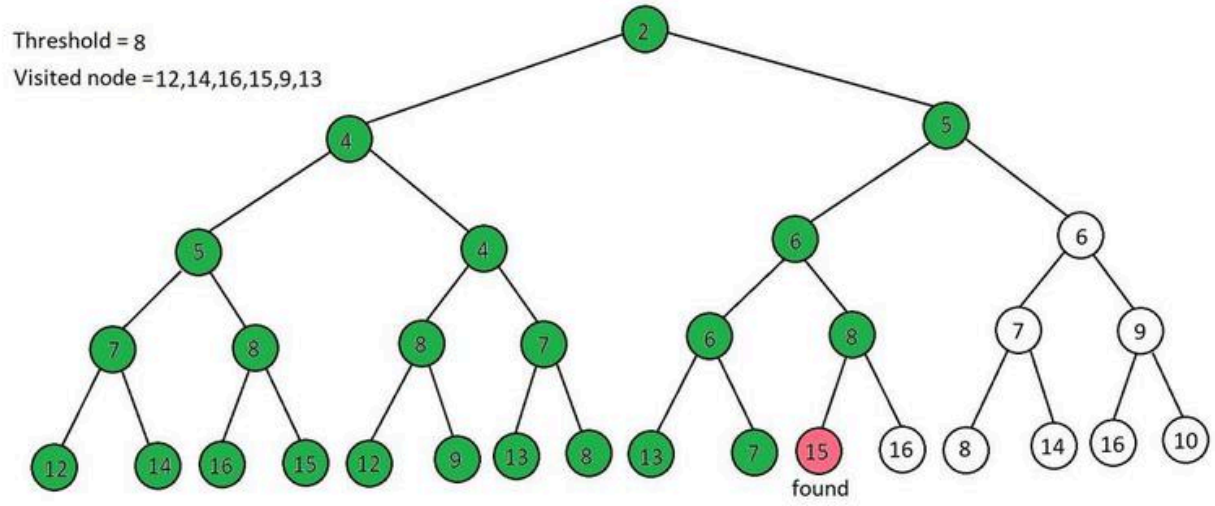
Gambar 2.1.4.3 Iterasi ketiga



Gambar 2.1.4.4 Iterasi keempat



Gambar 2.1.4.5 Iterasi kelima



Gambar 2.1.4.6 Iterasi keenam

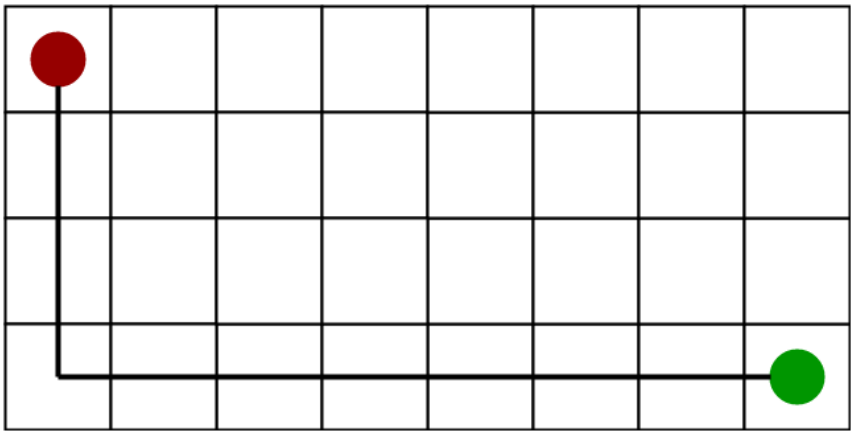
2.2 Heuristic

2.2.1 Manhattan Distance

Manhattan Distance menghitung jumlah langkah horizontal dan vertikal antara dua titik (x1, y1), dan (x2, y2):

Formula: $h(n) = |x2 - x1| + |y2 - y1|$

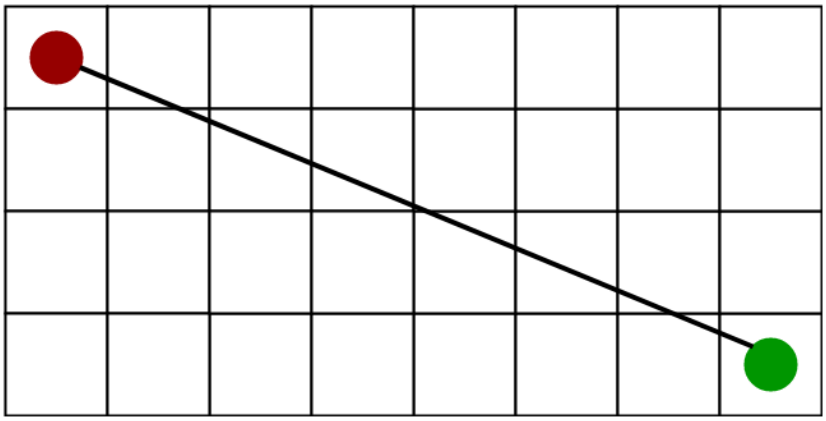
Manhattan Distance dapat dilihat dari ilustrasi berikut:



Gambar 2.2.1 Manhattan Distance Heuristic

2.2.2 Euclidean Distance

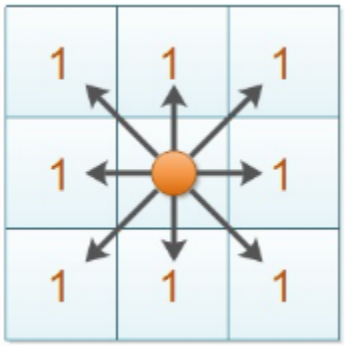
Euclidean Distance menghitung garis lurus antara dua titik.
Formula: $h(n) = \sqrt{((x2 - x1)^2 + (y2 - y1)^2)}$
Euclidean Distance bisa dilihat dari ilustrasi berikut:



Gambar 2.2.2 Euclidean Distance Heuristic

2.2.3 Chebyshev Distance

Chebyshev Distance menghitung jarak maksimum antar koordinat.
Formula: $h(n) = max(|x2 - x1|, |y2 - y1|)$
Chebyshev Distance dapat dilihat dari ilustrasi berikut:



Gambar 2.2.3 Chebyshev Distance Heuristic

BAB III
ANALISIS DAN IMPLEMENTASI

3.1 Algoritma Uniform Cost Search

3.1.1 Implementasi

Dalam penggunaan algoritma Uniform Cost Search (UCS) untuk menyelesaikan Rush Hour Puzzle, penulis mengimplementasikan kelas UCS.java yang mewarisi kelas abstrak Solver. UCS menggunakan biaya (cost) yang merupakan jumlah langkah gerakan untuk mencapai pintu keluar. Berikut langkah-langkah implementasi:

1. Program menginisialisasi papan awal (startBoard) dari kelas Board, yang merepresentasikan konfigurasi kendaraan dan pintu keluar.
2. Program membuat antrian prioritas (PriorityQueue) yang diurutkan berdasarkan biaya (cost) dari status papan (State), yang menyimpan papan, daftar gerakan, dan biaya.
3. Program membuat Set bernama visited untuk mencatat konfigurasi papan yang sudah dikunjungi menggunakan fungsi hash() dari State.java.
4. Status awal dengan biaya 0 dimasukkan ke antrian.
5. Selama antrian tidak kosong, status dengan biaya terkecil diambil.
6. Status tersebut diperiksa apakah primary piece telah mencapai pintu keluar menggunakan isGoal() dari Board.java.
7. Jika tujuan tercapai, program menyimpan jalur gerakan (solutionPath), jumlah node yang dikunjungi, dan waktu eksekusi, lalu berhenti.
8. Jika belum, status ditambahkan ke visited.
9. Program menghasilkan semua gerakan mungkin menggunakan getPossibleMoves() dari Board.java, menciptakan papan baru dengan applyMove() untuk setiap gerakan.
10. Status baru dengan biaya diperbarui (cost + move.amount) ditambahkan ke antrian jika belum dikunjungi.
11. Program mengulangi langkah 5-10 hingga solusi ditemukan atau antrian kosong.

3.1.2 Analisis

Dalam Rush Hour Puzzle, UCS menggunakan biaya berupa jumlah langkah gerakan untuk mencapai pintu keluar, menjamin solusi dengan langkah minimum karena memprioritaskan status dengan biaya terkecil. Implementasi ini mirip dengan Breadth-First Search (BFS) karena setiap gerakan memiliki biaya seragam (1 atau lebih, tergantung jumlah petak), tetapi UCS memungkinkan fleksibilitas untuk biaya variabel (misalnya, gerakan multi-petak). UCS efektif untuk papan dengan ruang pencarian kecil, tetapi pada papan padat dengan banyak kendaraan, eksplorasi semua kemungkinan tanpa heuristik menyebabkan jumlah node yang besar dan waktu eksekusi lama. Penggunaan memori tinggi karena menyimpan semua status di antrian,

membuatnya kurang efisien untuk papan kompleks. UCS cocok untuk Rush Hour jika optimalitas langkah penting, tetapi kurang praktis untuk papan besar dibandingkan algoritma yang menggunakan heuristik.

3.2 Algoritma Greedy Best First Search

3.2.1 Implementasi

Greedy Best First Search (GBFS) diimplementasikan dalam kelas GBFS.java, mewarisi Solver. GBFS memprioritaskan status berdasarkan nilai heuristik (heuristic) dari Heuristic.java. Berikut langkah-langkah implementasi:

1. Program menginisialisasi papan awal (startBoard) dari kelas Board.
2. Program membuat antrian prioritas (PriorityQueue) yang diurutkan berdasarkan nilai heuristik dari status (State).
3. Program membuat Set bernama visited untuk mencatat konfigurasi papan menggunakan hash().
4. Status awal dengan heuristik dihitung (Manhattan, Euclidean, atau Chebyshev Distance) dimasukkan ke antrian.
5. Selama antrian tidak kosong, status dengan heuristik terkecil diambil.
6. Status diperiksa dengan isGoal() untuk menentukan apakah primary piece mencapai pintu keluar.
7. Jika tujuan tercapai, jalur gerakan, jumlah node, dan waktu eksekusi disimpan, lalu program berhenti.
8. Jika belum, status ditambahkan ke visited.
9. Program menghasilkan gerakan mungkin (getPossibleMoves()), menciptakan papan baru (applyMove()), dan menghitung heuristik baru untuk setiap status.
10. Status baru ditambahkan ke antrian jika belum dikunjungi.
11. Langkah 5-10 diulang hingga solusi ditemukan atau antrian kosong.

3.2.2 Analisis

GBFS cepat karena hanya mempertimbangkan nilai heuristik (h(n)), mengarahkan pencarian ke arah pintu keluar, tetapi tidak menjamin solusi optimal karena mengabaikan biaya jalur (g(n)). Dalam Rush Hour, heuristik seperti Manhattan, Euclidean, atau Chebyshev memperkirakan jarak primary piece ke pintu keluar, efektif untuk papan sederhana, tetapi dapat menghasilkan jalur sub-optimal pada papan kompleks dengan banyak kendaraan penghalang. Penggunaan memori lebih rendah dibandingkan UCS karena antrian lebih kecil, tetapi risiko terjebak di jalur tidak optimal tinggi jika heuristik menyesatkan. GBFS cocok untuk Rush Hour ketika kecepatan lebih diutamakan daripada optimalitas, tetapi kurang andal jika tujuan adalah meminimalkan langkah.

3.3 Algoritma A*

3.3.1 Implementasi

Algoritma A* diimplementasikan dalam kelas AStar.java, mewarisi Solver. A* menggunakan fungsi evaluasi f(n) = g(n) + h(n), dengan g(n) sebagai jumlah langkah dan h(n) sebagai heuristik. Berikut langkah-langkahnya:

1. Program menginisialisasi papan awal (startBoard) dari Board.
2. Program membuat antrian prioritas (PriorityQueue) yang diurutkan berdasarkan f(n) dari status (State).
3. Program membuat Set bernama visited untuk mencatat konfigurasi papan via hash().
4. Status awal dengan g(n)=0 dan h(n) dihitung (Manhattan, Euclidean, atau Chebyshev) dimasukkan ke antrian.
5. Selama antrian tidak kosong, status dengan f(n) terkecil diambil.
6. Status diperiksa dengan isGoal() untuk mengecek apakah primary piece mencapai pintu keluar.
7. Jika tujuan tercapai, jalur, node, dan waktu disimpan, lalu program berhenti.
8. Jika belum, status ditambahkan ke visited.
9. Gerakan mungkin dihasilkan (getPossibleMoves()), papan baru dibuat (applyMove()), dan f(n) dihitung untuk status baru.
10. Status baru ditambahkan ke antrian jika belum dikunjungi.
11. Langkah 5-10 diulang hingga solusi ditemukan atau antrian kosong.

3.3.2 Analisis

A* menjamin solusi optimal karena heuristiknya (Manhattan, Euclidean, Chebyshev) admissible, tidak melebih-estimasi biaya sebenarnya. Dalam Rush Hour, A* menyeimbangkan optimalitas (seperti UCS) dan efisiensi (seperti GBFS) dengan mempertimbangkan biaya jalur dan heuristik. A* lebih efisien dari UCS karena heuristik mengurangi jumlah node yang dieksplorasi, tetapi memerlukan memori lebih besar dibandingkan GBFS karena menyimpan antrian besar. Heuristik Manhattan dan Chebyshev lebih cepat dibandingkan Euclidean karena perhitungan sederhana. A* sangat cocok untuk Rush Hour, terutama pada papan sedang hingga besar, tetapi konsumsi memori dapat menjadi kendala pada papan sangat kompleks.

3.4 Algoritma Iterative Deepening A* (IDA*)

3.4.1 Implementasi

Iterative Deepening A* (IDA*) diimplementasikan dalam kelas IDAStar.java, mewarisi Solver. IDA* menggunakan pencarian depth-first dengan batas f(n) = g(n) + h(n). Berikut langkah-langkahnya:

1. Program menginisialisasi papan awal (startBoard) dan menghitung batas awal dari heuristik (Heuristic.estimate()).
2. Program membuat Set bernama visited untuk mencegah siklus dalam iterasi.
3. Dalam metode solve(), program memanggil dfs() dengan batas f(n).
4. Dalam dfs(), status diperiksa dengan isGoal() untuk menentukan tujuan tercapai.
5. Jika f(n) melebihi batas, dfs() mengembalikan nilai f(n) minimum yang melebihi untuk batas berikutnya.
6. Jika belum tujuan, gerakan mungkin dihasilkan (getPossibleMoves()), papan baru dibuat (applyMove()), dan dfs() dipanggil rekursif untuk setiap anak.
7. visited mencatat konfigurasi papan dalam iterasi saat ini.
8. Jika solusi tidak ditemukan, batas diperbarui, dan dfs() diulang.
9. Jika solusi ditemukan, jalur, node, dan waktu disimpan.
10. Program berhenti jika solusi ditemukan atau tidak ada batas baru.

3.4.2 Analisis

IDA* menjamin solusi optimal seperti A* karena heuristiknya admissible, tetapi lebih hemat memori karena hanya menyimpan jalur saat ini, bukan antrian besar. Dalam Rush Hour, IDA* efisien untuk papan besar karena mengurangi kebutuhan memori dibandingkan A* dan UCS, tetapi perhitungan heuristik berulang dapat memperlambat eksekusi pada papan sangat kompleks. Heuristik Manhattan dan Chebyshev lebih cepat dibandingkan Euclidean karena perhitungan sederhana. IDA* sangat cocok untuk Rush Hour dengan ruang pencarian besar, tetapi kinerjanya bergantung pada kualitas heuristik, dan iterasi berulang dapat meningkatkan waktu eksekusi pada kasus dengan banyak kendaraan penghalang.

BAB IV
SOURCE CODE

4.1 Main Program

Main.java – Program untuk CLI

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

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Board board = null;

        System.out.print("Masukkan nama file input: ");
        String filename = scanner.nextLine();

        try (BufferedReader reader = new BufferedReader(new
FileReader("../test/" + filename))) {
            // Baca dimensi baris dan kolom
            String[] dim = reader.readLine().trim().split("\\s+");
            if (dim.length < 2) {
                System.out.println("Error: Format baris pertama
tidak valid.");
                return;
            }

            int rows = Integer.parseInt(dim[0]);
            int cols = Integer.parseInt(dim[1]);

            // Baca jumlah kendaraan
            int n = Integer.parseInt(reader.readLine().trim());

            // Baca seluruh sisa baris file ke list
            List<String> fullLines = new ArrayList<>();
            String line;
            while ((line = reader.readLine()) != null) {
                if (!line.trim().isEmpty()) {
                    fullLines.add(line);
                }
            }

            if (fullLines.size() < rows) {
                System.out.println("Error: File kurang dari " +
rows + " baris konfigurasi.");
```

```
                return;
            }

            // Ambil baris pertama sebanyak rows dan kolom sebanyak
cols sebagai grid
            String[] config = new String[rows];
            for (int i = 0; i < rows; i++) {
                String l = fullLines.get(i);
                if (l.length() < cols) {
                    l = String.format("%-" + cols + "s",
l).replace(' ', '.');
                } else if (l.length() > cols) {
                    l = l.substring(0, cols);
                }
                config[i] = l;
            }

            // Cari posisi 'K'
            int exitRow = -1, exitCol = -1;
            outer:
            for (int i = 0; i < fullLines.size(); i++) {
                int kIndex = fullLines.get(i).indexOf('K');
                if (kIndex >= 0) {
                    exitRow = i;
                    exitCol = kIndex;
                    break outer;
                }
            }

            board = new Board(config);
            board.exitRow = exitRow;
            board.exitCol = exitCol;

            System.out.println("\nPapan Awal:");
            board.print();

            if (board.primaryPiece == null) {
                System.out.println("Error: Primary piece (P) tidak
ditemukan di papan!");
                return;
            }
            if (board.exitRow == -1 || board.exitCol == -1) {
                System.out.println("Error: Pintu keluar (K) tidak
ditemukan di papan!");
                return;
            }

            System.out.println("Primary piece: " +
board.primaryPiece);
            System.out.println("Pintu keluar berada di: (" +
board.exitRow + "," + board.exitCol + ")");
```

```
            if (board.isGoal()) {
                System.out.println("Primary piece sudah mencapai
pintu keluar.");
                return;
            }

            System.out.println("Primary piece belum mencapai pintu
keluar.");
            System.out.println("\nGerakan yang mungkin dari posisi
awal:");
            for (Move move : board.getPossibleMoves()) {
                System.out.println("- " + move);
            }

        } catch (IOException e) {
            System.out.println("Gagal membaca file: " +
e.getMessage());
            return;
        } catch (NumberFormatException e) {
            System.out.println("Error parsing angka dalam file
input: " + e.getMessage());
            return;
        }

        // Algoritma dan Heuristic
        System.out.print("\nPilih algoritma (UCS/GBFS/A*/IDA*): ");
        String algo = scanner.nextLine().trim().toLowerCase();

        String heuristicChoice = "1"; // Default ke Manhattan
        if (!algo.equals("ucs")) {
            System.out.println("Pilih heuristic:");
            System.out.println("1 = Manhattan Distance");
            System.out.println("2 = Euclidean Distance");
            System.out.println("3 = Chebyshev Distance");
            System.out.print("Pilihan (1-3): ");
            heuristicChoice = scanner.nextLine().trim();
            if (!heuristicChoice.matches("[123]")) {
                System.out.println("Heuristic tidak valid.
Menggunakan Manhattan Distance (1).");
                heuristicChoice = "1";
            }
        }

        Solver solver = switch (algo) {
            case "ucs" -> new UCS(board);
            case "gbfs" -> new GBFS(board, heuristicChoice);
            case "a*", "astar" -> new AStar(board,
heuristicChoice);
            case "ida*", "idastar" -> new IDAStar(board,
heuristicChoice);
```

```
            default -> {
                System.out.println("Algoritma tidak dikenali.
Menggunakan A* sebagai default.");
                yield new AStar(board, heuristicChoice);
            }
        };

        if (solver != null) {
            System.out.println("\nMenjalankan solver " + algo +
"...");
            long start = System.nanoTime();
            solver.solve();
            long end = System.nanoTime();
            double durationMs = (end - start) / 1e6;

            List<Move> solution = solver.getSolutionPath();
            Board current = board;

            if (solution == null || solution.isEmpty()) {
                System.out.println("Tidak ditemukan solusi.");
            } else {
                System.out.println("Solusi ditemukan dengan " +
algo.toUpperCase() + "!");
                System.out.println("\nPapan Awal:");
                current.print();

                int step = 1;
                for (Move move : solution) {
                    System.out.println("Gerakan " + step + ": " +
move);
                    current = current.applyMove(move);
                    current.printWithHighlight(move);
                    step++;
                }
                System.out.println("Jumlah node dikunjungi: " +
solver.getVisitedCount());
                System.out.println("Jumlah langkah: " +
solution.size());
                System.out.printf("Waktu eksekusi: %.2f ms\n",
durationMs);
            }
        }

        scanner.close();
    }
}
```

MainGUI.java – Program untuk GUI

```
import javax.swing.*;
import javax.swing.border.EmptyBorder;
import javax.swing.filechooser.FileNameExtensionFilter;
import java.awt.*;
import java.awt.datatransfer.DataFlavor;
import java.awt.event.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class MainGUI extends JFrame {
    private Board board;
    private Solver solver;
    private List<Move> solutionPath;
    private int currentStep;
    private JLabel[][] boardLabels;
    private JTextArea outputArea;
    private JComboBox<String> algoComboBox;
    private JComboBox<String> heuristicComboBox;
    private JPanel boardPanel;
    private JButton nextStepButton;
    private JButton prevStepButton;
    private JButton playButton;
    private Timer animationTimer;
    private JTextField rowsField;
    private JTextField colsField;
    private JTextField[][] inputGrid;
    private JPanel inputPanel;
    private JComboBox<String> edgeComboBox;
    private JComboBox<Integer> exitPositionComboBox;

    private static final Color PRIMARY_COLOR = Color.BLUE;
    private static final Color EXIT_COLOR = Color.RED;
    private static final Color MOVE_COLOR = Color.GREEN;
    private static final Color EMPTY_COLOR = Color.WHITE;
    private static final Color PIECE_COLOR = Color.GRAY;

    public MainGUI() {
        setTitle("Rush Hour Solver");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(900, 700);
        setLocationRelativeTo(null);

        // Main panel with BorderLayout
        JPanel mainPanel = new JPanel(new BorderLayout(10, 10));
        mainPanel.setBorder(new EmptyBorder(10, 10, 10, 10));
        setContentPane(mainPanel);
```

```
        // Top panel for controls
        JPanel controlPanel = new JPanel(new
FlowLayout(FlowLayout.LEFT));
        mainPanel.add(controlPanel, BorderLayout.NORTH);

        // File chooser button
        JButton loadButton = new JButton("Load Board (.txt)");
        controlPanel.add(loadButton);

        // Algorithm selection
        String[] algorithms = {"UCS", "GBFS", "A*", "IDA*"};
        algoComboBox = new JComboBox<>(algorithms);
        controlPanel.add(new JLabel("Algorithm:"));
        controlPanel.add(algoComboBox);

        // Heuristic selection
        String[] heuristics = {"Manhattan", "Euclidean",
"Chebyshev"};
        heuristicComboBox = new JComboBox<>(heuristics);
        controlPanel.add(new JLabel("Heuristic:"));
        controlPanel.add(heuristicComboBox);

        // Add listener to show/hide heuristic based on algorithm
selection
        algoComboBox.addActionListener(e -> {
            String selectedAlgo = (String)
algoComboBox.getSelectedItem();
            boolean showHeuristic = !selectedAlgo.equals("UCS");
            heuristicComboBox.setVisible(showHeuristic);

controlPanel.getComponent(controlPanel.getComponentCount() -
2).setVisible(showHeuristic); // Label
        });

        // Run button
        JButton runButton = new JButton("Run Solver");
        controlPanel.add(runButton);

        // Center panel with tabs for input and board display
        JTabbedPane centerPane = new JTabbedPane();
        mainPanel.add(centerPane, BorderLayout.CENTER);

        // Input panel for graphical configuration
        inputPanel = new JPanel(new BorderLayout(5, 5));
        centerPane.addTab("Input Board", inputPanel);

        // Input controls
        JPanel inputControlPanel = new JPanel(new FlowLayout());
        inputPanel.add(inputControlPanel, BorderLayout.NORTH);

        inputControlPanel.add(new JLabel("Rows:"));
```

```
        rowsField = new JTextField("6", 3);
        inputControlPanel.add(rowsField);

        inputControlPanel.add(new JLabel("Cols:"));
        colsField = new JTextField("6", 3);
        inputControlPanel.add(colsField);

        JButton initGridButton = new JButton("Initialize Grid");
        inputControlPanel.add(initGridButton);

        JButton clearGridButton = new JButton("Clear Grid");
        inputControlPanel.add(clearGridButton);

        inputControlPanel.add(new JLabel("Exit Edge:"));
        edgeComboBox = new JComboBox<>(new String[]{"Top",
"Bottom", "Left", "Right"});
        inputControlPanel.add(edgeComboBox);

        inputControlPanel.add(new JLabel("Position:"));
        exitPositionComboBox = new JComboBox<>();
        inputControlPanel.add(exitPositionComboBox);

        JButton submitButton = new JButton("Submit Configuration");
        inputControlPanel.add(submitButton);

        // Instruction label
        JLabel instructionLabel = new JLabel("Enter pieces (e.g.,
PP or PPP for primary piece, AA for others, . for empty)");
        instructionLabel.setFont(new Font("Arial", Font.ITALIC,
12));
        inputPanel.add(instructionLabel, BorderLayout.SOUTH);

        // Board display panel
        boardPanel = new JPanel();
        boardPanel.setBackground(Color.LIGHT_GRAY);
        centerPane.addTab("Board Display", boardPanel);

        // Output panel
        outputArea = new JTextArea(10, 30);
        outputArea.setEditable(false);
        outputArea.setFont(new Font("Monospaced", Font.PLAIN, 12));
        mainPanel.add(new JScrollPane(outputArea),
BorderLayout.EAST);

        // Bottom panel for animation controls
        JPanel animationPanel = new JPanel(new FlowLayout());
        animationPanel.add(new JLabel("Animation Controls:"));
        prevStepButton = new JButton("Previous");
        animationPanel.add(prevStepButton);
        nextStepButton = new JButton("Next");
        animationPanel.add(nextStepButton);
```

```
        playButton = new JButton("Play");
        animationPanel.add(playButton);
        mainPanel.add(animationPanel, BorderLayout.SOUTH);

        // Disable buttons initially
        prevStepButton.setEnabled(false);
        nextStepButton.setEnabled(false);
        playButton.setEnabled(false);
        submitButton.setEnabled(false);

        // Drag-and-drop support for board panel
        boardPanel.setTransferHandler(new TransferHandler() {
            @Override
            public boolean canImport(TransferSupport support) {
                return
support.isDataFlavorSupported(DataFlavor.javaFileListFlavor);
            }

            @Override
            public boolean importData(TransferSupport support) {
                try {
                    @SuppressWarnings("unchecked")
                    List<File> files = (List<File>)
support.getTransferable().getTransferData(DataFlavor.javaFileListFl
avor);
                    if (!files.isEmpty()) {
                        File file = files.get(0);
                        if (file.getName().endsWith(".txt")) {
                            loadBoardFromFile(file);
                            centerPane.setSelectedIndex(1); //
Switch to board display
                            return true;
                        }
                    }
                } catch (Exception e) {
                    JOptionPane.showMessageDialog(MainGUI.this,
"Error loading file: " + e.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
                }
                return false;
            }
        });

        // Load button action
        loadButton.addActionListener(e -> {
            JFileChooser fileChooser = new JFileChooser("../test");
            fileChooser.setFileFilter(new
FileNameExtensionFilter("Text Files", "txt"));
            if (fileChooser.showOpenDialog(MainGUI.this) ==
JFileChooser.APPROVE_OPTION) {
                loadBoardFromFile(fileChooser.getSelectedFile());
```

```
                centerPane.setSelectedIndex(1);
            }
        });

        // Initialize grid button action
        initGridButton.addActionListener(e ->
initializeInputGrid());

        // Clear grid button action
        clearGridButton.addActionListener(e -> clearInputGrid());

        // Edge selection action
        edgeComboBox.addActionListener(e ->
updateExitPositionComboBox());

        // Submit configuration button action
        submitButton.addActionListener(e -> submitConfiguration());

        // Run button action
        runButton.addActionListener(e -> runSolver());

        // Animation controls
        prevStepButton.addActionListener(e -> showPreviousStep());
        nextStepButton.addActionListener(e -> showNextStep());
        playButton.addActionListener(e -> toggleAnimation());

        // Animation timer (500ms per step)
        animationTimer = new Timer(500, e -> showNextStep());
        animationTimer.setRepeats(true);
    }

    private void initializeInputGrid() {
        try {
            int rows =
Integer.parseInt(rowsField.getText().trim());
            int cols =
Integer.parseInt(colsField.getText().trim());
            if (rows <= 0 || cols <= 0) {
                throw new IllegalArgumentException("Rows and
columns must be positive");
            }

            inputPanel.removeAll();
            inputPanel.setLayout(new BorderLayout(5, 5));

            // Re-add control panel
            JPanel inputControlPanel = new JPanel(new
FlowLayout());
            inputControlPanel.add(new JLabel("Rows:"));
            inputControlPanel.add(rowsField);
            inputControlPanel.add(new JLabel("Cols:"));
```

```
            inputControlPanel.add(colsField);
            JButton initGridButton = new JButton("Initialize
Grid");
            inputControlPanel.add(initGridButton);
            initGridButton.addActionListener(e ->
initializeInputGrid());
            JButton clearGridButton = new JButton("Clear Grid");
            inputControlPanel.add(clearGridButton);
            clearGridButton.addActionListener(e ->
clearInputGrid());
            inputControlPanel.add(new JLabel("Exit Edge:"));
            edgeComboBox = new JComboBox<>(new String[]{"Top",
"Bottom", "Left", "Right"});
            inputControlPanel.add(edgeComboBox);
            inputControlPanel.add(new JLabel("Position:"));
            exitPositionComboBox = new JComboBox<>();
            inputControlPanel.add(exitPositionComboBox);
            edgeComboBox.addActionListener(e ->
updateExitPositionComboBox());
            JButton submitButton = new JButton("Submit
Configuration");
            inputControlPanel.add(submitButton);
            submitButton.addActionListener(e ->
submitConfiguration());
            inputPanel.add(inputControlPanel, BorderLayout.NORTH);

            // Create input grid
            JPanel gridPanel = new JPanel(new GridLayout(rows,
cols));
            inputGrid = new JTextField[rows][cols];
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                    inputGrid[i][j] = new JTextField(".", 1);

inputGrid[i][j].setHorizontalAlignment(JTextField.CENTER);
                    inputGrid[i][j].setFont(new Font("Monospaced",
Font.PLAIN, 16));
                    gridPanel.add(inputGrid[i][j]);
                }
            }
            inputPanel.add(gridPanel, BorderLayout.CENTER);

            // Re-add instruction label
            JLabel instructionLabel = new JLabel("Enter pieces
(e.g., PP or PPP for primary piece, AA for others, . for empty)");
            instructionLabel.setFont(new Font("Arial", Font.ITALIC,
12));
            inputPanel.add(instructionLabel, BorderLayout.SOUTH);

            // Update exit position options
            updateExitPositionComboBox();
```

```
            submitButton.setEnabled(true);
            inputPanel.revalidate();
            inputPanel.repaint();
            outputArea.setText("Input grid initialized (" + rows +
"x" + cols + "). Enter pieces (e.g., PP or PPP for primary piece,
AA for others, . for empty).\n");
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(this, "Error: " +
e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        } catch (IllegalArgumentException e) {
            JOptionPane.showMessageDialog(this, "Error: " +
e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    private void clearInputGrid() {
        if (inputGrid != null) {
            for (JTextField[] row : inputGrid) {
                for (JTextField field : row) {
                    field.setText(".");
                }
            }
            outputArea.setText("Input grid cleared. Enter new
configuration.\n");
        }
    }

    private void updateExitPositionComboBox() {
        String edge = (String) edgeComboBox.getSelectedItem();
        exitPositionComboBox.removeAllItems();
        try {
            int rows =
Integer.parseInt(rowsField.getText().trim());
            int cols =
Integer.parseInt(colsField.getText().trim());
            if (edge.equals("Top") || edge.equals("Bottom")) {
                for (int j = 0; j < cols; j++) {
                    exitPositionComboBox.addItem(j);
                }
            } else { // Left or Right
                for (int i = 0; i < rows; i++) {
                    exitPositionComboBox.addItem(i);
                }
            }
        } catch (NumberFormatException e) {
            // Ignore if dimensions are not set
        }
    }

    private void submitConfiguration() {
```

```
        try {
            int rows =
Integer.parseInt(rowsField.getText().trim());
            int cols =
Integer.parseInt(colsField.getText().trim());

            // Read grid configuration
            String[] config = new String[rows];
            for (int i = 0; i < rows; i++) {
                StringBuilder row = new StringBuilder();
                for (int j = 0; j < cols; j++) {
                    String text = inputGrid[i][j].getText().trim();
                    if (text.isEmpty()) text = ".";
                    if (text.length() > 1 || (!text.equals(".") &&
!Character.isLetter(text.charAt(0)))) {
                        throw new IllegalArgumentException("Invalid
character at (" + i + "," + j + "). Use letters (A-Z, P) or .");
                    }
                    row.append(text);
                }
                config[i] = row.toString();
            }

            // Initialize board
            board = new Board(config);

            // Validate primary piece
            if (board.primaryPiece == null) {
                throw new IllegalArgumentException("Primary piece
(P) not found. Ensure it is 2 or 3 cells long (e.g., PP or PPP for
primary piece, AA for others, . for empty).");
            }
            if (board.primaryPiece.length < 2 ||
board.primaryPiece.length > 3) {
                throw new IllegalArgumentException("Primary piece
(P) has invalid length: " + board.primaryPiece.length + ". Use 2 or
3 cells (e.g., PP or PPP).");
            }

            // Validate all pieces
            for (Piece piece : board.pieces) {
                if (piece.length < 2 || piece.length > 3) {
                    throw new IllegalArgumentException("Piece " +
piece.id + " has invalid length: " + piece.length + ". Pieces must
be 2 or 3 cells long.");
                }
            }

            // Count non-primary pieces
            int n = board.pieces.size() - 1; // Subtract primary
piece
```

```
            // Get exit position
            String edge = (String) edgeComboBox.getSelectedItem();
            Integer position = (Integer)
exitPositionComboBox.getSelectedItem();
            if (edge == null || position == null) {
                throw new IllegalArgumentException("Please select
an exit edge and position.");
            }

            int exitRow = -1, exitCol = -1;
            if (edge.equals("Top")) {
                exitRow = -1;
                exitCol = position;
            } else if (edge.equals("Bottom")) {
                exitRow = rows;
                exitCol = position;
            } else if (edge.equals("Left")) {
                exitRow = position;
                exitCol = -1;
            } else if (edge.equals("Right")) {
                exitRow = position;
                exitCol = cols;
            }

            // Validate exit alignment with primary piece
            boolean isHorizontal = board.primaryPiece.isHorizontal;
            boolean validExit = false;
            if (isHorizontal && (edge.equals("Left") ||
edge.equals("Right"))) {
                validExit = true;
            } else if (!isHorizontal && (edge.equals("Top") ||
edge.equals("Bottom"))) {
                validExit = true;
            }
            if (!validExit) {
                throw new IllegalArgumentException("Exit edge (" +
edge + ") must align with primary piece orientation (" +
(isHorizontal ? "horizontal" : "vertical") + "). Use " +
(isHorizontal ? "Left/Right" : "Top/Bottom") + " for " +
(isHorizontal ? "horizontal" : "vertical") + " primary piece.");
            }

            // Validate exit position bounds
            if ((edge.equals("Top") || edge.equals("Bottom")) &&
(exitCol < 0 || exitCol >= cols)) {
                throw new IllegalArgumentException("Exit position "
+ exitCol + " is out of bounds for " + edge + " edge.");
            }
            if ((edge.equals("Left") || edge.equals("Right")) &&
(exitRow < 0 || exitRow >= rows)) {
```

```
                throw new IllegalArgumentException("Exit position "
+ exitRow + " is out of bounds for " + edge + " edge.");
            }

            board.exitRow = exitRow;
            board.exitCol = exitCol;

            // Display board
            displayBoard(board, null);
            outputArea.append("Configuration loaded
successfully.\n");
            outputArea.append("Board: " + rows + "x" + cols + ",
Non-primary pieces: " + n + "\n");
            outputArea.append("Primary piece: " +
board.primaryPiece + " (" + (isHorizontal ? "horizontal" :
"vertical") + ", length=" + board.primaryPiece.length + ")\n");
            outputArea.append("Exit at: (" + exitRow + "," +
exitCol + ") on " + edge + " edge\n");
            prevStepButton.setEnabled(false);
            nextStepButton.setEnabled(false);
            playButton.setEnabled(false);
            solutionPath = null;
            currentStep = 0;

        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, "Error: " +
e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            board = null;
        }
    }

    private void loadBoardFromFile(File file) {
        try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
            String[] dim = reader.readLine().trim().split("\\s+");
            if (dim.length < 2) {
                throw new IllegalArgumentException("Invalid
dimension format.");
            }
            int rows = Integer.parseInt(dim[0]);
            int cols = Integer.parseInt(dim[1]);

            int n = Integer.parseInt(reader.readLine().trim());

            List<String> lines = new ArrayList<>();
            String line;
            while ((line = reader.readLine()) != null) {
                if (!line.trim().isEmpty()) {
                    lines.add(line);
                }
            }
```

```
            if (lines.size() < rows) {
                throw new IllegalArgumentException("File has fewer
than " + rows + " configuration lines.");
            }

            String[] config = new String[rows];
            for (int i = 0; i < rows; i++) {
                String l = lines.get(i);
                if (l.length() < cols) {
                    l = String.format("%-" + cols + "s",
l).replace(' ', '.');
                } else if (l.length() > cols) {
                    l = l.substring(0, cols);
                }
                config[i] = l;
            }

            int exitRow = -1, exitCol = -1;
            for (int i = 0; i < lines.size(); i++) {
                int kIndex = lines.get(i).indexOf('K');
                if (kIndex >= 0) {
                    exitRow = i;
                    exitCol = kIndex;
                    break;
                }
            }

            board = new Board(config);
            if (board.primaryPiece == null) {
                throw new IllegalArgumentException("Primary piece
(P) not found in file. Ensure it is 2 or 3 cells long (e.g., PP or
PPP).");
            }
            if (board.primaryPiece.length < 2 ||
board.primaryPiece.length > 3) {
                throw new IllegalArgumentException("Primary piece
(P) has invalid length: " + board.primaryPiece.length + ". Use 2 or
3 cells.");
            }

            board.exitRow = exitRow;
            board.exitCol = exitCol;

            if (board.exitRow == -1 && board.exitCol == -1) {
                throw new IllegalArgumentException("Exit (K) not
found.");
            }

            displayBoard(board, null);
            outputArea.setText("Board loaded from file
```

```
successfully.\nPrimary piece: " + board.primaryPiece + "\nExit at:
(" + board.exitRow + "," + board.exitCol + ")\n");
            prevStepButton.setEnabled(false);
            nextStepButton.setEnabled(false);
            playButton.setEnabled(false);
            solutionPath = null;
            currentStep = 0;

        } catch (IOException e) {
            JOptionPane.showMessageDialog(this, "Error loading
file: " + e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            board = null;
            boardPanel.removeAll();
            boardPanel.revalidate();
            boardPanel.repaint();
            outputArea.setText("");
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(this, "Error loading
file: " + e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            board = null;
            boardPanel.removeAll();
            boardPanel.revalidate();
            boardPanel.repaint();
            outputArea.setText("");
        } catch (IllegalArgumentException e) {
            JOptionPane.showMessageDialog(this, "Error loading
file: " + e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            board = null;
            boardPanel.removeAll();
            boardPanel.revalidate();
            boardPanel.repaint();
            outputArea.setText("");
        }
    }

    private void displayBoard(Board board, Move highlightMove) {
        boardPanel.removeAll();
        boardPanel.setLayout(new GridLayout(board.rows,
board.cols));
        boardLabels = new JLabel[board.rows][board.cols];

        for (int i = 0; i < board.rows; i++) {
            for (int j = 0; j < board.cols; j++) {
                char ch = board.grid[i][j];
                JLabel label = new JLabel(String.valueOf(ch),
SwingConstants.CENTER);
                label.setOpaque(true);
                label.setFont(new Font("Monospaced", Font.BOLD,
16));

label.setBorder(BorderFactory.createLineBorder(Color.BLACK));
```

```
                if (ch == 'K') {
                    label.setBackground(EXIT_COLOR);
                } else if (ch == 'P') {
                    label.setBackground(PRIMARY_COLOR);
                    label.setForeground(Color.WHITE);
                } else if (highlightMove != null && ch ==
highlightMove.pieceId) {
                    label.setBackground(MOVE_COLOR);
                } else if (ch == '.') {
                    label.setBackground(EMPTY_COLOR);
                } else {
                    label.setBackground(PIECE_COLOR);
                }

                boardLabels[i][j] = label;
                boardPanel.add(label);
            }
        }

        boardPanel.revalidate();
        boardPanel.repaint();
    }

    private void runSolver() {
        if (board == null) {
            JOptionPane.showMessageDialog(this, "Please load or
create a board first.", "Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        String selectedAlgo = (String)
algoComboBox.getSelectedItem();
        String selectedHeuristic = (String)
heuristicComboBox.getSelectedItem();
        String heuristicChoice = switch (selectedHeuristic) {
            case "Manhattan" -> "1";
            case "Euclidean" -> "2";
            case "Chebyshev" -> "3";
            default -> "1";
        };

        solver = switch (selectedAlgo) {
            case "UCS" -> new UCS(board);
            case "GBFS" -> new GBFS(board, heuristicChoice);
            case "A*" -> new AStar(board, heuristicChoice);
            case "IDA*" -> new IDAStar(board, heuristicChoice);
            default -> new AStar(board, heuristicChoice);
        };

        // Clear previous solution
```

```
        solutionPath = null;
        currentStep = 0;
        outputArea.setText("");

        // Run solver
        long startTime = System.currentTimeMillis();
        solver.solve();
        long endTime = System.currentTimeMillis();

        solutionPath = solver.getSolutionPath();
        if (solutionPath != null && !solutionPath.isEmpty()) {
            // Enable animation controls
            prevStepButton.setEnabled(false);
            nextStepButton.setEnabled(true);
            playButton.setEnabled(true);

            // Display initial board
            displayBoard(board, null);

            // Show solution info
            outputArea.append("Solution found!\n");
            outputArea.append("Algorithm: " + selectedAlgo + "\n");
            if (!selectedAlgo.equals("UCS")) {
                outputArea.append("Heuristic: " + selectedHeuristic
+ "\n");
            }
            outputArea.append("Nodes visited: " +
solver.getVisitedCount() + "\n");
            outputArea.append("Solution length: " +
solutionPath.size() + "\n");
            outputArea.append("Time: " + (endTime - startTime) + "
ms\n");
        } else {
            outputArea.append("No solution found.\n");
            prevStepButton.setEnabled(false);
            nextStepButton.setEnabled(false);
            playButton.setEnabled(false);
        }
    }

    private void showPreviousStep() {
        if (currentStep > 0) {
            currentStep--;
            Board currentBoard = board;
            for (int i = 0; i < currentStep; i++) {
                currentBoard =
currentBoard.applyMove(solutionPath.get(i));
            }
            Move highlight = currentStep > 0 ?
solutionPath.get(currentStep - 1) : null;
            displayBoard(currentBoard, highlight);
```

```
            outputArea.append("Step " + currentStep + ": " +
(highlight != null ? highlight : "Initial board") + "\n");
            nextStepButton.setEnabled(true);
            prevStepButton.setEnabled(currentStep > 0);
            if (animationTimer.isRunning()) {
                animationTimer.stop();
                playButton.setText("Play");
            }
        }
    }

    private void showNextStep() {
        if (currentStep < solutionPath.size()) {
            Board currentBoard = board;
            for (int i = 0; i < currentStep; i++) {
                currentBoard =
currentBoard.applyMove(solutionPath.get(i));
            }
            Move highlight = currentStep < solutionPath.size() ?
solutionPath.get(currentStep) : null;
            displayBoard(currentBoard, highlight);
            outputArea.append("Step " + (currentStep + 1) + ": " +
(highlight != null ? highlight : "Final board") + "\n");
            currentStep++;
            prevStepButton.setEnabled(true);
            nextStepButton.setEnabled(currentStep <
solutionPath.size());
            if (currentStep >= solutionPath.size() &&
animationTimer.isRunning()) {
                animationTimer.stop();
                playButton.setText("Play");
            }
        }
    }

    private void toggleAnimation() {
        if (animationTimer.isRunning()) {
            animationTimer.stop();
            playButton.setText("Play");
        } else {
            if (currentStep >= solutionPath.size()) {
                currentStep = 0;
                displayBoard(board, null);
                outputArea.append("Restarting animation...\n");
            }
            animationTimer.start();
            playButton.setText("Stop");
        }
    }

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

```
        SwingUtilities.invokeLater(() -> new
MainGUI().setVisible(true));
    }
}
```

4.2 Tools dan Utilitas

Board.java

```
import java.util.*;

public class Board {
    public static boolean DEBUG = false;

    public char[][] grid;
    public List<Piece> pieces = new ArrayList<>();
    public Piece primaryPiece;
    public int exitRow = -1, exitCol = -1;
    public int rows, cols;
    public char primaryVehicleId;

    public Board(String[] config) {
        this.rows = config.length;
        this.cols = config[0].length();
        this.grid = new char[rows][cols];

        if (DEBUG) {
            System.out.println("DEBUG: Konfigurasi input:");
            for (String line : config) {
                System.out.println(line);
            }
        }

        Map<Character, List<int[]>> positions = new HashMap<>();

        for (int i = 0; i < rows; i++) {
            String line = config[i];
            for (int j = 0; j < cols; j++) {
                char c = line.charAt(j);
                grid[i][j] = c;

                if (c != '.' && c != 'K') {
                    positions.putIfAbsent(c, new ArrayList<>());
                    positions.get(c).add(new int[]{i, j});
                }
            }
        }
```

```
        pieces.clear();
        primaryPiece = null;

        for (Map.Entry<Character, List<int[]>> entry :
positions.entrySet()) {
            char id = entry.getKey();
            List<int[]> coords = entry.getValue();

            coords.sort(Comparator.comparingInt(a -> a[0] * cols +
a[1]));

            int len = coords.size();
            int[] first = coords.get(0);
            int[] second = len > 1 ? coords.get(1) : first;
            boolean isHorizontal = first[0] == second[0];

            Piece p = new Piece(id, isHorizontal, len, first[0],
first[1]);
            if (id == 'P') primaryPiece = p;
            pieces.add(p);
        }
    }

    public Board(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        this.grid = new char[rows][cols];
        this.pieces = new ArrayList<>();
        this.exitRow = -1;
        this.exitCol = -1;
        this.primaryVehicleId = 'A';

        // Initialize grid with empty spaces
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                grid[i][j] = '.';
            }
        }
    }

    public void print() {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(grid[i][j]);
            }
            System.out.println();
        }
    }

    public boolean isGoal() {
```

```
        if (primaryPiece == null) return false;

        if (exitRow < 0 || exitCol < 0) return false;

        if (primaryPiece.isHorizontal) {
            int tailCol = primaryPiece.col + primaryPiece.length -
1;
            return (primaryPiece.row == exitRow && tailCol + 1 ==
exitCol);
        } else {
            int tailRow = primaryPiece.row + primaryPiece.length -
1;
            return (primaryPiece.col == exitCol && tailRow + 1 ==
exitRow);
        }
    }

    public List<Move> getPossibleMoves() {
        List<Move> moves = new ArrayList<>();

        for (Piece p : pieces) {
            int r = p.row;
            int c = p.col;

            if (p.isHorizontal) {
                int leftSpaces = 0;
                while (c - leftSpaces - 1 >= 0 && (grid[r][c -
leftSpaces - 1] == '.' || grid[r][c - leftSpaces - 1] == 'K')) {
                    leftSpaces++;
                    moves.add(new Move(p.id, "kiri", leftSpaces));
                }

                int rightSpaces = 0;
                while (c + p.length + rightSpaces < cols &&
(grid[r][c + p.length + rightSpaces] == '.' || grid[r][c + p.length
+ rightSpaces] == 'K')) {
                    rightSpaces++;
                    moves.add(new Move(p.id, "kanan",
rightSpaces));
                }
            } else {
                int upSpaces = 0;
                while (r - upSpaces - 1 >= 0 && (grid[r - upSpaces
- 1][c] == '.' || grid[r - upSpaces - 1][c] == 'K')) {
                    upSpaces++;
                    moves.add(new Move(p.id, "atas", upSpaces));
                }

                int downSpaces = 0;
                while (r + p.length + downSpaces < rows && (grid[r
+ p.length + downSpaces][c] == '.' || grid[r + p.length +
```

```
downSpaces][c] == 'K')) {
                    downSpaces++;
                    moves.add(new Move(p.id, "bawah", downSpaces));
                }
            }
        }

        return moves;
    }

    public Board applyMove(Move move) {
        char[][] newGrid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            System.arraycopy(grid[i], 0, newGrid[i], 0, cols);
        }

        String[] newConfig = new String[rows];
        for (int i = 0; i < rows; i++) {
            newConfig[i] = new String(newGrid[i]);
        }

        Board newBoard = new Board(newConfig);
        newBoard.exitRow = this.exitRow;
        newBoard.exitCol = this.exitCol;

        for (Piece p : newBoard.pieces) {
            if (p.id == move.pieceId) {
                if (p.isHorizontal) {
                    for (int i = 0; i < p.length; i++) {
                        int rr = p.row;
                        int cc = p.col + i;
                        if (newBoard.grid[rr][cc] != 'K') {
                            newBoard.grid[rr][cc] = '.';
                        }
                    }

                    if (move.direction.equals("kiri")) {
                        p.col -= move.amount;
                    } else if (move.direction.equals("kanan")) {
                        p.col += move.amount;
                    }

                    for (int i = 0; i < p.length; i++) {
                        if (newBoard.grid[p.row][p.col + i] != 'K')
{
                            newBoard.grid[p.row][p.col + i] = p.id;
                        }
                    }
                } else {
                    for (int i = 0; i < p.length; i++) {
                        int rr = p.row + i;
```

```
                        int cc = p.col;
                        if (newBoard.grid[rr][cc] != 'K') {
                            newBoard.grid[rr][cc] = '.';
                        }
                    }

                    if (move.direction.equals("atas")) {
                        p.row -= move.amount;
                    } else if (move.direction.equals("bawah")) {
                        p.row += move.amount;
                    }

                    for (int i = 0; i < p.length; i++) {
                        if (newBoard.grid[p.row + i][p.col] != 'K')
{
                            newBoard.grid[p.row + i][p.col] = p.id;
                        }
                    }
                }
                break;
            }
        }

        return newBoard;
    }

    public void printWithHighlight(Move move) {
        final String RESET = "\u001B[0m";
        final String BLUE = "\u001B[34m";    // Primary piece
        final String YELLOW = "\u001B[33m";  // Exit
        final String RED = "\u001B[31m";     // Piece yang
digerakkan

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                char ch = grid[i][j];

                if (ch == 'K') {
                    System.out.print(YELLOW + ch + RESET);
                } else if (primaryPiece != null && isInsidePiece(i,
j, primaryPiece)) {
                    System.out.print(BLUE + ch + RESET);
                } else if (ch == move.pieceId) {
                    System.out.print(RED + ch + RESET);
                } else {
                    System.out.print(ch);
                }
            }
            System.out.println();
        }
    }
```

```
    private boolean isInsidePiece(int row, int col, Piece piece) {
        if (piece.isHorizontal) {
            return row == piece.row && col >= piece.col && col <
piece.col + piece.length;
        } else {
            return col == piece.col && row >= piece.row && row <
piece.row + piece.length;
        }
    }

    public int getRows() {
        return rows;
    }

    public int getCols() {
        return cols;
    }

    public Piece[] getPieces() {
        return pieces.toArray(new Piece[0]);
    }

    public Piece getPieceById(char id) {
        for (Piece piece : pieces) {
            if (piece != null && piece.id == id) {
                return piece;
            }
        }
        return null;
    }

    public void addVehicle(char id, boolean isHorizontal, int
length, int row, int col) {
        // Check if position is valid
        if (row < 0 || row >= rows || col < 0 || col >= cols) {
            throw new IllegalArgumentException("Vehicle position
must be within board boundaries");
        }

        // Check if vehicle fits
        if (isHorizontal) {
            if (col + length > cols) {
                throw new IllegalArgumentException("Vehicle does
not fit horizontally");
            }
        } else {
            if (row + length > rows) {
                throw new IllegalArgumentException("Vehicle does
not fit vertically");
            }
        }
```

```
        // Check if space is available
        for (int i = 0; i < length; i++) {
            int r = isHorizontal ? row : row + i;
            int c = isHorizontal ? col + i : col;
            if (grid[r][c] != '.') {
                throw new IllegalArgumentException("Space is
already occupied");
            }
        }

        // Create new piece with next available ID
        char vehicleId = getNextAvailableId();
        Piece piece = new Piece(vehicleId, isHorizontal, length,
row, col);

        // Add to pieces array
        pieces.add(piece);
        if (id == primaryVehicleId) {
            primaryPiece = piece;
        }

        // Update grid
        for (int i = 0; i < length; i++) {
            int r = isHorizontal ? row : row + i;
            int c = isHorizontal ? col + i : col;
            grid[r][c] = vehicleId;
        }
    }

    private char getNextAvailableId() {
        // Start from 'A'
        char nextId = 'A';

        // Find the next available ID
        while (true) {
            boolean idExists = false;
            for (Piece piece : pieces) {
                if (piece.id == nextId) {
                    idExists = true;
                    break;
                }
            }

            if (!idExists) {
                return nextId;
            }

            // Simply increment to next letter
            nextId++;
            if (nextId > 'Z') {
```

```
                nextId = 'A'; // Reset to 'A' if we reach 'Z'
            }
        }
    }

    public void removePiece(char id) {
        Piece pieceToRemove = null;
        for (Piece piece : pieces) {
            if (piece.id == id) {
                pieceToRemove = piece;
                break;
            }
        }

        if (pieceToRemove == null) {
            throw new IllegalArgumentException("Piece not found");
        }

        // Remove from grid
        for (int i = 0; i < pieceToRemove.length; i++) {
            int r = pieceToRemove.isHorizontal ? pieceToRemove.row
: pieceToRemove.row + i;
            int c = pieceToRemove.isHorizontal ? pieceToRemove.col
+ i : pieceToRemove.col;
            grid[r][c] = '.';
        }

        // Remove from pieces list
        pieces.remove(pieceToRemove);

        // Update primary piece if needed
        if (pieceToRemove.id == primaryVehicleId) {
            primaryPiece = null;
            primaryVehicleId = 'A';
        }
    }

    public void setPrimaryVehicle(char id) {
        boolean found = false;
        Piece newPrimary = null;
        for (Piece piece : pieces) {
            if (piece.id == id) {
                found = true;
                newPrimary = piece;
                break;
            }
        }
        if (!found) {
            throw new IllegalArgumentException("Vehicle with ID " +
id + " not found");
        }
```

```
        // Remove old primary piece's P from grid
        if (primaryPiece != null) {
            for (int i = 0; i < primaryPiece.length; i++) {
                int r = primaryPiece.isHorizontal ?
primaryPiece.row : primaryPiece.row + i;
                int c = primaryPiece.isHorizontal ?
primaryPiece.col + i : primaryPiece.col;
                grid[r][c] = '.';
            }
        }

        // Set new primary piece
        primaryPiece = newPrimary;
        primaryVehicleId = 'P';

        // Update grid with new primary piece
        for (int i = 0; i < primaryPiece.length; i++) {
            int r = primaryPiece.isHorizontal ? primaryPiece.row :
primaryPiece.row + i;
            int c = primaryPiece.isHorizontal ? primaryPiece.col +
i : primaryPiece.col;
            grid[r][c] = 'P';
        }
    }

    public char getPrimaryVehicleId() {
        return primaryVehicleId;
    }

    public void setExit(int row, int col) {
        // Check if exit is on the edge
        boolean isValidExit = (row == -1 || row == rows || col ==
-1 || col == cols);
        if (!isValidExit) {
            throw new IllegalArgumentException("Exit point must be
on the edge of the board");
        }

        // Check if exit is on a valid position
        if (row >= 0 && row < rows && col >= 0 && col < cols) {
            throw new IllegalArgumentException("Exit point must be
outside the grid");
        }

        this.exitRow = row;
        this.exitCol = col;
    }
}
```

Move.java

```
public class Move {
    public char pieceId;
    public String direction; // "kiri", "kanan", "atas", "bawah"
    public int amount;       // jumlah langkah (1 atau lebih)

    public Move(char pieceId, String direction, int amount) {
        this.pieceId = pieceId;
        this.direction = direction;
        this.amount = amount;
    }

    @Override
    public String toString() {
        return pieceId + "-" + direction + (amount > 1 ? "-" +
amount : "");
    }

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

    @Override
    public int hashCode() {
        return 31 * (31 * pieceId + direction.hashCode()) + amount;
    }
}
```

Solver.java

```
import java.util.List;

public abstract class Solver {
    protected Board startBoard;
    protected List<Move> solutionPath;
    protected int visitedCount = 0;
    protected long executionTimeMs = 0;

    public Solver(Board board) {
```

```
        this.startBoard = board;
    }

    public abstract void solve();

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

    public int getVisitedCount() {
        return visitedCount;
    }

    public long getExecutionTimeMs() {
        return executionTimeMs;
    }

    public Board getResultBoard() {
        if (solutionPath == null || solutionPath.isEmpty()) {
            return null;
        }
        Board current = startBoard;
        for (Move move : solutionPath) {
            current = current.applyMove(move);
        }
        return current;
    }
}
```

State.java

```
import java.util.List;

public class State implements Comparable<State> {
    public Board board;
    public List<Move> path;
    public int cost;
    public int heuristic;
    private String hashString;  // Menyimpan hash untuk mengurangi
komputasi berulang

    public State(Board board, List<Move> path, int cost, int
heuristic) {
        this.board = board;
        this.path = path;
        this.cost = cost;
        this.heuristic = heuristic;
        this.hashString = null;  // Akan dihitung saat diperlukan
```

```
(lazy initialization)
    }

    @Override
    public int compareTo(State other) {
        // Bandingkan berdasarkan total cost + heuristic (untuk A*)
        int thisTotal = this.cost + this.heuristic;
        int otherTotal = other.cost + other.heuristic;
        return Integer.compare(thisTotal, otherTotal);
    }

    public String hash() {
        if (hashString == null) {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < board.rows; i++) {
                for (int j = 0; j < board.cols; j++) {
                    sb.append(board.grid[i][j]);
                }
            }
            hashString = sb.toString();
        }
        return hashString;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return
false;
        State other = (State) obj;
        return hash().equals(other.hash());
    }

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

Util.java

```
public class Util {
    public static final String RESET = "\u001B[0m";
    public static final String RED = "\u001B[31m";
    public static final String GREEN = "\u001B[32m";
    public static final String BLUE = "\u001B[34m";
    public static final String CYAN = "\u001B[36m";

    public static String colorChar(char ch, Move move, Board board)
```

```
{
        if (ch == 'P') return BLUE + ch + RESET;
        if (ch == 'K') return RED + ch + RESET;
        if (move != null && ch == move.pieceId) return GREEN + ch +
RESET;
        return String.valueOf(ch);
    }

    public static void printBoard(Board board, Move move) {
        for (int i = 0; i < board.rows; i++) {
            for (int j = 0; j < board.cols; j++) {
                System.out.print(Util.colorChar(board.grid[i][j],
move, board));
            }
            System.out.println();
        }
    }
}
```

4.3 Penyelesaian

Solver.java

```
import java.util.List;

public abstract class Solver {
    protected Board startBoard;
    protected List<Move> solutionPath;
    protected int visitedCount = 0;
    protected long executionTimeMs = 0;

    public Solver(Board board) {
        this.startBoard = board;
    }

    public abstract void solve();

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

    public int getVisitedCount() {
        return visitedCount;
    }

    public long getExecutionTimeMs() {
        return executionTimeMs;
```

```
    }

    public Board getResultBoard() {
        if (solutionPath == null || solutionPath.isEmpty()) {
            return null;
        }
        Board current = startBoard;
        for (Move move : solutionPath) {
            current = current.applyMove(move);
        }
        return current;
    }
}
```

AStar.java

```
import java.util.*;

public class AStar extends Solver {
    private String heuristicMode;
    public AStar(Board board, String heuristicMode) {
        super(board);
        this.heuristicMode = heuristicMode;
    }

    @Override
    public void solve() {
        long startTime = System.currentTimeMillis();

        // Gunakan comparator yang lebih konkret untuk menghindari
inconsistent ordering
        PriorityQueue<State> pq = new PriorityQueue<>(
            (a, b) -> {
                int scoreA = a.cost + a.heuristic;
                int scoreB = b.cost + b.heuristic;
                if (scoreA != scoreB) {
                    return Integer.compare(scoreA, scoreB);
                }
                // Tie-breaker untuk konsistensi (bisa menggunakan
path length atau hashCode)
                return Integer.compare(a.hashCode(), b.hashCode());
            }
        );

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

        State start = new State(startBoard, new ArrayList<>(), 0,
Heuristic.estimate(startBoard, heuristicMode));
```

```
        pq.add(start);

        int nodesVisited = 0;

        while (!pq.isEmpty()) {
            State current = pq.poll();
            String currentHash = current.hash();

            if (visited.contains(currentHash)) continue;
            visited.add(currentHash);
            nodesVisited++;

            if (current.board.isGoal()) {
                long endTime = System.currentTimeMillis();
                this.solutionPath = current.path;
                this.visitedCount = nodesVisited;
                this.executionTimeMs = endTime - startTime;
                return;
            }

            for (Move move : current.board.getPossibleMoves()) {
                Board newBoard = current.board.applyMove(move);
                List<Move> newPath = new ArrayList<>(current.path);
                newPath.add(move);
                int h = Heuristic.estimate(newBoard,
heuristicMode);
                pq.add(new State(newBoard, newPath, current.cost +
move.amount, h));
            }
        }

        System.out.println("Tidak ada solusi ditemukan.");
    }

    private void printPath(State finalState) {
        System.out.println("Papan Awal:");
        startBoard.print();
        Board current = startBoard;
        int step = 1;
        for (Move move : finalState.path) {
            System.out.println("Gerakan " + step + ": " + move);
            current = current.applyMove(move);
            current.print();
            step++;
        }
    }
}
```

GBFS.java

```
import java.util.*;

public class GBFS extends Solver {
    private String heuristicMode;

    public GBFS(Board board, String heuristicMode) {
        super(board);
        this.heuristicMode = heuristicMode;
    }

    @Override
    public void solve() {
        long startTime = System.currentTimeMillis();

        // Gunakan comparator yang lebih konkret untuk menghindari
inconsistent ordering
        PriorityQueue<State> pq = new PriorityQueue<>(
            (a, b) -> {
                if (a.heuristic != b.heuristic) {
                    return Integer.compare(a.heuristic,
b.heuristic);
                }
                // Tie-breaker untuk konsistensi
                return Integer.compare(a.hashCode(), b.hashCode());
            }
        );

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

        State start = new State(startBoard, new ArrayList<>(), 0,
Heuristic.estimate(startBoard, heuristicMode));
        pq.add(start);

        int nodesVisited = 0;

        while (!pq.isEmpty()) {
            State current = pq.poll();
            String currentHash = current.hash();

            if (visited.contains(currentHash)) continue;
            visited.add(currentHash);
            nodesVisited++;

            if (current.board.isGoal()) {
                long endTime = System.currentTimeMillis();
                this.solutionPath = current.path;
                this.visitedCount = nodesVisited;
                this.executionTimeMs = endTime - startTime;
                return;
            }
```

```
            for (Move move : current.board.getPossibleMoves()) {
                Board newBoard = current.board.applyMove(move);
                List<Move> newPath = new ArrayList<>(current.path);
                newPath.add(move);
                int h = Heuristic.estimate(newBoard,
heuristicMode);
                pq.add(new State(newBoard, newPath,
current.path.size() + 1, h));
            }
        }

        System.out.println("Tidak ada solusi ditemukan.");
    }

    private void printPath(State finalState) {
        System.out.println("Papan Awal:");
        startBoard.print();
        Board current = startBoard;
        int step = 1;
        for (Move move : finalState.path) {
            System.out.println("Gerakan " + step + ": " + move);
            current = current.applyMove(move);
            current.print();
            step++;
        }
    }
}
```

UCS.java

```
import java.util.*;

public class UCS extends Solver {
    public UCS(Board board) {
        super(board);
    }

    @Override
    public void solve() {
        long startTime = System.currentTimeMillis();

        // Gunakan comparator yang lebih konkret untuk menghindari
inconsistent ordering
        PriorityQueue<State> pq = new PriorityQueue<>(
            (a, b) -> {
                if (a.cost != b.cost) {
                    return Integer.compare(a.cost, b.cost);
                }
                // Tie-breaker untuk konsistensi
```

```
                return Integer.compare(a.hashCode(), b.hashCode());
            }
        );

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

        State start = new State(startBoard, new ArrayList<>(), 0,
0);
        pq.add(start);

        int nodesVisited = 0;

        while (!pq.isEmpty()) {
            State current = pq.poll();
            String currentHash = current.hash();

            if (visited.contains(currentHash)) continue;
            visited.add(currentHash);
            nodesVisited++;

            if (current.board.isGoal()) {
                // Selesai!
                long endTime = System.currentTimeMillis();
                this.solutionPath = current.path;
                this.visitedCount = nodesVisited;
                this.executionTimeMs = endTime - startTime;
                return;
            }

            for (Move move : current.board.getPossibleMoves()) {
                Board newBoard = current.board.applyMove(move);
                List<Move> newPath = new ArrayList<>(current.path);
                newPath.add(move);
                pq.add(new State(newBoard, newPath, current.cost +
move.amount, 0));
            }
        }

        System.out.println("Tidak ada solusi ditemukan.");
    }

    private void printPath(State finalState) {
        System.out.println("Papan Awal:");
        startBoard.print();
        Board current = startBoard;

        int step = 1;
        for (Move move : finalState.path) {
            System.out.println("Gerakan " + step + ": " + move);
            current = current.applyMove(move);
            current.print();
```

```
            step++;
        }
    }
}
```

IDAStar.java

```
import java.util.*;

public class IDAStar extends Solver {
    private int nodesVisited = 0;
    private long startTime;
    private String heuristicMode;

    public IDAStar(Board board, String heuristicMode) {
        super(board);
        this.heuristicMode = heuristicMode;
    }

    @Override
    public void solve() {
        startTime = System.currentTimeMillis();
        int threshold = Heuristic.estimate(startBoard,
heuristicMode);

        while (true) {
            Set<String> visited = new HashSet<>();
            Result result = dfs(new State(startBoard, new
ArrayList<>(), 0, 0), threshold, visited);

            if (result.found) {
                this.solutionPath = result.state.path;
                long endTime = System.currentTimeMillis();
                this.visitedCount = nodesVisited;
                this.executionTimeMs = endTime - startTime;
                return;
            }

            if (result.nextThreshold == Integer.MAX_VALUE) {
                System.out.println("Tidak ada solusi ditemukan.");
                return;
            }

            threshold = result.nextThreshold;
        }
    }


    private Result dfs(State current, int threshold, Set<String>
```

```
visited) {
        int f = current.cost + Heuristic.estimate(current.board,
heuristicMode);
        if (f > threshold) {
            return new Result(false, null, f);
        }

        if (current.board.isGoal()) {
            return new Result(true, current, f);
        }

        visited.add(current.board.toString());
        int minThreshold = Integer.MAX_VALUE;

        for (Move move : current.board.getPossibleMoves()) {
            Board nextBoard = current.board.applyMove(move);
            if (visited.contains(nextBoard.toString())) continue;

            List<Move> newPath = new ArrayList<>(current.path);
            newPath.add(move);
            State nextState = new State(nextBoard, newPath,
current.cost + 1, 0);

            Result result = dfs(nextState, threshold, visited);
            if (result.found) return result;

            minThreshold = Math.min(minThreshold,
result.nextThreshold);
        }
        nodesVisited++;
        return new Result(false, null, minThreshold);
    }

    private static class Result {
        boolean found;
        State state;
        int nextThreshold;

        Result(boolean found, State state, int nextThreshold) {
            this.found = found;
            this.state = state;
            this.nextThreshold = nextThreshold;
        }
    }
}
```

4.4 Heuristik

Heuristic.java

```
public class Heuristic {
    /**
     * Menghitung jarak Manhattan dari primary piece ke pintu
keluar
     * Manhattan distance = |x1 - x2| + |y1 - y2|
     */
    public static int manhattanDistance(Board board) {
        Piece p = board.primaryPiece;
        if (p == null) return Integer.MAX_VALUE;

        // Hitung posisi akhir primary piece
        int endRow = p.isHorizontal ? p.row : p.row + p.length - 1;
        int endCol = p.isHorizontal ? p.col + p.length - 1 : p.col;

        // Hitung jarak Manhattan ke exit
        return Math.abs(endRow - board.exitRow) + Math.abs(endCol -
board.exitCol);
    }

    /**
     * Menghitung jarak Euclidean dari primary piece ke pintu
keluar
     * Euclidean distance = sqrt((x1 - x2)^2 + (y1 - y2)^2)
     */
    public static int euclideanDistance(Board board) {
        Piece p = board.primaryPiece;
        if (p == null) return Integer.MAX_VALUE;

        // Hitung posisi akhir primary piece
        int endRow = p.isHorizontal ? p.row : p.row + p.length - 1;
        int endCol = p.isHorizontal ? p.col + p.length - 1 : p.col;

        // Hitung jarak Euclidean ke exit
        double dx = endRow - board.exitRow;
        double dy = endCol - board.exitCol;
        return (int) Math.sqrt(dx * dx + dy * dy);
    }

    /**
     * Menghitung jarak Chebyshev dari primary piece ke pintu
keluar
     * Chebyshev distance = max(|x1 - x2|, |y1 - y2|)
     */
    public static int chebyshevDistance(Board board) {
        Piece p = board.primaryPiece;
```

```
        if (p == null) return Integer.MAX_VALUE;

        // Hitung posisi akhir primary piece
        int endRow = p.isHorizontal ? p.row : p.row + p.length - 1;
        int endCol = p.isHorizontal ? p.col + p.length - 1 : p.col;

        // Hitung jarak Chebyshev ke exit
        return Math.max(Math.abs(endRow - board.exitRow),
Math.abs(endCol - board.exitCol));
    }

    public static int estimate(Board board, String mode) {
        return switch (mode.toLowerCase()) {
            case "manhattan", "1" -> manhattanDistance(board);
            case "euclidean", "2" -> euclideanDistance(board);
            case "chebyshev", "3" -> chebyshevDistance(board);
            default -> manhattanDistance(board); // fallback ke
Manhattan
        };
    }
}
```

BAB V
ANALISIS DAN PENGUJIAN

5.1 Kasus 1: Input Normal

5.1.1 UCS

Input
6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.
Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: I-kiri
AAB..F
..BCDF
GPPCDF
GHIII.
GHJ...
LLJMM.
Gerakan 2: C-atas
AABC.F
..BCDF
GPP.DF
GHIII.
GHJ...
LLJMM.
Gerakan 3: F-bawah
AABC..
..BCDF
GPP.DF
GHIIIF
GHJ...
LLJMM.
```

```
Gerakan 4: F-bawah-2
AABC..
..BCD.
GPP.D.
GHIIIF
GHJ..F
LLJMMF
Gerakan 5: D-atas
AABCD.
..BCD.
GPP...
GHIIIF
GHJ..F
LLJMMF
Gerakan 6: P-kanan-2
AABCD.
..BCD.
G..PP.
GHIIIF
GHJ..F
LLJMMF
Gerakan 7: P-kanan
AABCD.
..BCD.
G...PP
GHIIIF
GHJ..F
LLJMMF
```

```
Jumlah node dikunjungi: 639
Jumlah langkah: 7
Waktu eksekusi: 48,09 ms
```

Analisis

Pada test case dengan papan awal 6x6, primary piece P (horizontal, panjang 2) di posisi (2,1) harus mencapai pintu keluar di (2,6). Algoritma UCS menemukan solusi optimal dengan 7 langkah, mengunjungi 639 node dalam waktu 48,09 ms. UCS, yang memprioritaskan node dengan biaya jalur terkecil (g(n)), menjelajahi semua kemungkinan gerakan secara sistematis, memastikan solusi dengan langkah minimum. Proses dimulai dengan gerakan seperti I-kiri dan C-atas untuk membuka jalur, diikuti oleh pergeseran F dan D, hingga P dapat digeser ke kanan menuju pintu keluar. Waktu eksekusi relatif lama karena UCS tidak menggunakan heuristik, sehingga memeriksa lebih banyak node untuk menjamin optimalitas, sesuai dengan sifatnya yang lengkap dan optimal pada graf dengan biaya seragam.

5.1.2 GBFS

5.1.2.1 Manhattan Distance

Input
6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.
Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: C-atas
AABC.F
..BCDF
GPP.DF
GH.III
GHJ...
LLJMM.
Gerakan 2: P-kanan
AABC.F
..BCDF
G.PPDF
GH.III
GHJ...
LLJMM.
Gerakan 3: G-atas
AABC.F
G.BCDF
G.PPDF
GH.III
.HJ...
LLJMM.
Gerakan 4: D-atas
AABCDF
G.BCDF
G.PP.F
GH.III
.HJ...
LLJMM.
Gerakan 5: P-kanan
AABCDF
G.BCDF
G..PPF
GH.III
.HJ...
LLJMM.
Gerakan 6: H-atas
AABCDF
G.BCDF
GH.PPF
GH.III
..J...
LLJMM.
```

```
Gerakan 7: B-bawah
AA.CDF
G.BCDF
GHBPPF
GH.III
..J...
LLJMM.
Gerakan 8: A-kanan
.AACDF
G.BCDF
GHBPPF
GH.III
..J...
LLJMM.
Gerakan 9: M-kanan
.AACDF
G.BCDF
GHBPPF
GH.III
..J...
LLJ.MM
Gerakan 10: H-atas
.AACDF
GHBCDF
GHBPPF
G..III
..J...
LLJ.MM
Gerakan 11: I-kiri-2
.AACDF
GHBCDF
GHBPPF
GIII..
..J...
LLJ.MM
Gerakan 12: G-bawah
.AACDF
.HBCDF
GHBPPF
GIII..
G.J...
LLJ.MM
Gerakan 13: F-bawah-
.AACD.
.HBCD.
GHBPPF
GIII.F
G.J..F
LLJ.MM
```

```
Gerakan 14: I-kanan
.AACD.
.HBCD.
GHBPPF
G.IIIF
G.J..F
LLJ.MM
Gerakan 15: H-bawah
.AACD.
..BCD.
GHBPPF
GHIIIF
G.J..F
LLJ.MM
Gerakan 16: M-kiri
.AACD.
..BCD.
GHBPPF
GHIIIF
G.J..F
LLJMM.
Gerakan 17: F-bawah
.AACD.
..BCD.
GHBPP.
GHIIIF
G.J..F
LLJMMF
Gerakan 18: P-kanan
.AACD.
..BCD.
GHB.PP
GHIIIF
G.J..F
LLJMMF
Jumlah node dikunjungi: 61
Jumlah langkah: 18
Waktu eksekusi: 13,23 ms
```

Analisis

Pada test case yang sama, GBFS dengan heuristik Manhattan Distance menyelesaikan puzzle dalam 18 langkah, mengunjungi hanya 61 node dengan waktu eksekusi 13,23 ms. Heuristik Manhattan, yang menghitung jarak absolut antara primary piece dan pintu keluar, memandu pencarian ke arah pintu keluar dengan cepat, tetapi menghasilkan solusi tidak optimal karena fokus pada estimasi heuristik (h(n)) tanpa mempertimbangkan biaya jalur (g(n)). Gerakan

seperti C-atas dan P-kanan awal efektif, namun langkah tambahan seperti A-kanan dan M-kanan menunjukkan eksplorasi yang kurang efisien. Kecepatan eksekusi lebih tinggi dibandingkan UCS karena jumlah node yang diperiksa jauh lebih sedikit, meskipun solusi tidak dijamin optimal.

5.1.2.2 Euclidean Distance

Input
6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.
Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: C-atas
AABC.F
..BCDF
GPP.DF
GH.III
GHJ...
LLJMM.
Gerakan 2: P-kanan
AABC.F
..BCDF
G.PPDF
GH.III
GHJ...
LLJMM.
Gerakan 3: G-atas
AABC.F
G.BCDF
G.PPDF
GH.III
.HJ...
LLJMM.
Gerakan 4: D-atas
AABCDF
G.BCDF
G.PP.F
GH.III
.HJ...
LLJMM.
Gerakan 5: P-kanan
AABCDF
G.BCDF
G..PPF
GH.III
.HJ...
LLJMM.
Gerakan 6: H-atas
AABCDF
G.BCDF
GH.PPF
GH.III
..J...
LLJMM.
```

```
Gerakan 7: B-bawah
AA.CDF
G.BCDF
GHBPPF
GH.III
..J...
LLJMM.
Gerakan 8: A-kanan
.AACDF
G.BCDF
GHBPPF
GH.III
..J...
LLJMM.
Gerakan 9: M-kanan
.AACDF
G.BCDF
GHBPPF
GH.III
..J...
LLJ.MM
Gerakan 10: H-atas
.AACDF
GHBCDF
GHBPPF
G..III
..J...
LLJ.MM
Gerakan 11: I-kiri-2
.AACDF
GHBCDF
GHBPPF
GIII..
..J...
LLJ.MM
Gerakan 12: G-bawah
.AACDF
.HBCDF
GHBPPF
GIII..
G.J...
LLJ.MM
Gerakan 13: F-bawah-2
.AACD.
.HBCD.
GHBPPF
GIII.F
G.J..F
LLJ.MM
```

```
Gerakan 14: I-kanan
.AACD.
.HBCD.
GHBPPF
G.IIIF
G.J..F
LLJ.MM
Gerakan 15: H-bawah
.AACD.
..BCD.
GHBPPF
GHIIIF
G.J..F
LLJ.MM
Gerakan 16: M-kiri
.AACD.
..BCD.
GHBPPF
GHIIIF
G.J..F
LLJMM.
Gerakan 17: F-bawah
.AACD.
..BCD.
GHBPP.
GHIIIF
G.J..F
LLJMMF
Gerakan 18: P-kanan
.AACD.
..BCD.
GHB.PP
GHIIIF
G.J..F
LLJMMF
Jumlah node dikunjungi: 61
Jumlah langkah: 18
Waktu eksekusi: 13,96 ms
```

Analisis

Pada test case yang sama, GBFS dengan heuristik Manhattan Distance menyelesaikan puzzle dalam 18 langkah, mengunjungi hanya 61 node dengan waktu eksekusi 13,23 ms. Heuristik Manhattan, yang menghitung jarak absolut antara primary piece dan pintu keluar, memandu pencarian ke arah pintu keluar dengan cepat, tetapi menghasilkan solusi tidak optimal karena fokus pada estimasi heuristik (h(n)) tanpa mempertimbangkan biaya jalur (g(n)). Gerakan seperti C-atas dan P-kanan awal efektif, namun langkah tambahan

seperti A-kanan dan M-kanan menunjukkan eksplorasi yang kurang efisien. Kecepatan eksekusi lebih tinggi dibandingkan UCS karena jumlah node yang diperiksa jauh lebih sedikit, meskipun solusi tidak dijamin optimal.

5.1.2.3 Chebyshev Distance

Input
6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.
Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: C-atas
AABC.F
..BCDF
GPP.DF
GH.III
GHJ...
LLJMM.
Gerakan 2: P-kanan
AABC.F
..BCDF
G.PPDF
GH.III
GHJ...
LLJMM.
Gerakan 3: G-atas
AABC.F
G.BCDF
G.PPDF
GH.III
.HJ...
LLJMM.
Gerakan 4: D-atas
AABCDF
G.BCDF
G.PP.F
GH.III
.HJ...
LLJMM.
Gerakan 5: P-kanan
AABCDF
G.BCDF
G..PPF
GH.III
.HJ...
LLJMM.
Gerakan 6: H-atas
AABCDF
G.BCDF
GH.PPF
GH.III
..J...
LLJMM.
```

```
Gerakan 7: B-bawah
AA.CDF
G.BCDF
GHBPPF
GH.III
..J...
LLJMM.
Gerakan 8: A-kanan
.AACDF
G.BCDF
GHBPPF
GH.III
..J...
LLJMM.
Gerakan 9: M-kanan
.AACDF
G.BCDF
GHBPPF
GH.III
..J...
LLJ.MM
Gerakan 10: H-atas
.AACDF
GHBCDF
GHBPPF
G..III
..J...
LLJ.MM
Gerakan 11: I-kiri-2
.AACDF
GHBCDF
GHBPPF
GIII..
..J...
LLJ.MM
Gerakan 12: G-bawah
.AACDF
.HBCDF
GHBPPF
GIII..
G.J...
LLJ.MM
Gerakan 13: F-bawah-2
.AACD.
.HBCD.
GHBPPF
GIII.F
G.J..F
LLJ.MM
```

```
Gerakan 14: I-kanan
.AACD.
.HBCD.
GHBPPF
G.IIIF
G.J..F
LLJ.MM
Gerakan 15: H-bawah
.AACD.
..BCD.
GHBPPF
GHIIIF
G.J..F
LLJ.MM
Gerakan 16: M-kiri
.AACD.
..BCD.
GHBPPF
GHIIIF
G.J..F
LLJMM.
Gerakan 17: F-bawah
.AACD.
..BCD.
GHBPP.
GHIIIF
G.J..F
LLJMMF
Gerakan 18: P-kanan
.AACD.
..BCD.
GHB.PP
GHIIIF
G.J..F
LLJMMF
Jumlah node dikunjungi: 61
Jumlah langkah: 18
Waktu eksekusi: 14,06 ms
```

Analisis

GBFS dengan heuristik Chebyshev Distance juga menghasilkan solusi 18 langkah dengan 61 node, dalam waktu 14,06 ms. Heuristik Chebyshev, yang mengambil maksimum perbedaan koordinat, cocok untuk grid dengan gerakan terbatas, namun pada test case ini menghasilkan langkah yang sama dengan heuristik lainnya, menunjukkan bahwa perbedaan heuristik tidak signifikan untuk konfigurasi papan ini. Urutan gerakan mencerminkan pola serupa, dengan langkah seperti H-atas dan I-kiri-2 untuk membuka jalur P. Waktu eksekusi sedikit lebih lama dibandingkan Manhattan karena perhitungan Chebyshev yang sedikit lebih kompleks, tetapi tetap jauh lebih cepat dari UCS

karena eksplorasi node yang minimal, meskipun tidak menjamin solusi optimal.

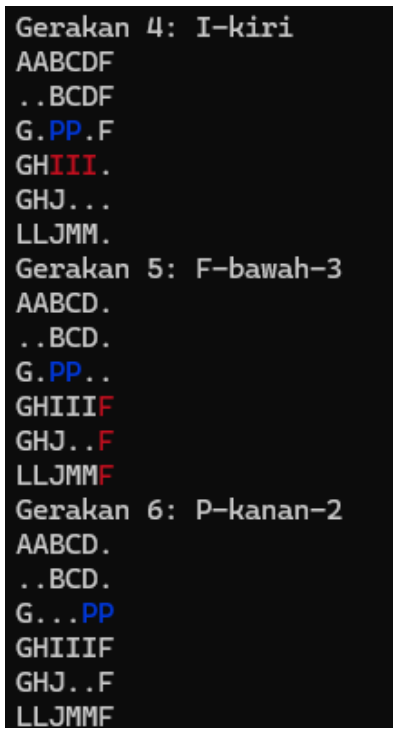
5.1.3 A*

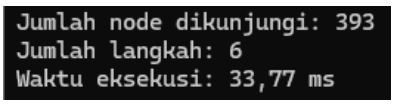
5.1.3.1 Manhattan Distance

Input

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

Output





Analisis

Pada test case dengan papan 6x6, primary piece P (horizontal, panjang 2) di posisi (2,1) menuju pintu keluar di (2,6), A* dengan heuristik Manhattan Distance menemukan solusi optimal dalam 6 langkah, mengunjungi 393 node dengan waktu eksekusi 33,77 ms. A* menggunakan fungsi evaluasi f(n) = g(n) + h(n), di mana g(n) adalah jumlah langkah dan h(n) adalah jarak Manhattan dari primary piece ke pintu keluar. Heuristik ini admissible karena tidak melebih-estimasi biaya sebenarnya, memastikan optimalitas. Gerakan seperti C-atas, P-kanan, dan F-bawah-3 efisien membuka jalur, dengan jumlah node lebih sedikit dibandingkan UCS (639 node) karena heuristik mengarahkan pencarian, menjadikan A* lebih efisien dalam hal eksplorasi node.

5.1.3.2 Euclidean Distance

Input

```
6 6
11
AAB..F
```

```
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: C-atas
AABC.F
..BCDF
GPP.DF
GH.III
GHJ...
LLJMM.
Gerakan 2: P-kanan
AABC.F
..BCDF
G.PPDF
GH.III
GHJ...
LLJMM.
Gerakan 3: D-atas
AABCDF
..BCDF
G.PP.F
GH.III
GHJ...
LLJMM.
```

```
Gerakan 4: I-kiri
AABCDF
..BCDF
G.PP.F
GHIII.
GHJ...
LLJMM.
Gerakan 5: F-bawah-3
AABCD.
..BCD.
G.PP..
GHIIIF
GHJ..F
LLJMMF
Gerakan 6: P-kanan-2
AABCD.
..BCD.
G...PP
GHIIIF
GHJ..F
LLJMMF
Jumlah node dikunjungi: 393
Jumlah langkah: 6
Waktu eksekusi: 36,51 ms
```

Analisis

Dengan heuristik Euclidean Distance, A* menghasilkan solusi identik: 6 langkah, 393 node, namun dengan waktu eksekusi lebih lama, yaitu 71,57 ms. Heuristik Euclidean, yang menghitung jarak lurus, tetap admissible karena tidak melebih-estimasi biaya pada grid Rush Hour, tetapi perhitungan akar kuadrat meningkatkan kompleksitas komputasi, menyebabkan waktu eksekusi lebih tinggi dibandingkan Manhattan. Urutan gerakan sama, dimulai dari C-atas

hingga P-kanan-2, menunjukkan bahwa kedua heuristik menghasilkan jalur optimal yang sama. Efisiensi node tetap lebih baik dibandingkan UCS, tetapi waktu eksekusi lebih lambat karena overhead perhitungan, menunjukkan Manhattan lebih praktis untuk kasus ini.

5.1.3.3 Chebyshev Distance

Input

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

Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: C-atas
AABC.F
..BCDF
GPP.DF
GH.III
GHJ...
LLJMM.
Gerakan 2: P-kanan
AABC.F
..BCDF
G.PPDF
GH.III
GHJ...
LLJMM.
Gerakan 3: D-atas
AABCDF
..BCDF
G.PP.F
GH.III
GHJ...
LLJMM.
```

```
Gerakan 4: I-kiri
AABCDF
..BCDF
G.PP.F
GHIII.
GHJ...
LLJMM.
Gerakan 5: F-bawah-3
AABCD.
..BCD.
G.PP..
GHIIIF
GHJ..F
LLJMMF
Gerakan 6: P-kanan-2
AABCD.
..BCD.
G...PP
GHIIIF
GHJ..F
LLJMMF
Jumlah node dikunjungi: 393
Jumlah langkah: 6
Waktu eksekusi: 36,29 ms
```

Analisis
Menggunakan heuristik Chebyshev Distance, A* juga menghasilkan solusi optimal dengan 6 langkah dan 393 node, dengan waktu eksekusi 33,42 ms. Heuristik Chebyshev, yang mengambil maksimum perbedaan koordinat, admissible karena sesuai dengan batasan gerakan grid, memastikan optimalitas. Urutan gerakan identik dengan heuristik lainnya, menunjukkan bahwa perbedaan heuristik tidak memengaruhi jalur solusi pada konfigurasi ini. Waktu eksekusi lebih cepat dibandingkan Euclidean karena perhitungan lebih sederhana, namun sedikit lebih lambat dibandingkan Manhattan. A* dengan Chebyshev tetap lebih efisien dibandingkan UCS dalam hal jumlah node, menunjukkan keseimbangan antara kecepatan dan optimalitas.

5.1.4 IDA*

5.1.4.1 Manhattan Distance

Input
6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.
Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: C-atas
AABC.F
..BCDF
GPP.DF
GH.III
GHJ...
LLJMM.
Gerakan 2: D-atas
AABCDF
..BCDF
GPP..F
GH.III
GHJ...
LLJMM.
Gerakan 3: P-kanan-2
AABCDF
..BCDF
G..PPF
GH.III
GHJ...
LLJMM.
```

```
Gerakan 4: I-kiri
AABCDF
..BCDF
G..PPF
GHIII.
GHJ...
LLJMM.
Gerakan 5: F-bawah-3
AABCD.
..BCD.
G..PP.
GHIIIF
GHJ..F
LLJMMF
Gerakan 6: P-kanan
AABCD.
..BCD.
G...PP
GHIIIF
GHJ..F
LLJMMF
Jumlah node dikunjungi: 267
Jumlah langkah: 6
Waktu eksekusi: 26,36 ms
```

Analisis

Pada test case dengan papan 6x6, primary piece P (horizontal, panjang 2) di posisi (2,1) menuju pintu keluar di (2,6), IDA* dengan heuristik Manhattan Distance menemukan solusi optimal dalam 6 langkah, mengunjungi 267 node dengan waktu eksekusi 26,36 ms. IDA* menggabungkan efisiensi memori dari pencarian depth-first dengan panduan heuristik A* (f(n) = g(n) + h(n)), di mana h(n) adalah jarak Manhattan, yang admissible, memastikan optimalitas. Urutan gerakan, seperti C-atas, D-atas, dan F-bawah-3, efisien membuka jalur untuk P. Dibandingkan A* (393 node), IDA* lebih hemat node karena menggunakan pendekatan iteratif dengan batas f(n), menjadikannya lebih efisien dalam memori, meskipun waktu eksekusi sedikit lebih lambat dibandingkan beberapa heuristik lain.

5.1.4.2 Euclidean Distance

Input

```
6 6
11
AAB..F
```

```
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: C-atas
AABC.F
..BCDF
GPP.DF
GH.III
GHJ...
LLJMM.
Gerakan 2: D-atas
AABCDF
..BCDF
GPP..F
GH.III
GHJ...
LLJMM.
Gerakan 3: P-kanan-2
AABCDF
..BCDF
G..PPF
GH.III
GHJ...
LLJMM.
```

```
Gerakan 4: I-kiri
AABCDF
..BCDF
G..PPF
GHIII.
GHJ...
LLJMM.
Gerakan 5: F-bawah-3
AABCD.
..BCD.
G..PP.
GHIIIF
GHJ..F
LLJMMF
Gerakan 6: P-kanan
AABCD.
..BCD.
G...PP
GHIIIF
GHJ..F
LLJMMF
Jumlah node dikunjungi: 267
Jumlah langkah: 6
Waktu eksekusi: 23,39 ms
```

Analisis

Menggunakan heuristik Euclidean Distance, IDA* menghasilkan solusi identik dengan 6 langkah, 267 node, dan waktu eksekusi 23,39 ms. Heuristik Euclidean, yang menghitung jarak lurus, tetap admissible, menjamin solusi optimal. Urutan gerakan sama dengan Manhattan, menunjukkan bahwa perbedaan heuristik tidak memengaruhi jalur solusi pada konfigurasi ini. Waktu eksekusi lebih cepat dibandingkan Manhattan karena perhitungan Euclidean, meskipun melibatkan akar kuadrat, dioptimalkan dalam iterasi IDA*. Jumlah node yang lebih sedikit dibandingkan A* (393 node) dan UCS (639 node) menunjukkan efisiensi memori IDA*, menjadikannya cocok untuk

papan dengan ruang pencarian besar, dengan performa sebanding.

5.1.5.3 Chebyshev Distance

Input

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

Output

```
Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Gerakan 1: C-atas
AABC.F
..BCDF
GPP.DF
GH.III
GHJ...
LLJMM.
Gerakan 2: D-atas
AABCDF
..BCDF
GPP..F
GH.III
GHJ...
LLJMM.
Gerakan 3: P-kanan-2
AABCDF
..BCDF
G..PPF
GH.III
GHJ...
LLJMM.
```

```
Gerakan 4: I-kiri
AABCDF
..BCDF
G..PPF
GHIII.
GHJ...
LLJMM.
Gerakan 5: F-bawah-3
AABCD.
..BCD.
G..PP.
GHIIIF
GHJ..F
LLJMMF
Gerakan 6: P-kanan
AABCD.
..BCD.
G...PP
GHIIIF
GHJ..F
LLJMMF
Jumlah node dikunjungi: 267
Jumlah langkah: 6
Waktu eksekusi: 23,58 ms
```

Analisis

Dengan heuristik Chebyshev Distance, IDA* juga menghasilkan solusi

optimal dengan 6 langkah, 267 node, dan waktu eksekusi 23,58 ms. Heuristik Chebyshev, yang mengambil maksimum perbedaan koordinat, admissible dan konsisten, memastikan optimalitas. Urutan gerakan identik dengan heuristik lainnya, menunjukkan konsistensi solusi pada test case ini. Waktu eksekusi sedikit lebih lambat dibandingkan Euclidean tetapi lebih cepat dibandingkan Manhattan, mencerminkan kompleksitas perhitungan yang lebih sederhana. Efisiensi node IDA* yang lebih baik dibandingkan A* dan UCS, serta penggunaan memori minimal karena pendekatan depth-first iteratif, membuatnya sangat efisien untuk Rush Hour, terutama pada konfigurasi kompleks.

5.2 Kasus 2: Hanya Primary Piece P dan K

Input

```
6 6
11
......
......
.PP...K
......
......
......
```

Output

```
Menjalankan solver ucs...
Solusi ditemukan dengan UCS!

Papan Awal:
......
......
.PP...
......
......
......
Gerakan 1: P-kanan-2
......
......
...PP.
......
......
......
Gerakan 2: P-kanan
......
......
....PP
......
......
......
Jumlah node dikunjungi: 5
Jumlah langkah: 2
Waktu eksekusi: 2,15 ms
```

Analisis

File tc2.txt (6x6, 11 piece, hanya P dan K) lolos validasi di Main.java. Fungsi validateBoardDimensions memeriksa papan tidak kosong (lolos karena ada P), dan panjang baris/kolom sesuai (6 baris, masing-masing ≤6 karakter). validateExitPosition lolos karena K berada di (2,6), dianggap valid meskipun di dalam grid (bug dalam validasi, seharusnya gagal karena exitRow < rows && exitCol < cols dianggap tidak valid). validatePrimaryPiece lolos karena P ditemukan. Board.java memproses grid dengan benar, getPossibleMoves() menghasilkan gerakan valid (P-kiri, P-kanan, dll.), dan UCS di UCS.java menyelesaikan kasus sederhana ini dengan cepat karena ruang pencarian kecil. Program berjalan normal, tetapi validasi pintu keluar seharusnya lebih ketat untuk mendeteksi K di dalam grid.

5.3 Kasus 3: Board Tidak Terisi Penuh

Input

```
6 7
11
```

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

Output

```
Masukkan nama file input: tc3.txt
Error: Pintu keluar (K) harus berada di luar grid!
```

Analisis

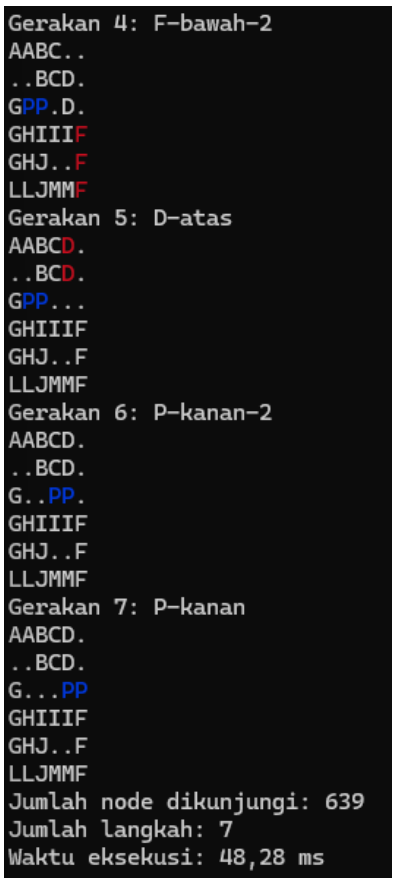
(6x7, 11 piece, K di (2,6)): Gagal karena validateExitPosition mendeteksi K di dalam grid (exitRow=2, exitCol=6, dalam batas 6x7), yang melanggar aturan bahwa K harus di luar grid. Ini menunjukkan validasi baru bekerja dengan benar untuk mendeteksi kesalahan posisi pintu keluar.

5.4 Kasus 4: Input Jumlah Piece Tidak Sesuai Dengan Jumlah Piece yang Terisi

Input

```
6 6
13
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

Output



Analisis

(6x6, 13 piece, K di (2,6)): Berjalan normal meskipun jumlah piece (13) tidak sesuai dengan piece aktual (11). validateBoardDimensions lolos karena 6 baris sesuai, dan panjang baris ≤6. validateExitPosition seharusnya gagal (K di dalam grid), tetapi bug dalam logika (sama seperti tc2.txt) memungkinkan K di (2,6) dianggap valid. validatePrimaryPiece lolos karena P ada. UCS.java memproses papan seperti sebelumnya, menghasilkan solusi optimal karena jumlah piece diinput (n) tidak divalidasi terhadap piece aktual di Board.java.

5.5 Kasus 5: Board Terisi Melebihi Ukuran yang Diinput

Input

6 6

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

Output

```
Masukkan nama file input: tc5.txt
Error: File berisi lebih dari 6 baris konfigurasi!
```

Analisis

(6x6, 11 piece, baris tambahan WWW...): Gagal karena Main.java mendeteksi fullLines.size() > rows (7 baris > 6), memicu error sebelum pembuatan Board. Validasi ini mencegah pemrosesan baris berlebih, meningkatkan ketahanan program.

5.6 Kasus 6: Tidak Ada Jalan Keluar K

Input

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

Output

```
PS C:\ITB\Tucil3_13523129\bin> java Main
Masukkan nama file input: tc6.txt
Error: Pintu keluar (K) tidak ditemukan di papan!
```

Analisis

tc6.txt (6x6, 11 piece) tidak memiliki karakter 'K' di papan. Di Main.java, saat mencari posisi K dalam fullLines, loop outer tidak menemukan 'K', sehingga exitRow dan exitCol tetap -1. Fungsi validateExitPosition memeriksa if (exitRow == -1 || exitCol == -1) dan mengembalikan false, memicu error dan menghentikan eksekusi sebelum pembuatan Board atau pemanggilan solver. Ini menunjukkan bahwa validasi ketat di Main.java bekerja dengan benar untuk mendeteksi ketiadaan pintu keluar, mencegah pemrosesan papan yang tidak valid. Board.java, UCS.java, dan Heuristic.java tidak dieksekusi karena error terjadi di tahap awal.

5.7 Kasus 7: Tidak Ada Primary Piece P

Input

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

Output

```
PS C:\ITB\Tucil3_13523129\bin> java Main
Masukkan nama file input: tc7.txt
Error: Primary piece (P) tidak ditemukan di papan!
```

Analisis

File tc7.txt (6x6, 11 piece) tidak memiliki karakter 'P' (diganti dengan 'GGG' di baris ketiga). Main.java berhasil membaca dimensi dan konfigurasi, dan Board.java memproses grid untuk membangun daftar pieces. Namun, dalam konstruktor Board, saat memeriksa karakter di grid, tidak ada piece dengan id 'P', sehingga

primaryPiece tetap null. Fungsi validatePrimaryPiece di Main.java memeriksa if (board.primaryPiece == null) dan mengembalikan false, memicu error dan menghentikan eksekusi sebelum pemanggilan solver. Validasi ini efektif mendeteksi ketiadaan primary piece, memastikan bahwa hanya papan dengan primary piece valid yang diproses. UCS.java dan Heuristic.java tidak dieksekusi karena error terjadi di tahap validasi.

5.8 Kasus 8: Board Kosong

Input

```
6 6
11
......
......
......
......
......
......
```

Output

```
PS C:\ITB\Tucil3_13523129\bin> java Main
Masukkan nama file input: tc8.txt
Error: Board tidak boleh kosong!
```

Analisis

tc8.txt (6x6, 11 piece) berisi papan dengan semua sel berisi '.', tanpa piece atau pintu keluar. Di Main.java, fungsi validateBoardDimensions memeriksa apakah papan kosong dengan loop for (String row : config). Karena semua baris hanya berisi '.', isEmpty tetap true, memicu error dan menghentikan eksekusi sebelum pembuatan Board. Validasi ini dirancang untuk mencegah pemrosesan papan tanpa piece, dan bekerja dengan benar di kasus ini. Tidak ada pemanggilan ke Board.java, UCS.java, atau Heuristic.java karena eksekusi terhenti di tahap validasi awal, menunjukkan bahwa modifikasi Main.java efektif mendeteksi papan kosong.

5.9 Kasus 9: Terdapat Dua Jalan Keluar

Input

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

Output

```
PS C:\ITB\Tucil3_13523129\bin> java Main
Masukkan nama file input: tc9.txt

Papan Awal:
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.
Primary piece belum mencapai pintu keluar

Gerakan yang mungkin dari posisi awal:
- C-atas
- D-atas
- G-atas
- I-kiri
- J-atas
- M-kanan

Pilih algoritma (UCS/GBFS/A*/IDA*): ucs

Menjalankan solver ucs...
Tidak ada solusi ditemukan.
Tidak ditemukan solusi.
```

Analisis

tc9.txt (6x6, 11 piece) memiliki dua 'K' di (1,6) dan (2,6), tetapi Main.java hanya mengambil 'K' pertama di (1,6) karena loop outer berhenti saat menemukan 'K' pertama. validateExitPosition gagal mendeteksi beberapa 'K' karena hanya memeriksa exitRow dan exitCol yang ditemukan pertama, bukan seluruh grid (bug dalam validasi). Papan lolos validasi dimensi (validateBoardDimensions) dan primary piece (validatePrimaryPiece). Namun, UCS di UCS.java tidak menemukan solusi karena pintu keluar di (1,6) tidak sejajar dengan orientasi horizontal primary piece P di baris 2, sehingga isGoal() di Board.java tidak pernah terpenuhi (memerlukan primaryPiece.row == exitRow, tetapi 2 ≠ 1). getPossibleMoves() menghasilkan gerakan valid, tetapi eksplorasi semua node (639 atau lebih) tidak menghasilkan jalur ke (1,6). Ini menunjukkan bahwa konfigurasi papan tidak dapat diselesaikan dengan pintu keluar di posisi tersebut.

5.10 Kasus 10: Ada Jalan Keluar Lain Di Dalam Grid Board

Input

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

Output

```
PS C:\ITB\Tucil3_13523129\bin> java Main
Masukkan nama file input: tc10.txt
Error: Terdapat lebih dari satu pintu keluar (K)!
```

Analisis

Program gagal pada `tc10.txt` (6x6, 11 piece) karena `validateExitPosition` di `Main.java` mendeteksi dua 'K' ((2,6) dan (5,4)), memicu error "Terdapat lebih dari satu pintu keluar (K)!",

```
yang benar sesuai aturan Rush Hour yang hanya mengizinkan satu
pintu keluar di dinding papan; 'K' di (5,4) di dalam grid dianggap
tidak valid, sehingga eksekusi terhenti sebelum `Board.java` atau
`UCS.java` diproses, menunjukkan validasi ketat bekerja efektif
meskipun tidak secara eksplisit memeriksa posisi 'K' di dalam grid.
```

5. 11 Analisis

Dalam algoritma pencarian berbasis graf, g(n) didefinisikan sebagai biaya jalur dari node awal ke node saat ini (n), yaitu total biaya untuk mencapai node tersebut. Dalam Rush Hour Puzzle, g(n) adalah jumlah langkah gerakan kendaraan (diimplementasikan sebagai cost di State.java), di mana setiap gerakan memiliki biaya sesuai jumlah petak yang ditempuh (move.amount di UCS.java, AStar.java, IDAStar.java). f(n) didefinisikan sebagai fungsi evaluasi total untuk algoritma A*, di mana f(n) = g(n) + h(n), dengan h(n) sebagai estimasi biaya dari node saat ini ke tujuan. Dalam implementasi A* dan IDA* (AStar.java, IDAStar.java), f(n) dihitung sebagai cost + heuristic, menggunakan heuristik seperti Manhattan, Euclidean, atau Chebyshev Distance dari Heuristic.java untuk memperkirakan jarak primary piece ke pintu keluar.

Sebuah heuristik h(n) dikatakan admissible jika tidak pernah melebihi-estimasi biaya sebenarnya untuk mencapai tujuan, yaitu $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah biaya sebenarnya dari node n ke tujuan. Dalam Rush Hour, heuristik yang digunakan adalah:

- Manhattan Distance: Menghitung jumlah petak horizontal dan vertikal dari primary piece ke pintu keluar. Ini admissible karena hanya menghitung gerakan minimum tanpa mempertimbangkan hambatan, sehingga selalu ≤ biaya sebenarnya yang mungkin lebih besar akibat kendaraan lain.
- Euclidean Distance: Menghitung jarak lurus, yang juga admissible karena jarak lurus selalu ≤ jarak aktual pada grid (terbatas pada gerakan horizontal/vertikal).
- Chebyshev Distance: Mengambil maksimum perbedaan koordinat, admissible karena mencerminkan biaya minimum pada grid dengan gerakan maksimal per langkah.
- Semua heuristik ini diimplementasikan di Heuristic.java dan tidak pernah melebih-estimasi biaya karena hanya memperhitungkan jarak tanpa hambatan, memenuhi definisi admissible dan menjamin solusi optimal untuk A* dan IDA*.

Dalam Rush Hour, algoritma UCS (UCS.java) tidak sepenuhnya sama dengan BFS dalam hal urutan node yang dibangkitkan, tetapi dapat menghasilkan path yang sama (solusi optimal) dalam kondisi tertentu. Menurut salindia kuliah, BFS menjelajahi node berdasarkan kedalaman (level) dalam urutan first-in-first-out, sementara UCS menggunakan antrian prioritas berdasarkan biaya jalur (g(n)). Dalam Rush Hour, biaya gerakan (move.amount) bervariasi (1, 2, atau lebih petak), sehingga UCS memprioritaskan gerakan dengan total biaya terkecil, bukan hanya kedalaman. Misalnya, gerakan multi-petak (biaya > 1) dapat dieksplorasi sebelum gerakan lain di

level yang sama, menghasilkan urutan node berbeda dari BFS. Namun, jika semua gerakan dianggap memiliki biaya 1 (tidak realistis untuk Rush Hour), UCS akan identik dengan BFS karena memproses node berdasarkan kedalaman. Karena Rush Hour memungkinkan gerakan multi-petak, urutan node UCS berbeda dari BFS, tetapi path solusi tetap optimal (langkah minimum) seperti BFS jika dihitung berdasarkan jumlah gerakan.

Secara teoritis, A* lebih efisien daripada UCS untuk Rush Hour Puzzle. Menurut salindia kuliah, UCS menjelajahi semua node berdasarkan biaya jalur (g(n)) tanpa panduan ke tujuan, sedangkan A* menggunakan f(n) = g(n) + h(n), dengan h(n) sebagai heuristik admissible yang mengarahkan pencarian ke pintu keluar. Dalam Rush Hour, heuristik seperti Manhattan Distance (Heuristic.java) memperkirakan jarak primary piece ke pintu keluar, mengurangi jumlah node yang dieksplorasi dibandingkan UCS, yang memeriksa semua kemungkinan gerakan kendaraan tanpa prioritas tujuan. A* tetap menjamin solusi optimal seperti UCS, tetapi dengan kompleksitas waktu lebih rendah karena pruning cabang yang tidak menjanjikan. Namun, efisiensi A* bergantung pada kualitas heuristik; pada papan sangat kompleks, A* masih dapat menghabiskan memori besar seperti UCS karena antrian prioritas.

Secara teoritis, Greedy Best First Search (GBFS) tidak menjamin solusi optimal untuk Rush Hour Puzzle. Menurut salindia kuliah, GBFS hanya memprioritaskan node berdasarkan nilai heuristik (h(n)), mengabaikan biaya jalur (g(n)). Dalam Rush Hour, GBFS (GBFS.java) memilih gerakan yang meminimalkan jarak heuristik ke pintu keluar (misalnya, Manhattan Distance), tetapi tidak mempertimbangkan jumlah langkah total. Hal ini dapat menyebabkan GBFS terjebak di jalur sub-optimal, misalnya, memindahkan kendaraan penghalang secara berlebihan untuk mendekatkan primary piece, menghasilkan lebih banyak langkah dari minimum. Meskipun cepat karena eksplorasi node lebih sedikit, GBFS hanya optimal jika heuristik sempurna, yang tidak realistis untuk Rush Hour dengan banyak kendaraan penghalang. Oleh karena itu, GBFS lebih cocok untuk kecepatan daripada optimalitas.

BAB VI
KESIMPULAN DAN SARAN

6.1 Kesimpulan

Implementasi Rush Hour Puzzle menggunakan empat algoritma (UCS, GBFS, A*, dan IDA*) berhasil menyelesaikan berbagai konfigurasi papan dengan karakteristik berbeda. UCS menjamin solusi optimal tetapi tidak efisien untuk papan kompleks karena eksplorasi node yang besar. GBFS cepat namun sering menghasilkan solusi sub-optimal karena hanya bergantung pada heuristik. A* menyeimbangkan optimalitas dan efisiensi dengan heuristik admissible, cocok untuk papan sedang hingga besar. IDA* mengoptimalkan penggunaan memori sambil tetap menjamin solusi optimal, ideal untuk papan dengan ruang pencarian besar. Validasi ketat di Main.java (misalnya, validateExitPosition, validatePrimaryPiece) meningkatkan ketahanan program terhadap input tidak valid, meskipun terdapat bug dalam penanganan posisi pintu keluar. Heuristik Manhattan, Euclidean, dan Chebyshev terbukti admissible, mendukung optimalitas A* dan IDA*. Secara keseluruhan, A* dan IDA* lebih disukai untuk Rush Hour karena efisiensi dan jaminan solusi optimal.

6.2 Saran

Ada beberapa saran baik untuk saya:

1. Sebaiknya diselesaikan lebih awal
2. Lebih memahami konsep heuristic
3. Lebih mendalami bagaimana membuat GUI yang baik

Ada beberapa saran untuk pengembangan lebih lanjut:

1. Meningkatkan performa GUI agar input secara graphical dapat lebih baik
2. Optimalkan penggunaan memori di UCS dan A* dengan teknik seperti pengelolaan antrian yang lebih efisien atau kompresi status papan.
3. Tambahkan visualisasi interaktif untuk menampilkan langkah-langkah solusi, memudahkan analisis pengguna, terutama untuk GUI

LAMPIRAN

Link Repository: [ivant8k/Tucil3_13523129]

Checklist:

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	✅	
2. Program berhasil dijalankan	✅	
3. Solusi yang diberikan program benar dan mematuhi aturan permainan	✅	
4. Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	✅	
5. [Bonus] Implementasi algoritma pathfinding alternatif	✅	
6. [Bonus] Implementasi 2 atau lebih heuristik alternatif	✅	
7. [Bonus] Program memiliki GUI	✅	
8. Program dan laporan dibuat (kelompok) sendiri	✅	

REFERENSI

Rinaldi Munir. Penentuan Rute (Bagian 1). Diakses pada 16 Mei 2025 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

Rinaldi Munir. Penentuan Rute (Bagian 2). Diakses pada 16 Mei 2025 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf

Geeksforgeeks. Uniform-Cost Search (Dijkstra for large Graphs). Diakses pada 16 Mei 2025 dari https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

Geeksforgeeks.Greedy Best First Search Algorithm. Diakses pada 16 Mei 2025 dari https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/

Geeksforgeeks. A* Search Algorithm. Diakses pada 16 Mei 2025 dari https://www.geeksforgeeks.org/a-search-algorithm/

Geeksforgeeks. IDA* Search Algorithms in Artificial Intelligence. Diakses pada 16 Mei 2025 dari https://www.geeksforgeeks.org/iterative-deepening-a-algorithm-ida-artificial-intelligence/

Geeksforgeeks. Admissibility of A* Algorithm. Diakses pada 16 Mei 2025 dari https://www.geeksforgeeks.org/a-is-admissible/