

IF2211 Strategi Algoritma

Penyelesaian Rush Hour Puzzle menggunakan Algoritma Pathfinding

Laporan Tugas Kecil 3

Disusun sebagai pemenuhan tugas kecil 3 IF2211 Strategi Algoritma
Tahun Akademik 2024/2025 Semester 2



Disusun oleh:

Jovandra Otniel P. S. (13523141)

Daftar Isi

BAB I: DESKRIPSI MASALAH.....	2
BAB II: LANDASAN TEORI.....	4
2.1 Representasi State dan Graph-Search.....	4
2.2 Uniform-Cost Search (UCS).....	4
2.3 Greedy Best-First Search (GBFS).....	5
2.4 A* Search.....	6
BAB III: SPESIFIKASI TUGAS.....	7
BAB IV: IMPLEMENTASI ALGORITMA.....	10
4.1 Struktur Kode dan Class (Java).....	10
4.2 Input Parser dan Representasi Board (Papan).....	10
4.3 Perluasan Child State (Status Anak).....	11
BAB V: KODE SUMBER.....	11
BAB VI: PENGUJIAN ALGORITMA.....	34
BAB VII: ANALISIS ALGORITMA.....	39
7.1 Definisi $f(n)$ dan $g(n)$	39
7.2 Admissibility Heuristik A*.....	39
7.3 Apakah UCS sama dengan BFS di Rush Hour?.....	39
7.4 Efisiensi A* dibandingkan UCS.....	40
7.5 Optimalitas Greedy Best-First Search.....	40
7.6 Kompleksitas Waktu dan Memori.....	40
7.7 Pembahasan Hasil Uji.....	41
BAB VIII: Kesimpulan dan Saran.....	42
Lampiran.....	43

BAB I: DESKRIPSI MASALAH

Permainan Rush Hour adalah permainan papan (*boardgame*) baik melalui media fisik maupun media digital (*online*). Permainan ini sekilas tampak sederhana, biasanya terdiri papan 6×6 dan sejumlah kendaraan yang bisa digeser secara horizontal atau vertikal. Namun, di dalam kesederhanaan aturan tersebut, tersimpan tantangan besar: memindahkan kendaraan sehingga kendaraan primer (label 'P') bisa keluar melalui celah di tepi papan. Meski tak ada rotasi atau tumpang tindih, kombinasi posisi kendaraan bisa mencapai puluhan ribu variasi, sehingga solusi yang efisien memerlukan strategi pencarian yang tepat.

Secara teknis, setiap susunan kendaraan dianggap sebagai simpul dalam graf, dengan setiap gerakan menggeser satu kendaraan sejauh beberapa sel mewakili tepi berbiaya seragam (biaya = 1 per sel). Tugas utamanya adalah menemukan jalur terpendek dari keadaan awal ke keadaan tujuan, di mana 'P' sudah melewati pintu keluar ('K'). Menambahkan rintangan statis ('X') dan variasi letak pintu ('K') di empat sisi: atas, bawah, kiri, atau kanan sehingga semakin memperluas ruang status dan menambah kompleksitas.

Tiga algoritma pencarian graf menjadi andalan:

- *Uniform-Cost Search (UCS)*, yang menjamin solusi optimal dengan menjelajahi simpul berdasarkan biaya kumulatif terendah;
- *Greedy Best-First Search (GBFS)*, yang mengutamakan kecepatan dengan memilih simpul terdekat menurut fungsi heuristik;
- *A* Search*, yang mengkombinasikan keduanya melalui fungsi $f(n) = g(n) + h(n)$, dengan $h(n)$ harus *admissible* (tidak melebihi-lebihkan) dan, jika memungkinkan, *consistent* (monotonik).

Dalam konteks permainan Rush Hour, heuristik yang sering digunakan adalah jumlah sel kosong antara ekor 'P' dan pintu keluar, ditambah bobot untuk setiap kendaraan penghalang. Heuristik ini sederhana, mudah dihitung, dan terbukti cukup efektif dalam mempercepat pencarian.

Tugas kecil ini bertujuan untuk:

1. Mengimplementasikan UCS, GBFS, dan A* pada konfigurasi papan generik berukuran hingga 6×6 , termasuk posisi rintangan 'X' dan pintu 'K' di empat sisi;
2. Membandingkan performa ketiga algoritma berdasarkan jumlah simpul yang diekspansi, waktu eksekusi, dan panjang solusi;
3. Menguji bagaimana fungsi heuristik memengaruhi efisiensi A* dalam mencapai solusi optimal.

Untuk memudahkan eksperimen, dikembangkan antarmuka grafis sederhana menggunakan

Java Swing. User dapat:

- Memuat berkas input yang menjelaskan papan dan rintangan;
- Menjalankan salah satu algoritma dan melihat langkah penyelesaian secara visual;
- Menyimpan hasil analisis dalam format teks, lengkap dengan kondisi papan di setiap langkah.



Gambar 1: Rush Hour Puzzle
(sumber: best-toys.net)

BAB II: LANDASAN TEORI

2.1 Representasi State dan Graph-Search

Permasalahan pada Rush Hour dapat direpresentasikan sebagai graf berarah (*directed graph*) di mana:

- Simpul (*node*) mewakili konfigurasi papan—posisi semua kendaraan dan rintangan 'X'.
- Busur (*edge*) menghubungkan dua simpul jika konfigurasi kedua dapat dicapai dari konfigurasi pertama dengan “menggeser” satu kendaraan sejauh satu atau beberapa cell sesuai orientasi.
- Biaya setiap busur ($\text{cost}(u \rightarrow v)$) diasumsikan sama dengan $|\Delta|$, jumlah cell yang digeser. Bila di-set 1 per langkah unit, UCS akan meniru BFS.
- Goal tercapai ketika primary piece (P) menempati atau melewati lokasi pintu keluar (K).

Mencari solusi tersingkat berarti menemukan jalur (lintasan) berbiaya minimum dari simpul awal ke simpul *goal*. Graph-search tradisional menggunakan struktur *open set* (*fringe*) dan *closed set* (*visited*) untuk menghindari siklus dan eksplorasi berlebihan.

2.2 Uniform-Cost Search (UCS)

Uniform-Cost Search adalah generalisasi dari algoritma **Dijkstra** untuk pencarian jalur pada graf dengan bobot non-negatif. Karakteristiknya antara lain:

- $f(n) = g(n)$
- $g(n)$ = akumulasi biaya terendah dari root ke simpul n , dihitung sebagai penjumlahan $\text{cost}(\text{edge})$ sepanjang jalur.
- **Open set**: priority queue terurut berdasarkan nilai terkecil $g(n)$.
- **Closed set**: simpul yang sudah diekstraksi (ditandai “visited”).

Algoritma:

1. Inisialisasi $\text{open} = \{\text{start}\}$ dengan $g(\text{start})=0$.
2. Ulangi sampai open kosong atau menemukan goal:
 - Ambil n dengan $g(n)$ terendah dari open .
 - Jika $n = \text{goal}$, return solusi.
 - Tandai n sebagai visited; untuk tiap tetangga m dari n :

- Hitung $g' = g(n) + \text{cost}(n \rightarrow m)$.
- Jika m belum dikunjungi atau $g' < g(m)$, set $g(m) = g'$ dan enqueue m .

Keuntungan:

- *Optimal* (menjamin solusi minimum) bila semua edge $\text{cost} \geq 0$.
- *Complete* (menemukan solusi jika ada) karena akan mengeksplorasi semua simpul dengan $g \leq g(\text{goal})$.
- *Kompleksitas waktu* $O(|E| + |V| \log |V|)$ atau secara kasar $O(b^d)$, di mana b = branching factor, d = banyaknya solusi.

2.3 Greedy Best-First Search (GBFS)

Greedy Best-First Search adalah algoritma berbasis **heuristik semata-mata** yang hanya mengandalkan estimasi jarak menuju *goal* tanpa mempertimbangkan biaya sejauh ini. Ciri-cirinya adalah sebagai berikut:

- $f(n) = h(n)$
- $g(n)$ = diabaikan (atau 0)
- $h(n)$ = estimasi jarak dari n ke goal, biasanya *admissible* (tidak melebihi biaya riil).
- *Open set*: priority queue terurut berdasarkan nilai $h(n)$.
- *Closed set*: simpul yang telah di-pop.

Algoritma:

1. Inisialisasi $\text{open} = \{\text{start}\}$ dengan $h(\text{start})$.
2. Ulangi:
 - Pop n dengan $h(n)$ terkecil.
 - Jika $n = \text{goal}$, stop.
 - Tandai n *visited*; untuk tiap tetangga m , hitung $h(m)$ dan enqueue jika belum visited.

Keuntungan dan Kelemahan GBFS:

- Tidak menjamin optimal, karena hanya mengejar estimasi, dapat melewati jalur biaya rendah tetapi h tinggi.
- Tidak lengkap bila heuristik menyesatkan, dapat terjebak tanpa mencapai goal.
- Cepat dan hemat memori jika heuristik sangat informatif.
- Kompleksitas tergantung kualitas heuristik: terbaik $O(d)$ jika $h(n) \rightarrow 0$ di sepanjang jalur, terburuk $O(b^m)$ dimana m = kedalaman simpul terakhir yang

diperiksa.

2.4 A* Search

A* Search mengkombinasikan keunggulan *UCS* dan *GBFS*: menggunakan nilai **gabungan** antara biaya sejauh ini dan estimasi ke depan.

- $f(n) = g(n) + h(n)$
- $g(n)$ = biaya kumulatif dari root ke n
- $h(n)$ = estimasi biaya minimum dari n ke goal
- Open set: priority queue berdasarkan $f(n)$ terkecil.
- Closed set: simpul yang sudah di-*expand*.

Algoritma:

1. open = {start}, $g(\text{start})=0$, $h(\text{start})$
2. Sambil open tidak kosong:
 - Pop n dengan $f(n)$ terendah.
 - Jika n = goal, return path.
 - Untuk setiap tetangga m :
 - Hitung $g' = g(n) + \text{cost}(n \rightarrow m)$, $h' = h(m)$, $f' = g' + h'$.
 - Jika m belum di-open/closed atau $g' < g(m)$, perbarui $g(m) = g'$, enqueue/update m .

Syarat Optimalitas:

- Heuristik $h(n)$ harus *admissible* ($\forall n, h(n) \leq h^*(n)$ di mana $h^*(n)$ = biaya sebenarnya terkecil ke goal).
- Jika h juga konsisten (monotonik):
 $h(n) \leq \text{cost}(n \rightarrow m) + h(m)$ untuk setiap tetangga m , maka A* menjamin $f(n)$ tidak menurun sepanjang jalur.

Keunggulan:

- Optimal dan lengkap bila h *admissible*.
- Efisien: memeriksa paling sedikit simpul di antara semua algoritma yang menggunakan heuristik *admissible*.
- Kompleksitas $O(b^{d-\epsilon})$ di mana ϵ tergantung ketajaman heuristik. Semakin informatif h , semakin kecil eksplorasi

BAB III: SPESIFIKASI TUGAS

- Buatlah program sederhana dalam bahasa **C/C++/Java/Javascript** yang mengimplementasikan **algoritma *pathfinding Greedy Best First Search*, UCS (*Uniform Cost Search*), dan *A**** dalam menyelesaikan permainan Rush Hour.
- Tugas dapat dikerjakan **individu atau berkelompok** dengan anggota **maksimal 2 orang** (sangat disarankan). Boleh lintas kelas dan lintas kampus, tetapi **tidak boleh sama** dengan anggota kelompok pada **tugas kecil Strategi Algoritma sebelumnya**.
- Algoritma *pathfinding* minimal menggunakan **satu heuristic** (2 atau lebih jika mengerjakan *bonus*) yang ditentukan sendiri. Jika mengerjakan *bonus*, *heuristic* yang digunakan ditentukan berdasarkan input pengguna.
- Algoritma dijalankan secara terpisah. Algoritma yang digunakan ditentukan berdasarkan Input pengguna.
- **Alur Program:**
 1. **[INPUT] konfigurasi permainan/test case** dalam format ekstensi **.txt**. File *test case* tersebut berisi:
 1. **Dimensi Papan** terdiri atas dua buah variabel **A** dan **B** yang membentuk papan berdimensi $A \times B$
 2. **Banyak *piece* BUKAN *primary piece*** direpresentasikan oleh variabel integer **N**.
 3. **Konfigurasi papan** yang mencakup penempatan *piece* dan *primary piece*, serta lokasi *pintu keluar*. *Primary Piece* dilambangkan dengan huruf **P** dan *pintu keluar* dilambangkan dengan huruf **K**. *Piece* dilambangkan dengan huruf dan karakter selain **P** dan **K**, dan huruf/karakter berbeda melambangkan *piece* yang berbeda. *Cell* kosong dilambangkan dengan karakter **'.'** (titik). (**Catatan:** ingat bahwa *pintu keluar* pasti berada di *dinding* papan dan sejajar dengan orientasi *primary piece*)

File .txt yang akan dibaca memiliki format sebagai berikut:

```
A B
N
konfigurasi_papan
```

Contoh Input

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


GPPCDFK

GH.III

GHJ...

LLJMM.

keterangan: “K” adalah pintu keluar, “P” adalah primary piece, Titik (“.”) adalah cell kosong.

Contoh konfigurasi papan lain yang mungkin berdasarkan letak *pintu keluar* (X adalah *piece/cell random*)

K	XXX	XXX
XXX	KXXX	XXX
XXX	XXX	XXX
XXX		K

2. [INPUT] **algoritma *pathfinding*** yang digunakan
3. [INPUT] ***heuristic*** yang digunakan (**bonus**)
4. [OUTPUT] Banyaknya **gerakan** yang diperiksa (alias banyak ‘node’ yang dikunjungi)
5. [OUTPUT] Waktu eksekusi program
6. [OUTPUT] **konfigurasi *papan*** pada setiap tahap pergerakan/pergeseran. Output ini tidak harus diimplementasi apabila mengerjakan *bonus output GUI*. **Gunakan print berwarna** untuk menunjukkan pergerakan *piece* dengan jelas. Cukup mewarnakan ***primary piece***, ***pintu keluar***, dan ***piece yang digerakkan*** saja (boleh dengan *highlight* atau *text color*). Pastikan ketiga komponen tersebut memiliki warna berbeda.

Format sekuens adalah sebagai berikut:

Papan Awal

[konfigurasi_papan_awal]

Gerakan 1: [piece]-[arah gerak]

[konfigurasi_papan_gerakan_1]

Gerakan 2: [piece]-[arah gerak]

[konfigurasi_papan_gerakan_2]

Gerakan [N]: [piece]-[arah gerak]

[konfigurasi_papan_gerakan_N]

dst

Contoh Output

Papan Awal

AAB..F
..BCDF
G**PP**CDF**K**
GH.III
GHJ...
LLJMM.

Gerakan 1: I-kiri

AAB..F
..BCDF
G**PP**CDF**K**
GH**III**.
GHJ...
LLJMM.

Gerakan 2: F-bawah

AAB..**F**
..BCD**F**
G**PP**CDF**K**
GHIII**F**
GHJ...
LLJMM.

dst

Keterangan: **hanya sebagai contoh.** Pastikan output jelas dan mudah dimengerti. Warna dan highlight hanya untuk menunjukkan perubahan.

7. **[OUTPUT] animasi** gerakan-gerakan untuk mencapai solusi (**bonus GUI**).

BAB IV: IMPLEMENTASI ALGORITMA

4.1 Struktur Kode dan Class (Java)

```
|— Main.java
|— Parser.java
|— Printer.java
|— Board.java
|   |— record Pos          // (r,c) untuk rintangan 'X'
|   |— class Piece         // id, row, col, length, orient
|   |— class State         // Board state, parent, Move, g, h, isGoal
|   |— enum Orientation    // HORIZONTAL, VERTICAL
|   |— methods:
|       • parse(...)       // baca file, deteksi pintu 4 sisi,   rintang
|       • expand(State)    // gen semua child State
|           - slideHorizontal(...)
|           - slideVertical(...)
|       • createChild(...) // buat State baru & grid baru
|— Searchers.java
|   |— interface Pathfinder { search(Board,Heuristic) }
|   |— class SearchResult  // goal State + visitedCount
|   |— UniformCostSearch   // implements Pathfinder
|   |— GreedyBestFirstSearch // implements Pathfinder
|   |— AStarSearch         // implements Pathfinder
|— Heuristics.java
|   |— interface Heuristic { estimate(Board) }
|   |— class Heuristics
|       • H1, H2, H3       // tiga fungsi h(n)
|       • byId(int)        // selector
```

4.2 Input Parser dan Representasi Board (Papan)

Proses memuat konfigurasi papan dimulai di `Parser.load()` yang membaca seluruh baris dari berkas teks input. Baris pertama berisi dimensi papan (baris dan kolom), sedangkan baris kedua yang memuat jumlah kendaraan non-primary hanya dilewati tanpa diproses lebih lanjut. Sisanya dikumpulkan sebagai “rawRows”.

Ketika membaca rawRows, parser mengecek baris yang hanya berisi huruf ‘K’: jika ini adalah baris pertama dari kumpulan, maka pintu keluar terletak tepat di atas grid; jika muncul setelah tepat rows baris, maka pintu keluar ada di bawah. Baris yang panjangnya satu lebih besar dari kolom, dengan ‘K’ di ujung kiri atau kanan, ditafsirkan sebagai pintu di samping grid, lalu ‘K’ dibuang untuk menjaga agar setiap baris papan persis sepanjang kolom. Setelah teridentifikasi, tepat rows baris papan diubah menjadi matriks `char[rows][cols]`, dengan sel ‘.’ untuk kosong, huruf selain ‘.’ dan ‘X’ membentuk peta pieces, dan karakter ‘X’ ditandai sebagai halangan statis (tidak dapat bergerak).

4.3 Perluasan Child State (Status Anak)

Setiap objek Board menyimpan peta kendaraan, rintangan, dan posisi pintu keluar, serta menyediakan metode `expand(State parent)` untuk menghasilkan daftar State berikutnya. Di dalamnya, untuk tiap Piece yang horizontal, dilakukan dua loop: geser ke kiri sejauh sel kosong hingga bertemu halangan atau tepi, lalu geser ke kanan—dan serupa untuk orientasi vertikal. Setiap pergeseran sepanjang delta sel memanggil `createChild(parent, piece, delta)`, yang membuat salinan peta kendaraan, memindahkan satu kendaraan, lalu membangun ulang grid baru beserta rintangan. Bila primary piece dapat mencapai posisi “keluar” di salah satu sisi papan—kiri, kanan, atas, atau bawah—maka dibuat child khusus dengan `isGoal=true` dan delta yang tepat agar melintasi tepi grid, menandai solusi.

BAB V: KODE SUMBER

Main.java:

```
public class Main {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(() -> {
            new MainGUI().setVisible(true);
        });
    }
}
```

MainGUI.java:

```
import java.awt.*;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;
import javax.swing.*;

public class MainGUI extends JFrame {
    private final JTextField fileField = new JTextField(20);
    private final JComboBox<String> algBox = new JComboBox<>(new
String[]{"ucs", "gbfs", "astar"});
    private final JComboBox<Integer> hBox = new JComboBox<>(new
Integer[]{1,2,3});
    private final BoardPanel boardPanel = new BoardPanel();
    private final JButton prevBtn = new JButton("Prev"), nextBtn = new
JButton("Next");
```

```

private List<State> path; private int currentIndex;
private final JLabel statusLabel;
private long execTime;
private long visitedCount;
private final JButton saveBtn = new JButton("Save Results");
private final JButton playBtn = new JButton("Stop");
private Timer playTimer;

public MainGUI() {
    super("RushHour Solver");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JPanel top = new JPanel();
    statusLabel = new JLabel("Node Dikunjungi: 0 Waktu: 0ms Step: 0/0");
    JButton browse = new JButton("Browse...");
    browse.addActionListener(e -> {
        JFileChooser fc = new JFileChooser();
        if (fc.showOpenDialog(this)==JFileChooser.APPROVE_OPTION)
            fileField.setText(fc.getSelectedFile().getAbsolutePath());
    });
    top.add(new JLabel("File:")); top.add(fileField); top.add(browse);
    top.add(new JLabel("Algoritma:")); top.add(algBox);
    top.add(new JLabel("H-ID:")); top.add(hBox);
    JButton solve = new JButton("Pecahkan");
    solve.addActionListener(e -> doSolve());
    top.add(solve);

    // menambahkan panel papan ke dalam panel utama
    add(top, BorderLayout.NORTH);
    add(boardPanel, BorderLayout.CENTER);
    JPanel nav = new JPanel();
    prevBtn.addActionListener(e -> {
        if (path!=null && currentIndex>0) { currentIndex--; updateBoard();
    }

    });
    nextBtn.addActionListener(e -> {
        if (path!=null && currentIndex<path.size()-1) { currentIndex++;
updateBoard(); }
    });
    saveBtn.addActionListener(e -> doSave());
    nav.add(prevBtn); nav.add(nextBtn);
    nav.add(saveBtn);

```

```

        nav.add(playBtn);
        // menyatukan tombol navigasi dan status label
        JPanel bottom = new JPanel(new BorderLayout());
        bottom.add(nav, BorderLayout.WEST);
        bottom.add(statusLabel, BorderLayout.EAST);
        add(bottom, BorderLayout.SOUTH);
        // setup timer auto-play
        playTimer = new Timer(1000, e -> autoNext());
        playTimer.start();
        playBtn.addActionListener(e -> {
            if (playTimer.isRunning()) stopSlidePlay();
            else startSlidePlay();
        });
        pack();
        setLocationRelativeTo(null);
    }

    // auto-play next step
    private void autoNext() {
        if (path != null && currentIndex < path.size() - 1) {
            currentIndex++;
            SwingUtilities.invokeLater(this::updateBoard);
        } else {
            stopSlidePlay();
        }
    }

    private void startSlidePlay() {
        // Kalau sudah sampai akhir, ulangi dari awal
        if (path != null && currentIndex >= path.size() - 1) {
            currentIndex = 0;
            updateBoard();
        }
        playTimer.start();
        playBtn.setText("Stop");
    }

    private void stopSlidePlay() {
        playTimer.stop();
        playBtn.setText("Play");
    }
}

```

```

private void doSolve() {
    try {
        String file = fileField.getText().trim();
        String alg = ((String)algBox.getSelectedItem()).toLowerCase();
        int hid = (Integer)hBox.getSelectedItem();
        Board start = Parser.load(file);
        Pathfinder pf = switch (alg) {
            case "ucs" -> new UniformCostSearch();
            case "gbfs" -> new GreedyBestFirstSearch();
            case "astar" -> new AStarSearch();
            default -> throw new IllegalArgumentException();
        };
        Heuristic h = Heuristics.byId(hid);
        // mencari solusi
        long t0 = System.currentTimeMillis();
        SearchResult res = pf.search(start, h);
        execTime = System.currentTimeMillis() - t0;
        visitedCount = res.visitedCount();
        // menampilkan hasil pencarian
        path = res.path();
        currentIndex = 0;
        // error kalau tidak ada solusi
        if (path.isEmpty()) {
            JOptionPane.showMessageDialog(this,
                "Tidak ada solusi ditemukan.",
                "Error", JOptionPane.ERROR_MESSAGE);
            return;
        }
        updateBoard();
        // auto-play setelah menyelesaikan pencarian
        startSlidePlay();
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(this, ex.getMessage(),
            "Error", JOptionPane.ERROR_MESSAGE);
    }
}

private void updateBoard() {
    boardPanel.setBoard(path.get(currentIndex).board);
    // update status label
}

```

```

        statusLabel.setText(String.format("Node Dikunjungi: %d Waktu: %dms  
Step: %d/%d", visitedCount, execTime, currentIndex+1, path.size()));
    }

    // save hasil pencarian ke file
    private void doSave() {
        if (path == null) {
            JOptionPane.showMessageDialog(this, "No results to save.", "Info",
JOptionPane.INFORMATION_MESSAGE);
            return;
        }
        JFileChooser fc = new JFileChooser();
        fc.setDialogTitle("Save Results");
        fc.setSelectedFile(new File("results.txt"));
        if (fc.showSaveDialog(this) == JFileChooser.APPROVE_OPTION) {
            File out = fc.getSelectedFile();
            // periksa apakah file sudah ada
            if (out.exists()) {
                int choice = JOptionPane.showConfirmDialog(this,
                    "File '" + out.getName() + "' already exists.  
Overwrite?",
                    "Confirm Overwrite", JOptionPane.YES_NO_OPTION,
                    JOptionPane.WARNING_MESSAGE);
                if (choice != JOptionPane.YES_OPTION) return;
            }
            try (FileWriter fw = new FileWriter(out)) {
                fw.write(buildResultText());
                JOptionPane.showMessageDialog(this, "Results saved to " +
out.getAbsolutePath(), "Saved", JOptionPane.INFORMATION_MESSAGE);
            } catch (IOException ex) {
                JOptionPane.showMessageDialog(this, "Error saving file: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    // Membangun teks hasil pencarian
    private String buildResultText() {
        StringBuilder sb = new StringBuilder();
        sb.append("Node Dikunjungi: ").append(visitedCount).append("\n")
            .append("Waktu (ms): ").append(execTime).append("\n")

```



```

        .append("Algoritma: ").append(algBox.getSelectedItem()).append("\n")
        .append("H-ID: ").append(hBox.getSelectedItem()).append("\n\n");
        for (int i = 0; i < path.size(); i++) {
            State s = path.get(i);
            if (i == 0) sb.append("Kondisi Awal Papan:\n");
            else sb.append("Step ").append(i).append(":
").append(s.move).append("\n");
            // hapus ANSI escape codes
            String plain = stripAnsi(Printer.pretty(s.board, s.move==null ?
'\0' : s.move.piece));
            sb.append(plain).append("\n");
        }
        return sb.toString();
    }

    private String stripAnsi(String s) {
        return s.replaceAll("\\x1B\\[[;\\d]*m", "");
    }
}

```

AStarSearch.java:

```

import java.util.*;

public class AStarSearch implements Pathfinder {
    @Override
    public SearchResult search(Board start, Heuristic h) {
        PriorityQueue<State> open = new PriorityQueue<>();
        Map<Board, Integer> best = new HashMap<>();
        State root = new State(start, null, new Move('-', 0), 0);
        root.h = h.estimate(start); open.add(root); best.put(start, 0);
        long visited = 0;
        while (!open.isEmpty()) {
            State cur = open.poll(); visited++;
            if (cur.isGoal) return new SearchResult(cur, visited);
            for (State nxt : cur.board.expand(cur)) {
                nxt.h = h.estimate(nxt.board);
                if (best.getDefault(nxt.board, Integer.MAX_VALUE) > nxt.g) {
                    best.put(nxt.board, nxt.g);
                    open.add(nxt);
                }
            }
        }
    }
}

```

```

    }
}
return new SearchResult(null, visited);
}
}

```

Board.java:

```

import java.util.*;

public class Board {
    // atribut public final untuk memudahkan akses
    public final int rows, cols;
    public final char[][] grid; // '.' untuk kosong, 'X' untuk
    halangan
    public final Map<Character, Piece> pieces; // id -> Piece (tidak termasuk
    exit)
    public final Piece primary; // primary piece 'P'
    public final int exitRow, exitCol; // koordinat 'K'

    public record Pos(int r,int c){}
    public final Set<Pos> obstacles;

    public Board(int rows, int cols, char[][] grid,
        Map<Character, Piece> pieces, Piece primary,
        int exitRow, int exitCol, Set<Pos> obstacles) {
        this.rows = rows; this.cols = cols;
        // Deep copy grid
        this.grid = new char[grid.length][];
        for (int i = 0; i < grid.length; i++) {
            this.grid[i] = Arrays.copyOf(grid[i], grid[i].length);
        }
        this.pieces = pieces;
        this.primary = primary; this.exitRow = exitRow; this.exitCol = exitCol;
        this.obstacles = obstacles;
    }

    // Parse input dari file
    public static Board parse(List<String> lines) {
        if (lines.size() < 3) throw new IllegalArgumentException("Input too
    short");
    }
}

```

```

String[] dim = lines.get(0).trim().split("\\s+");
int R = Integer.parseInt(dim[0]);
int C = Integer.parseInt(dim[1]);
int idx = 1;
// Baris kedua (banyaknya piece) bisa diabaikan
// karena dihitung dari banyaknya karakter unik di baris-baris
berikutnya
try { Integer.valueOf(lines.get(idx).trim()); idx++; } catch
(NumberFormatException ignored) {}

if (lines.size() - idx < R)
    throw new IllegalArgumentException("Not enough board rows -
expected " + R);

char[][] g = new char[R][C];
for (char[] row : g) Arrays.fill(row, '.');

Map<Character, List<int[]>> locs = new HashMap<>();
int exR = -2, exC = -2;

List<String> rawRows = lines.subList(idx, lines.size());
List<String> boardLines = new ArrayList<>();
for (int i = 0; i < rawRows.size(); i++) {
    String rawLine = rawRows.get(i);
    String trim = rawLine.trim();
    // baris hanya 'K'? berarti pintu atas (jika boardLines kosong)
    // atau pintu bawah (jika boardLines sudah lengkap)
    if (trim.equals("K")) {
        int c = rawLine.indexOf('K');
        if (boardLines.isEmpty()) {
            exR = -1;    // exit di atas baris-0
            exC = c;
        } else if (boardLines.size() == R) {
            exR = R;    // exit di bawah baris-(R-1)
            exC = c;
        } else {
            throw new IllegalArgumentException("Baris 'K' muncul di
tengah input");
        }
        continue; // jangan masukkan ke boardLines
    }
}

```

```

        // normal row -> harus jadi bagian board
        boardLines.add(rawLine);
    }
    if (boardLines.size() < R)
        throw new IllegalArgumentException("Tidak cukup baris board;
dibutuhkan " + R);

    // 2) Sekarang isi grid dari boardLines[0..R-1], deteksi juga K di
kiri/kanan:
    Set<Pos> obstacles = new HashSet<>();
    for (int r = 0; r < R; r++) {
        // preserve original spacing to handle side-exit blanks
        String rawLine = boardLines.get(r);
        String raw = rawLine;
        // deteksi K di kiri/kanan seperti sebelumnya:
        if (raw.length() == C + 1) {
            if (raw.charAt(0) == 'K') {
                exR = r; exC = -1;
                raw = raw.substring(1);
            } else if (raw.charAt(raw.length()-1) == 'K') {
                exR = r; exC = C;
                raw = raw.substring(0, raw.length()-1);
            } else if (raw.charAt(0) == ' ') {
                // ignore leading blank column for left-side exit
                raw = raw.substring(1);
            } else if (raw.charAt(raw.length()-1) == ' ') {
                // ignore trailing blank column for right-side exit
                raw = raw.substring(0, raw.length()-1);
            } else {
                throw new IllegalArgumentException("Extra char bukan 'K' di
row " + r);
            }
        }
        // after stripping, ensure proper length
        raw = raw.trim();
        if (raw.length() != C) {
            throw new IllegalArgumentException("Row " + r + " length " +
raw.length() + " != " + C);
        }
        // isi cell dan kumpulkan lokasi piece
        for (int c = 0; c < C; c++) {

```

```

        char ch = raw.charAt(c);
        if (ch == 'X') {
            g[r][c] = 'X';
            obstacles.add(new Pos(r,c));
        }
        else if (ch != '.') {
            g[r][c] = ch;
            locs.computeIfAbsent(ch, k -> new ArrayList<>()).add(new
int[] {r,c});
        }
    }
}

if (exR == -2)
    throw new IllegalArgumentException("Exit 'K' tidak ditemukan di
sisi mana pun");

Map<Character, Piece> pcs = new HashMap<>();
Piece prim = null;
for (var e : locs.entrySet()) {
    char id = e.getKey();
    List<int[]> cells = e.getValue();
    int len = cells.size();
    int r0 = cells.get(0)[0], c0 = cells.get(0)[1];
    boolean horizontal = cells.stream().allMatch(p -> p[0] == r0);
    Orientation o = horizontal ? Orientation.HORIZONTAL :
Orientation.VERTICAL;
    Piece p = new Piece(id, r0, c0, len, o);
    pcs.put(id, p);
    if (id == 'P') prim = p;
}
if (prim == null) throw new IllegalArgumentException("Primary piece 'P'
not found");

return new Board(R, C, g, pcs, prim, exR, exC,
Collections.unmodifiableSet(obstacles));
}

/* ----- Neighbours ----- */
public List<State> expand(State parent) {
    List<State> next = new ArrayList<>();
    for (Piece p : pieces.values()) {
        if (p.orient == Orientation.HORIZONTAL) {

```

```

        slideHorizontal(parent, next, p);
    } else {
        slideVertical(parent, next, p);
    }
}
return next;
}

private void slideHorizontal(State parent, List<State> out, Piece p) {
    // left
    int c = p.col - 1;
    while (c >= 0 && grid[p.row][c] == '.') { out.add(createChild(parent,
p, c - p.col)); c--; }
    // right
    c = p.tailCol() + 1;
    while (c < cols && grid[p.row][c] == '.') { out.add(createChild(parent,
p, c - p.tailCol())); c++; }
    // special: horizontal exit for primary piece
    if (p.id == 'P' && p.row == exitRow) {
        // exit on right side
        if (exitCol == cols) {
            boolean clear = true;
            for (int cc = p.tailCol() + 1; cc < cols; cc++) {
                if (grid[p.row][cc] != '.') { clear = false; break; }
            }
            if (clear) {
                State goal = createChild(parent, p, cols - p.tailCol());
                goal.isGoal = true; out.add(goal);
            }
        }
        // exit on left side
        else if (exitCol == -1) {
            boolean clear = true;
            for (int cc = p.col - 1; cc >= 0; cc--) {
                if (grid[p.row][cc] != '.') { clear = false; break; }
            }
            if (clear) {
                int delta = -(p.col + 1);
                State goal = createChild(parent, p, delta);
                goal.isGoal = true; out.add(goal);
            }
        }
    }
}

```

```

    }
}

private void slideVertical(State parent, List<State> out, Piece p) {
    // geser ke atas
    int r = p.row - 1;
    while (r >= 0 && grid[r][p.col] == '.') {
        out.add(createChild(parent, p, r - p.row));
        r--;
    }
    // geser ke bawah
    r = p.tailRow() + 1;
    while (r < rows && grid[r][p.col] == '.') {
        out.add(createChild(parent, p, r - p.tailRow()));
        r++;
    }
    // special: exit vertikal (atas/bawah)
    if (p.id == 'P' && p.col == exitCol) {
        // exit di bawah (exitRow == rows)
        if (exitRow == rows) {
            boolean clear = true;
            for (int rr = p.tailRow() + 1; rr < rows; rr++) {
                if (grid[rr][p.col] != '.') { clear = false; break; }
            }
            if (clear) {
                // geser P sampai keluar bawah
                State goal = createChild(parent, p, rows - p.tailRow());
                goal.isGoal = true;
                out.add(goal);
            }
        }
        // exit di atas (exitRow == -1)
        else if (exitRow == -1) {
            boolean clear = true;
            for (int rr = p.row - 1; rr >= 0; rr--) {
                if (grid[rr][p.col] != '.') { clear = false; break; }
            }
            if (clear) {
                // geser P sampai keluar atas
                int delta = -(p.row + 1);

```

```

        State goal = createChild(parent, p, delta);
        goal.isGoal = true;
        out.add(goal);
    }
}

private State createChild(State parent, Piece moving, int delta) {
    Map<Character, Piece> np = new HashMap<>(pieces);
    Piece mv = moving.moved(delta);
    np.put(mv.id, mv);

    // rebuild grid (cheap for 6x6 boards)
    char[][] ng = new char[rows][cols];
    for (char[] row : ng) Arrays.fill(row, '.');
    for (Piece pc : np.values()) {
        for (int i = 0; i < pc.length; i++) {
            int r = pc.orient == Orientation.HORIZONTAL ? pc.row : pc.row +
i;

            int c = pc.orient == Orientation.HORIZONTAL ? pc.col + i :
pc.col;

            if (r >= 0 && r < rows && c >= 0 && c < cols) {
                ng[r][c] = pc.id;
            }
        }
    }
    for (Pos o : obstacles) ng[o.r()][o.c()] = 'X';
    int newG = parent == null ? Math.abs(delta) : parent.g +
Math.abs(delta);
    return new State(new Board(rows, cols, ng, np, np.get('P'), exitRow,
exitCol, obstacles),
        parent, new Move(moving.id, delta), newG);
}

/* ----- Utils ----- */
@Override public boolean equals(Object o) { return o instanceof Board b &&
Arrays.deepEquals(grid, b.grid); }
@Override public int hashCode() { return
Arrays.deepHashCode(grid); }
@Override public String toString() {

```



```

        StringBuilder sb = new StringBuilder();
        for (char[] row : grid) { sb.append(row); sb.append('\n'); }
        return sb.toString();
    }
}

```

BoardPanel.java:

```

import java.awt.*;
import java.util.*;
import javax.swing.*;

public class BoardPanel extends JPanel {
    private Board board;
    private final Map<Character,Color> colorMap = new HashMap<>();
    private static final Color EMPTY    = Color.WHITE;
    private static final Color OBSTACLE = Color.DARK_GRAY;
    private static final Color EXIT     = Color.GREEN;

    public BoardPanel() {
        setPreferredSize(new Dimension(360,360));
    }

    public void setBoard(Board b) {
        this.board = b;
        assignColors(b);
        repaint();
    }

    private void assignColors(Board b) {
        for (char id : b.pieces.keySet()) {
            colorMap.computeIfAbsent(id, k -> {
                if (k=='P') return Color.RED;
                float hue = ((k*13)%360)/360f;
                return Color.getHSBColor(hue,0.5f,0.9f);
            });
        }
        colorMap.put('X', OBSTACLE);
    }

    @Override
    protected void paintComponent(Graphics g) {

```

```

super.paintComponent(g);
if (board == null) return;

int R = board.rows, C = board.cols;
int w = getWidth(), h = getHeight();
int cellW = w / C, cellH = h / R;
int offX = (w - C*cellW)/2, offY = (h - R*cellH)/2;

// grid
for (int r = 0; r < R; r++) {
    for (int c = 0; c < C; c++) {
        char id = board.grid[r][c];
        Color col = switch (id) {
            case 'X'      -> OBSTACLE;
            case '.'      -> EMPTY;
            default       -> colorMap.getOrDefault(id, EMPTY);
        };
        g.setColor(col);
        g.fillRect(offX + c*cellW, offY + r*cellH, cellW, cellH);
        g.setColor(Color.BLACK);
        g.drawRect(offX + c*cellW, offY + r*cellH, cellW, cellH);
    }
}

// exit
g.setColor(EXIT);
int er = board.exitRow, ec = board.exitCol;
if (er == -1) // atas
    g.fillRect(offX + ec*cellW, offY, cellW, cellH/4);
else if (er == R) // bawah
    g.fillRect(offX + ec*cellW, offY + R*cellH - cellH/4, cellW,
cellH/4);
else if (ec == -1) // kiri
    g.fillRect(offX, offY + er*cellH, cellW/4, cellH);
else if (ec == C) // kanan
    g.fillRect(offX + C*cellW - cellW/4, offY + er*cellH, cellW/4,
cellH);
}
}

```

GreedyBestFirstSearch.java:

```
import java.util.*;

public class GreedyBestFirstSearch implements Pathfinder {
    @Override
    public SearchResult search(Board start, Heuristic h) {
        PriorityQueue<State> open = new
PriorityQueue<>(Comparator.comparingInt(s -> s.h));
        Set<Board> closed = new HashSet<>();
        State root = new State(start, null, new Move('-', 0), 0);
        root.h = h.estimate(start); open.add(root);
        long visited = 0;
        while (!open.isEmpty()) {
            State cur = open.poll(); visited++;
            if (cur.isGoal) return new SearchResult(cur, visited);
            closed.add(cur.board);
            for (State nxt : cur.board.expand(cur)) {
                nxt.h = h.estimate(nxt.board);
                if (closed.contains(nxt.board)) continue;
                open.add(nxt);
            }
        }
        return new SearchResult(null, visited);
    }
}
```

Heuristic.java:

```
public interface Heuristic {
    int estimate(Board b);
}
```

Heuristics.java:

```
public class Heuristics {
    private Heuristics() {}

    /*
     * H1 - "mobil penghalang + jarak" (admissible)
     * f(n) = (#sel kosong antara ekor primary & exit)
     *       + 2 × (#potongan penghalang di sepanjang jalur tersebut)
     */
}
```

```

public static final Heuristic H1 = b -> {
    Piece p = b.primary;
    if (p.orient == Orientation.HORIZONTAL) {
        int dist = b.exitCol - p.tailCol() - 1; // cell kosong
        int blockers = 0;
        for (int c = p.tailCol() + 1; c < b.exitCol; c++) if
(b.grid[p.row][c] != '.') blockers++;
        return dist + blockers * 2;
    } else {
        int dist = b.exitRow - p.tailRow() - 1;
        int blockers = 0;
        for (int r = p.tailRow() + 1; r < b.exitRow; r++) if
(b.grid[r][p.col] != '.') blockers++;
        return dist + blockers * 2;
    }
};

/*-----
 * H2 - distance + 1xblocking (lebih longgar, tetap admissible)
 * Alasan: setiap kendaraan penghalang butuh  $\geq 1$  gerakan.
 *-----*/
public static final Heuristic H2 = b -> {
    Piece p = b.primary;
    if (p.orient == Orientation.HORIZONTAL) {
        int dist = b.exitCol - p.tailCol() - 1;
        int block = 0;
        for (int c = p.tailCol() + 1; c < b.exitCol; c++)
            if (b.grid[p.row][c] != '.') block++;
        return dist + block; // koefisien 1 -> selalu  $\leq$  biaya
real
    } else {
        int dist = b.exitRow - p.tailRow() - 1;
        int block = 0;
        for (int r = p.tailRow() + 1; r < b.exitRow; r++)
            if (b.grid[r][p.col] != '.') block++;
        return dist + block;
    }
};

/*-----

```

```

* H3 - distance + 2 × blocking + “secondary blocker” penalty
* 0 Hitung blocker langsung di jalur P -> K (seperti H1)
* - Tambah +1 lagi per kendaraan yang menghalangi blocker
* pertama bila ia hendak bergeser satu sel (estimasi kasar).
* - Jika sel rintangan 'X' tepat di jalur -> kembalikan
Integer.MAX_VALUE
* agar A* / GBFS praktis menghindari state tak-solvable.
* - Heuristik ini bias ke arah besar -> bisa *inadmissible*,
* namun sering memangkas eksplorasi.
*-----*/
public static final Heuristic H3 = b -> {
    Piece p = b.primary;

    // fungsi bantu: true bila koordinat dalam grid & bukan '.'
    java.util.function.BiPredicate<Integer,Integer> occupied = (r,c) ->
        r>=0 && r<b.rows && c>=0 && c<b.cols && b.grid[r][c] != '.';

    int dist, block = 0, secBlock = 0;

    if (p.orient == Orientation.HORIZONTAL) {
        dist = b.exitCol - p.tailCol() - 1;
        for (int c = p.tailCol()+1; c < b.exitCol; c++) {
            char id = b.grid[p.row][c];
            if (id == '.') continue;
            if (id == 'X') return Integer.MAX_VALUE; // dinding permanen
            block++;
            // cek satu sel di atas & bawah blocker, apakah juga terisi?
            if (occupied.test(p.row-1, c) || occupied.test(p.row+1, c))
                secBlock++;
        }
    } else {
        dist = b.exitRow - p.tailRow() - 1;
        for (int r = p.tailRow()+1; r < b.exitRow; r++) {
            char id = b.grid[r][p.col];
            if (id == '.') continue;
            if (id == 'X') return Integer.MAX_VALUE;
            block++;
            if (occupied.test(r, p.col-1) || occupied.test(r, p.col+1))
                secBlock++;
        }
    }
}

```

```

        return dist + block * 2 + secBlock;    // lebih agresif
    };

    // Selektor dari ID heuristik
    public static Heuristic byId(int id) {
        return switch (id) {
            case 1 -> H1;
            case 2 -> H2;
            case 3 -> H3;
            default -> H1;
        };
    }
}

```

Move.java:

```

public class Move {
    public final char piece; public final int delta;
    Move(char piece, int delta) { this.piece = piece; this.delta = delta; }
    @Override public String toString() {
        if (delta == 0) return piece + "-0";
        String dir = (delta < 0) ? "L" : "R";
        if (dir.equals("L") && Math.abs(delta) > 1) dir += Math.abs(delta);
        if (dir.equals("R") && delta > 1) dir += delta;
        return piece + '-' + dir;
    }
}

```

Orientation.java:

```

public enum Orientation { HORIZONTAL, VERTICAL }

```

Parser.java:

```

import java.nio.file.*;
import java.util.*;

public class Parser {
    public static Board load(String path) throws Exception {
        List<String> lines = Files.readAllLines(Path.of(path));
        return Board.parse(lines);
    }
}

```

PathFinder.java:

```
interface PathFinder { SearchResult search(Board start, Heuristic h); }
```

Piece.java:

```
public class Piece {
    public final char id; public final int row, col, length; public final
    Orientation orient;
    Piece(char id, int row, int col, int length, Orientation orient) {
        this.id=id; this.row=row; this.col=col; this.length=length;
        this.orient=orient;
    }
    int tailRow() { return orient == Orientation.VERTICAL ? row + length - 1
: row; }
    int tailCol() { return orient == Orientation.HORIZONTAL ? col + length - 1
: col; }
    Piece moved(int d) {
        return orient == Orientation.HORIZONTAL ? new Piece(id, row, col + d,
length, orient)
: new Piece(id, row + d, col,
length, orient);
    }
}
```

Printer.java:

```
import java.util.*;

public class Printer {
    private static final String RESET = "\u001B[0m";
    private static final String RED = "\u001B[31m";
    private static final String GREEN = "\u001B[32m";
    private static final String BLUE = "\u001B[34m";

    public static void printPath(List<State> path) {
        if (path.isEmpty()) {
            System.out.println("No solution found.");
            return;
        }
        System.out.println("Papan Awal\n" + pretty(path.get(0).board, '\0'));
        for (int i = 1; i < path.size(); i++) {
            State s = path.get(i);
```

```

        System.out.println("Gerakan " + i + ": " + s.move);
        System.out.println(pretty(s.board, s.move.piece));
    }
}

public static String pretty(Board b, char moved) {
    StringBuilder sb = new StringBuilder();

    // Baris ekstra di ATAS (exitRow == -1)
    if (b.exitRow == -1) {
        for (int c = 0; c < b.cols; c++)
            sb.append(c == b.exitCol ? GREEN + 'K' + RESET : ' ');
        sb.append('\n');
    }

    // Baris-baris grid
    for (int r = 0; r < b.rows; r++) {
        // K di kiri?
        if (b.exitCol == -1 && r == b.exitRow)
            sb.append(GREEN).append('K').append(RESET);

        for (int c = 0; c < b.cols; c++) {
            char ch = b.grid[r][c];
            if (ch == 'P')
                sb.append(RED).append(ch).append(RESET);
            else if (ch == moved && moved != '\0')
                sb.append(BLUE).append(ch).append(RESET);
            else
                sb.append(ch);
        }

        // K di kanan?
        if (b.exitCol == b.cols && r == b.exitRow)
            sb.append(GREEN).append('K').append(RESET);

        sb.append('\n');
    }

    // Baris ekstra di BAWAH (exitRow == rows)
    if (b.exitRow == b.rows) {
        for (int c = 0; c < b.cols; c++)

```



```

        sb.append(c == b.exitCol ? GREEN + 'K' + RESET : ' ');
        sb.append('\n');
    }

    return sb.toString();
}
}

```

SearchResult.java:

```

import java.util.*;

public class SearchResult {
    private final State goal; private final long visitedCount;
    public SearchResult(State goal, long visited) { this.goal = goal;
this.visitedCount = visited; }
    long visitedCount() { return visitedCount; }
    List<State> path() {
        if (goal == null) return List.of();
        List<State> rev = new ArrayList<>();
        for (State s = goal; s != null; s = s.parent) rev.add(s);
        Collections.reverse(rev); return rev;
    }
}

```

State.java:

```

public class State implements Comparable<State> {
    final Board board; final State parent; final Move move; final int g; int h;
    boolean isGoal = false;
    State(Board b, State p, Move m, int g) { board = b; parent = p; move = m;
this.g = g; }
    int f() { return g + h; }
    @Override public int compareTo(State o) { return Integer.compare(f(),
o.f()); }
}

```

UniformCostSearch.java:

```

import java.util.*;

public class UniformCostSearch implements Pathfinder {

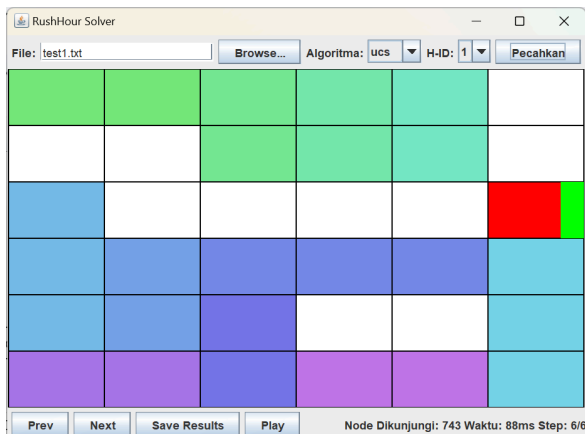

```



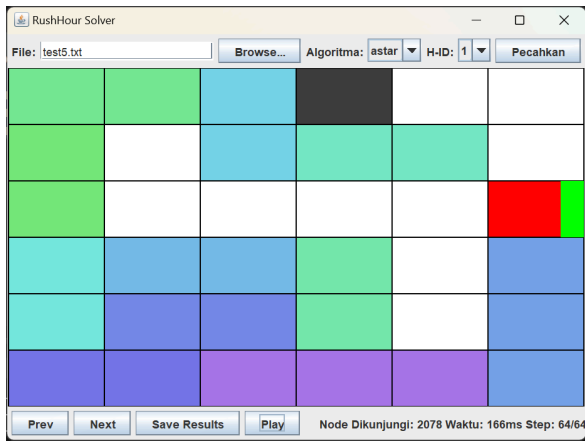

```

@Override
public SearchResult search(Board start, Heuristic h) {
    PriorityQueue<State> open = new
PriorityQueue<>(Comparator.comparingInt(s -> s.g));
    Map<Board, Integer> best = new HashMap<>();
    State root = new State(start, null, new Move('-', 0), 0);
    root.h = 0; open.add(root); best.put(start, 0);
    long visited = 0;
    while (!open.isEmpty()) {
        State cur = open.poll(); visited++;
        if (cur.isGoal) return new SearchResult(cur, visited);
        for (State nxt : cur.board.expand(cur)) {
            nxt.h = 0; // UCS ignores heuristic
            if (best.getDefault(nxt.board, Integer.MAX_VALUE) > nxt.g) {
                best.put(nxt.board, nxt.g);
                open.add(nxt);
            }
        }
    }
    return new SearchResult(null, visited);
}
}

```

BAB VI: PENGUJIAN ALGORITMA

Test case	Hasil
6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	
3 3 3 PPAK ..A .BB	

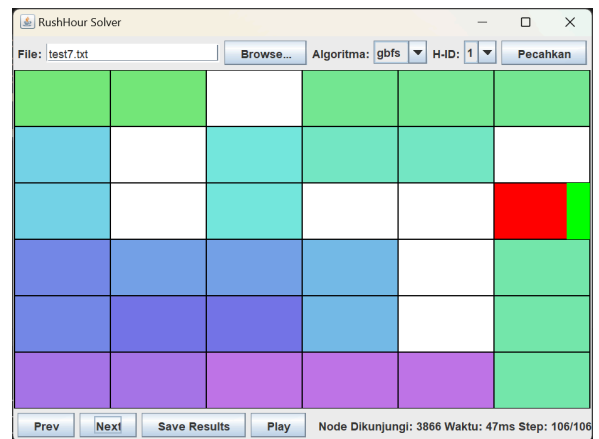
<pre> 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. </pre>	
<pre> 3 3 2 K BAA B.P ..P </pre>	
<pre> 6 6 12 ABBX.. A..CDD EPPC..K E.FGGH IIF..H JJLLH </pre>	
<pre> 4 4 3 BP.. BP.C BAAC K </pre>	

6 6
 13
 AABBBB
 DDE..C
 F.EPPCK
 F..GHH
 I..GJJ
 ILLMMM

UCS:



GBFS:



A*:



6 6
 12
 A.XBCC
 ADDB.E
 AFPP.EK
 .FGHHI
 JJG..I
 ..XLL.

Heuristic H1 (A*):



Heuristic H2 (A*):



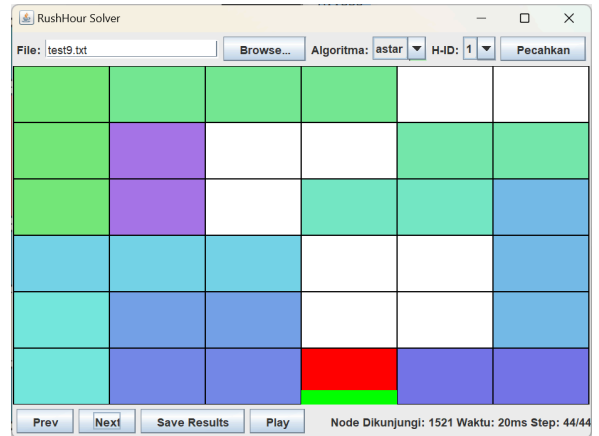
Heuristic H3 (A*):



```

6 6
11
A..BBB
ACCP..
AL.PDD
ELFFFG
EHH..G
IIJJ.G
  K

```



BAB VII: ANALISIS ALGORITMA

7.1 Definisi $f(n)$ dan $g(n)$

Dalam konteks graph-search untuk Rush Hour, setiap simpul n mewakili konfigurasi papan, dan gerakan menggeser sebuah kendaraan dari satu konfigurasi ke konfigurasi lain diberi bobot sesuai jumlah sel yang digeser.

- $g(n)$ didefinisikan sebagai akumulasi total biaya—yaitu jumlah langkah unit—yang telah ditempuh dari simpul awal (start) ke simpul n . Misalnya, jika sebuah kendaraan digeser 2 sel, g bertambah 2.
- $f(n)$ bergantung pada algoritma:
 - Pada UCS, $f(n) = g(n)$. Ini berarti UCS hanya mempertimbangkan jarak riil sejauh ini tanpa estimasi masa depan.
 - Pada **Greedy Best-First**, $f(n) = h(n)$, sehingga $g(n)$ sama sekali diabaikan. Algoritma hanya fokus pada estimasi jarak ke goal.
 - Pada **A***, $f(n) = g(n) + h(n)$, yaitu penjumlahan antara biaya sejauh ini dan estimasi biaya ke goal.

7.2 Admissibility Heuristik A*

Heuristik dikatakan *admissible* jika ia **tidak pernah melebihi** biaya riil minimum dari simpul n ke goal. Heuristik H1 (jarak sel kosong ke pintu + $2 \times$ jumlah blocker) dan H2 (jarak + $1 \times$ blocker) keduanya memenuhi kriteria ini:

1. **Jarak** (*distance*) pasti \leq biaya sejetinya, karena setiap langkah unit minimal memerlukan satu gerakan.
2. **Blocking**: setiap kendaraan penghalang harus digeser sekurang-kurangnya sekali (H2) atau dua kali (H1 mengasumsikan mundur-maju), sehingga blockers $\times k$ tidak akan melebihi riil.

Karena $h(n)$ pada H1/H2 \leq biaya riil, A* dengan heuristik ini terjamin **optimal** (menemukan solusi terpendek) dan **complete** (menemukan solusi jika ada).

7.3 Apakah UCS sama dengan BFS di Rush Hour?

Secara teoritis, UCS dengan semua cost edge = 1 memang identik dengan BFS tradisional yang menelusuri level demi level. Namun dalam implementasi ini, kita memberi bobot $|\Delta|$ yang memperbolehkan satu gerakan menggeser beberapa sel sekaligus (cost > 1). Oleh karena

itu, UCS di sini **tidak persis sama** dengan BFS. Meski begitu, bila dikonfigurasi bahwa setiap gerakan unit (satu sel) menambah satu simpul, maka urutan eksplorasi UCS identik dengan BFS, dan panjang solusi yang dihasilkan juga akan sama.

7.4 Efisiensi A* dibandingkan UCS

Secara teoretis, A* dengan heuristik admissible selalu mengunjungi sedikitnya simpul dibanding UCS yang sama-sama optimal. UCS mengekskansi semua simpul dengan $g(n) \leq C^*$, di mana C^* adalah biaya optimal, tanpa memandang seberapa dekat simpul tersebut ke goal. Sebaliknya, A* hanya mengekskansi simpul dengan $f(n) = g(n) + h(n) \leq C^*$. Karena $h(n) \geq 0$, ruang pencarian A* menjadi subset ruang UCS, secara dramatis mengurangi jumlah simpul yang diperiksa. Dalam praktik Rush Hour, ini berakibat A* memeriksa jauh lebih sedikit node—khususnya pada papan yang kompleks—dan menjalankan lebih cepat, asalkan heuristik relatif informatif.

7.5 Optimalitas Greedy Best-First Search

GBFS memilih simpul berikutnya semata-mata berdasarkan $h(n)$, tanpa mempertimbangkan biaya sejauh ini. Akibatnya, GBFS **tidak menjamin** menemukan jalur terpendek. Ia dapat tersesat mengejar simpul yang estimasi heuristiknya rendah, namun sebenarnya memerlukan banyak langkah mundur atau memutar untuk mencapai goal. Pada puzzle Rush Hour, GBFS seringkali menemukan solusi cepat (sedikit simpul), tetapi jalur yang dihasilkan bisa lebih panjang daripada optimal—sering terlihat selisih 10–20% dari jumlah langkah minimal.

7.6 Kompleksitas Waktu dan Memori

- UCS
 - Waktu dan memori: $O(b^d)$, dengan b rata-rata banyak tetangga per state (branching factor) dan d kedalaman solusi optimal. Karena mengabaikan heuristik, UCS eksplorasi menyeluruh hingga kedalaman d .
- GBFS
 - Waktu dan memori: bergantung pada kualitas heuristik; terbaik $O(d)$ jika heuristik sangat tajam, terburuk $O(b^m)$ dengan m kedalaman simpul terakhir yang diekspansi (sering $> d$).
- A*
 - Waktu: $O(b^{d-\epsilon})$, di mana ϵ mencerminkan “efektivitas” heuristik ($0 \leq \epsilon \leq d$).
 - Memori: $O(b^d)$, karena menyimpan semua simpul frontier hingga kedalaman solusi.

7.7 Pembahasan Hasil Uji

Pada kasus uji “test7.txt” yang sama terlihat perbedaan yang signifikan antara ketiga algoritma. *UCS* mengekskansi 1.170 simpul dalam waktu sekitar 12 ms dan menghasilkan solusi 49 langkah. *GBFS*, meskipun sekilas “hemat biaya ekspansi awal”, justru mengunjung jauh lebih banyak simpul (3.866 nodes) dan butuh waktu sekitar 47 ms, serta jalurnya malah jauh lebih panjang (106 langkah) karena ia murni mengejar heuristik tanpa mempertimbangkan biaya sejauh ini.

Sementara itu, A^* mencapai keseimbangan yang terbaik: hanya 1.028 simpul yang diperiksa (lebih sedikit daripada *UCS*), waktu 17 ms, dan rute terpendek (46 langkah). Ini menegaskan bahwa A^* bukan saja lebih efisien dalam jumlah node dibanding *UCS* berkat heuristik admissible-nya, tetapi juga lebih optimal dan lebih cepat daripada *GBFS* pada puzzle Rush Hour ini.

Pada *test case* “test8.txt” dengan A^* dan tiga heuristik yang berbeda tampak bahwa **H2** (jarak + $1 \times \text{blocker}$) memberikan kinerja terbaik: meski semua varian memeriksa jumlah simpul yang hampir sama (≈ 2.687), H2 menuntaskan pencarian dalam 37 ms, lebih cepat daripada H1 (61 ms) dan H3 (99 ms). Hal ini mengindikasikan bahwa koefisien *blocking* yang lebih ringan pada H2 mengurangi *overhead* per perhitungan heuristik, sekaligus tetap cukup informatif untuk memotong sebagian besar ruang pencarian.

Sementara H1 ($2 \times \text{blocking}$) sedikit lebih lambat karena perhitungan penalti ganda, H3 meski paling “tajam” malah paling lambat karena kompleksitas tambahan cek sekunder dan kasus `Integer.MAX_VALUE` saat ‘X’ berada di jalur. Dengan demikian, H2 menyeimbangkan antara efektivitas pemangkasan simpul dan kecepatan eksekusi, menjadikannya pilihan praktis untuk A^* pada puzzle Rush Hour ini.

BAB VIII: Kesimpulan dan Saran

Setelah mengimplementasikan dan menguji tiga algoritma pencarian Uniform-Cost Search (UCS), Greedy Best-First Search (GBFS), dan A* pada puzzle *Rush Hour* dengan berbagai konfigurasi pintu keluar dan rintangan, dapat disimpulkan bahwa A* dengan heuristik “jarak + $1 \times \text{blocker}$ ” (H2) menawarkan kombinasi terbaik antara keoptimalan dan efisiensi. A* secara konsisten mengekskansi lebih sedikit simpul daripada UCS, sehingga mengurangi waktu eksekusi tanpa mengorbankan panjang solusi, sedangkan UCS meski selalu menemukan jalur terpendek, cenderung mengekskansi wilayah pencarian yang jauh lebih luas sehingga lebih lambat. GBFS, walaupun sangat cepat pada langkah-langkah awal karena hanya mengejar estimasi heuristik, tidak menjamin jalur optimal dan dalam beberapa kasus mengunjungi jauh lebih banyak simpul serta menghasilkan langkah solusi yang lebih panjang.

Untuk *development* selanjutnya, beberapa perbaikan dapat dipertimbangkan. Pertama, memperkaya metode heuristik menggunakan *pattern database* atau strategi *critical block* dapat meningkatkan ketajaman estimasi tanpa menambah overhead komputasi secara drastis. Kedua, teknik *pruning* seperti *symmetry reduction* dan *transposition tables* dapat mencegah eksplorasi konfigurasi papan yang identik akibat rotasi atau refleksi, sehingga meminimalkan duplikasi usaha.

Ketiga, eksperimen dengan pencarian dua arah (*bidirectional A**) mungkin menurunkan eksponensial ruang pencarian dengan bertemu di tengah. Keempat, pada antarmuka GUI, menambahkan fitur “Play” otomatisasi animasi gerakan dan ekspor hasil ke format GIF atau video akan meningkatkan pengalaman user (UX). Terakhir, implementasi lintas platform atau pemanfaatan *multithreading* dapat memberikan skalabilitas dan performa lebih tinggi pada puzzle dengan ukuran lebih besar atau mode heuristik paralel.

Lampiran

Link repository GitHub:

https://github.com/jovan196/Tucil3_13523141

Tabel Checklist:

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