

IF2211 Strategi Algoritma  
**Penyelesaian Puzzle Rush Hour Menggunakan Algoritma  
Pathfinding**

**Laporan Tugas Kecil 3**

Disusun untuk memenuhi tugas mata kuliah IF2211 Strategi Algoritma pada  
Semester 2 Tahun Akademik 2024/2025



Disusun oleh:

**13523135 - Ahmad Syafiq**

**13523162 - Fachriza Ahmad Setiyono**

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

**2024**

## DAFTAR ISI

DAFTAR ISI.....	2
<b>BAB I</b>	
DESKRIPSI MASALAH.....	4
<b>BAB II</b>	
TEORI SINGKAT.....	7
2.1. Algoritma Uniform Cost Search.....	7
2.2. Algoritma Greedy Best First Search.....	7
2.3. Algoritma A* Search.....	8
<b>BAB III</b>	
METODE PENYELESAIAN.....	10
3.1. Mapping Persoalan Rush Hour.....	10
3.1.1 Algoritma Uniform Cost Search.....	10
3.1.2 Algoritma Greedy Best First Search.....	11
3.1.3 Algoritma A* Search.....	11
3.2 Analisis Algoritma.....	12
3.2.1 Definisi $f(n)$ dan $g(n)$ .....	12
3.2.2 Kelayakan Heuristik Algoritma A*.....	12
3.2.3 Perbandingan UCS dan BFS dalam Puzzle Rush Hour.....	13
3.2.4 Perbandingan Efisiensi A* dan UCS.....	13
3.2.5 Optimalitas GBFS.....	14
<b>BAB IV</b>	
IMPLEMENTASI.....	15
4.1. Implementasi Program.....	15
4.2. Kode Sumber.....	15
4.2.1. Model.....	15
4.2.1.1. Car.java.....	15
4.2.1.2. Board.java.....	16
4.2.1.3. State.java.....	22
4.2.1.3. State.java.....	23
4.2.1.4. Utils.java.....	24
4.2.1.5. Algorithm.....	29
4.2.1.5.1. Algorithm.java.....	29
4.2.1.5.2. InformedSearch.java.....	29
4.2.1.5.3. Node.java.....	29
4.2.1.5.4. Astar.java.....	30
4.2.1.5.5. GreedyBestFirstSearch.java.....	31
4.2.1.5.6. UniformCostSearch.java.....	32
4.2.1.6. Heuristic.....	34
4.2.1.6.1. Heuristic.java.....	34

4.2.1.6.2. BlockingHeuristic.java.....	34
4.2.1.6.3. DistanceHeuristic.java.....	35
4.2.1.5.3. Astar.java.....	36
4.2.2. View.....	37
4.2.2.1. CLI.java.....	37
4.2.2.2. GUI.java.....	40
4.2.2.1. BoardView.java.....	49
4.2.3 Main.java.....	52
<b>BAB V</b>	
<b>IMPLEMENTASI BONUS.....</b>	<b>53</b>
5.1. Graphical User Interface.....	53
5.2. Heuristik Alternatif.....	55
<b>BAB VI</b>	
<b>EKSPERIMEN DAN ANALISIS.....</b>	<b>56</b>
6.1. Pengujian Program.....	56
6.2. Analisis Hasil.....	65
6.3. Analisis Kompleksitas Algoritma.....	65
6.3.1. Kompleksitas Algoritma UCS.....	65
6.3.2. Kompleksitas Algoritma GBFS.....	66
6.3.3. Kompleksitas Algoritma A*.....	66
<b>LAMPIRAN.....</b>	<b>67</b>
Lampiran Checklist Spesifikasi Program.....	67
Lampiran Repository Program.....	67

## BAB I

### DESKRIPSI MASALAH



Gambar 1.1. Permainan Rush Hour

(sumber: <https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

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

Komponen penting dalam permainan ini meliputi papan, piece, primary piece, pintu keluar, dan gerakan. Papan merupakan area permainan yang terdiri atas sejumlah cell, yaitu unit terkecil yang dapat ditempati kendaraan (piece). Setiap piece memiliki ukuran (biasanya 2 atau 3 cell), posisi awal, dan orientasi yang hanya bisa horizontal atau vertikal. Semua piece akan diletakkan pada papan sejak awal permainan sesuai konfigurasi tertentu. Primary piece adalah satu-satunya kendaraan yang harus dikeluarkan dari papan dan satu-satunya yang dapat melewati

pintu keluar, yang selalu terletak sejajar dengan orientasi primary piece. Piece lain hanya berfungsi sebagai penghalang yang harus dipindahkan untuk memberi jalan. Selama permainan, pemain hanya dapat menggeser piece sesuai orientasinya tanpa melompati atau menembus piece lain. Tantangan dari permainan ini terletak pada bagaimana menyusun urutan gerakan secara logis dan efisien agar primary piece dapat mencapai pintu keluar sesegera mungkin.

Pada tugas kecil kali ini, mahasiswa ditugaskan untuk membuat program sederhana dalam bahasa C/C++/Java/Javascript yang mengimplementasikan algoritma pathfinding Greedy Best First Search, Uniform Cost Search, dan A\* dalam menyelesaikan permainan Rush Hour. Algoritma pathfinding minimal menggunakan satu heuristik yang ditentukan sendiri dan dijalankan secara terpisah. Algoritma dan heuristik yang digunakan ditentukan berdasarkan input pengguna.

Alur program yang ditugaskan:

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).

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.
----------------------------

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

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

## **BAB II**

### **TEORI SINGKAT**

#### **2.1. Algoritma Uniform Cost Search**

Algoritma Uniform Cost Search (UCS\*) adalah salah satu algoritma pencarian berbasis graph traversal yang digunakan untuk menemukan jalur dengan biaya minimum dari titik awal (start node) ke titik tujuan (goal node) dalam suatu ruang pencarian. UCS termasuk dalam kategori algoritma uninformed search, karena tidak menggunakan informasi heuristik mengenai jarak ke tujuan. Fokus utama dari UCS adalah pada biaya total terkecil dari start node menuju suatu node tertentu, bukan pada kedekatan terhadap goal secara estimasi.

Algoritma UCS menggunakan fungsi evaluasi:

$$f(n) = g(n)$$

dengan  $g(n)$  adalah biaya total dari node awal ke simpul  $n$  saat ini.

Setiap iterasi, UCS akan memilih dan memperluas node dengan nilai  $f(n)$  terkecil, yaitu node dengan biaya kumulatif terendah dari start. Proses ini dilakukan dengan menggunakan priority queue yang selalu mengevaluasi node berbiaya terkecil lebih dahulu. Jika sebuah node bisa dicapai melalui jalur yang lebih murah dari jalur sebelumnya, maka jalur yang lebih mahal diabaikan.

UCS menjamin menemukan solusi optimal jika semua biaya langkah (path cost) adalah non-negatif. Oleh karena itu, algoritma ini cocok digunakan dalam permasalahan yang mengutamakan biaya minimum, seperti perencanaan rute, penjadwalan, dan permainan puzzle yang memiliki sistem bobot untuk setiap langkah. Namun, UCS cenderung lebih lambat dibanding algoritma heuristik seperti A\* jika tidak dibatasi dengan heuristik, karena UCS akan menjelajahi semua kemungkinan jalur dengan biaya terendah secara menyeluruh sebelum mempertimbangkan jalur lain.

#### **2.2. Algoritma Greedy Best First Search**

Greedy Best First Search (GBFS) adalah algoritma pencarian informed search yang memanfaatkan informasi heuristik untuk memandu proses pencarian menuju goal dengan cara yang dianggap paling cepat berdasarkan estimasi jarak. Pada setiap langkahnya, algoritma ini selalu memilih node yang memiliki nilai heuristik terkecil (paling "dekat" ke *goal* secara estimasi), tanpa mempertimbangkan biaya yang telah dikeluarkan untuk mencapai node tersebut.

Algoritma GBFS menggunakan fungsi evaluasi:

$$f(n) = h(n)$$

dengan  $h(n)$  adalah fungsi heuristik yang mengestimasi jarak dari simpul  $n$  ke tujuan.

Karena hanya menggunakan heuristik tanpa memperhitungkan cost dari start node ke node saat ini, GBFS bisa sangat cepat dalam menemukan solusi jika heuristiknya baik (informatif dan mendekati nilai sebenarnya). Namun, karena mengabaikan biaya sebelumnya, solusi yang dihasilkan tidak dijamin optimal, terutama jika ada jalur dengan total biaya lebih rendah yang dilewati karena memiliki nilai heuristik lebih tinggi.

Keunggulan dari GBFS terletak pada kecepatannya dalam menjelajahi ruang pencarian yang besar, karena fokusnya hanya pada arah menuju goal. Namun kelemahannya adalah bisa terjebak dalam optimum lokal atau mengikuti jalur yang terlihat menjanjikan secara heuristik tetapi sebenarnya mahal atau bahkan buntu. Oleh karena itu, efektivitas GBFS sangat bergantung pada kualitas fungsi heuristik yang digunakan.

Algoritma ini cocok untuk aplikasi yang memerlukan respons cepat dan tidak terlalu mementingkan optimalitas solusi, seperti navigasi awal, pengambilan keputusan sementara, atau sistem real-time dengan batasan waktu.

### 2.3. Algoritma A\* Search

A\* Search adalah algoritma pencarian informed search yang menggabungkan keunggulan dari Uniform Cost Search (UCS) dan Greedy Best First Search (GBFS). A\* menggunakan fungsi evaluasi yang mempertimbangkan baik biaya aktual dari start ke node saat ini maupun estimasi biaya dari node tersebut ke goal, sehingga mampu menemukan solusi dengan jalur paling murah secara optimal sekaligus efisien dalam proses pencariannya.

Algoritma A\* Search menggunakan fungsi evaluasi:

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

dengan  $g(n)$  adalah biaya dari start ke node  $n$  dan  $h(n)$  adalah fungsi heuristik yang mengestimasi jarak dari simpul  $n$  ke tujuan.

A\* Search akan selalu memilih node dengan nilai  $f(n)$  terendah untuk dieksplorasi berikutnya. Dengan cara ini, A\* dapat menyeimbangkan antara jalur yang murah sejauh ini dan jalur yang menjanjikan menuju tujuan. Jika heuristik  $h(n)$  yang digunakan admissible (tidak melebihi-lebihkan jarak ke goal) dan konsisten, maka A\* dijamin akan menemukan solusi yang optimal.

Keunggulan utama A\* adalah kemampuannya menemukan solusi terbaik dengan efisiensi yang lebih baik dibanding UCS, karena menggunakan heuristik untuk mempersempit ruang



pencarian. Namun,  $A^*$  tetap bisa menjadi mahal dari sisi memori dan waktu jika ruang pencariannya besar dan heuristiknya kurang akurat.

## **BAB III**

### **METODE PENYELESAIAN**

#### **3.1. Mapping Persoalan Rush Hour**

Persoalan Rush Hour dapat dipandang sebagai sebuah pencarian dalam graf ruang status, di mana setiap simpul merepresentasikan satu konfigurasi papan, dan setiap sisi merepresentasikan aksi legal berupa pergeseran satu kendaraan ke arah yang sesuai dengan orientasinya. Dengan memodelkan masalah ini sebagai pencarian jalur dalam graf ruang status, algoritma pencarian jalur seperti Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A\* Search dapat diterapkan untuk menemukan solusi.

Pencarian solusi dilakukan untuk mencapai status akhir, yaitu konfigurasi di mana mobil utama berhasil mencapai pintu keluar. Solusi yang diperoleh berupa rangkaian status yang membentuk lintasan dari status awal hingga status akhir, merepresentasikan urutan langkah-langkah yang perlu diambil untuk menyelesaikan permainan.

Pencarian solusi dilakukan dengan memproses status awal dan menghasilkan semua status turunan yang mungkin melalui langkah-langkah legal. Setiap status baru kemudian dimasukkan ke dalam priority queue, yang akan menentukan urutan eksplorasi berdasarkan nilai dari fungsi evaluasi tertentu. Priority queue ini memastikan bahwa status yang paling menjanjikan diproses lebih dahulu sesuai strategi pencarian yang digunakan. Adapun perhitungan fungsi evaluasi untuk setiap algoritma akan dijelaskan lebih lanjut.

##### **3.1.1 Algoritma Uniform Cost Search**

Penerapan algoritma Uniform Cost Search (UCS) pada persoalan Rush Hour dilakukan dengan menetapkan komponen-komponen UCS yang merepresentasikan elemen-elemen dalam permainan. Komponen utama yang harus ditentukan adalah fungsi evaluasi yang digunakan untuk menilai biaya total yang dibutuhkan untuk mencapai status saat ini dari status awal.

Dalam konteks Rush Hour, solusi optimal adalah solusi dengan jumlah langkah paling sedikit yang diperlukan untuk memindahkan mobil utama ke pintu keluar. Oleh karena itu, biaya (cost) yang digunakan dalam fungsi evaluasi UCS adalah jumlah langkah yang telah dilakukan dari status awal hingga status saat ini. Dengan demikian, fungsi evaluasi UCS dapat dituliskan sebagai:

$$f(n) = g(n)$$

dengan  $g(n)$  adalah banyaknya langkah dari status awal ke status  $n$ .

Karena semua langkah dalam Rush Hour memiliki biaya yang sama, maka algoritma UCS dalam hal ini akan berperilaku serupa dengan Breadth First Search (BFS)—yaitu mengeksplorasi semua status dengan jumlah langkah terkecil terlebih dahulu sebelum melanjutkan ke status dengan langkah lebih banyak.

### 3.1.2 Algoritma Greedy Best First Search

Penerapan algoritma Greedy Best First Search (GBFS) pada persoalan Rush Hour dilakukan dengan memanfaatkan fungsi heuristik untuk memperkirakan seberapa dekat suatu status terhadap status tujuan. Dalam GBFS, komponen utama yang perlu ditentukan adalah fungsi evaluasi yang mengestimasi jarak dari status saat ini ke status akhir.

Dalam konteks permainan Rush Hour, jarak dari status saat ini ke status akhir dapat diestimasi menggunakan fungsi heuristik yang mempertimbangkan beberapa parameter, seperti jumlah kendaraan yang menghalangi jalur mobil utama menuju pintu keluar, dan jarak mobil utama ke pintu keluar, dihitung dalam jumlah cell secara horizontal (karena orientasi mobil utama horizontal).

Berdasarkan dua pendekatan ini, terdapat dua versi algoritma Greedy Best First Search (GBFS) yang diimplementasikan. Versi pertama menggunakan heuristik berdasarkan jumlah kendaraan penghalang, sementara versi kedua menggunakan heuristik berdasarkan jarak langsung mobil utama ke pintu keluar. Dengan demikian, fungsi evaluasi GBFS dapat dituliskan sebagai:

$$f(n) = h(n)$$

dengan  $h(n)$  adalah jumlah kendaraan penghalang pada status  $n$  untuk versi pertama dan  $h(n)$  adalah jarak mobil utama ke pintu keluar pada status  $n$  untuk versi kedua.

### 3.1.3 Algoritma A\* Search

Penerapan algoritma A\* Search pada persoalan Rush Hour dilakukan dengan menggabungkan dua komponen evaluasi utama: biaya aktual dari status awal ke status saat ini seperti UCS dan estimasi jarak dari status saat ini ke status akhir seperti GBFS. Penjumlahan komponen evaluasi dilakukan tanpa pembobotan, sehingga fungsi evaluasi A\* Search dapat dituliskan sebagai:

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

dengan  $g(n)$  adalah banyaknya langkah dari status awal ke status  $n$  dan  $h(n)$  adalah jumlah kendaraan penghalang pada status  $n$  untuk versi pertama dan  $h(n)$  adalah jarak mobil utama ke pintu keluar pada status  $n$  untuk versi kedua.

## 3.2 Analisis Algoritma

### 3.2.1 Definisi $f(n)$ dan $g(n)$

Secara umum  $f(n)$  dapat didefinisikan sebagai fungsi evaluasi biaya memilih status  $n$ . Sedangkan  $g(n)$  didefinisikan sebagai fungsi yang merepresentasikan biaya total atau akumulasi dari status awal ke status  $n$ .

Untuk algoritma *uninformed search/blind search*, didefinisikan  $f(n) = g(n)$ . Sedangkan untuk algoritma *informed search*, didefinisikan  $f(n) = g(n) + h(n)$ , dimana  $h(n)$  adalah fungsi untuk mengestimasi biaya dari status  $n$  ke status tujuan menggunakan suatu heuristik. Setiap algoritma dapat memiliki implementasi fungsinya masing-masing.

- UCS:  $f(n) = g(n)$  dengan  $g(n)$  adalah jumlah langkah dari status awal ke status  $n$
- GBFS:  $f(n) = h(n)$  dengan  $h(n)$  adalah estimasi jarak ke tujuan berdasarkan jumlah kendaraan penghalang atau jarak mobil utama yang searah ke pintu keluar, sesuai pilihan heuristik.
- A\*:  $f(n) = g(n) + h(n)$  dengan  $g(n)$  adalah jumlah langkah dari status awal ke status  $n$  dan  $h(n)$  adalah estimasi jarak ke tujuan berdasarkan jumlah kendaraan penghalang atau jarak mobil utama yang searah ke pintu keluar, sesuai pilihan heuristik.

### 3.2.2 Kelayakan Heuristik Algoritma A\*

Agar algoritma A\* dapat menjamin solusi yang optimal dalam menyelesaikan persoalan Rush Hour, fungsi heuristik  $h(n)$  yang digunakan harus layak diterima (admissible), yaitu tidak melebih-lebihkan (tidak overestimate) biaya sebenarnya dari status saat ini ke status tujuan, atau  $h(n) \leq h^*(n)$ .

Dalam implementasi pada permainan Rush Hour yang telah dijelaskan sebelumnya, digunakan dua versi heuristik:

1. Jumlah kendaraan penghalang (blocking cars)

Heuristik ini menghitung banyaknya mobil yang menghalangi jalur langsung menuju pintu keluar tanpa memperkirakan langkah untuk menggerakkan mobil tersebut. Untuk setiap mobil yang menghalangi, diperlukan setidaknya satu langkah untuk menyingkirkannya sehingga nilai ini selalu kurang atau sama

dengan biaya minimum sebenarnya untuk mencapai pintu keluar, maka heuristik ini *admissible*.

## 2. Jarak mobil utama searah ke pintu keluar

Heuristik ini menghitung jarak yang memisahkan mobil utama dari pintu keluar. Karena pada spesifikasi tugas dijelaskan bahwa baik langkah panjang maupun langkah pendek dihitung sebagai satu langkah, maka heuristik ini tidak *admissible*. Namun demikian, jika setiap langkah hanya bisa menggeser mobil sebanyak satu petak, langkah ini *admissible* karena minimal diperlukan langkah sebanyak jarak menuju pintu keluar.

### 3.2.3 Perbandingan UCS dan BFS dalam Puzzle Rush Hour

Pada permasalahan puzzle Rush Hour, fungsi  $g(n)$  pada UCS didefinisikan sebagai banyaknya langkah dari status awal ke status  $n$ . Oleh karena itu, semua status yang ada di kedalaman yang sama akan memiliki nilai  $f(n)$  yang sama. Karena pemilihan status untuk pemrosesan selanjutnya menggunakan antrian prioritas pada nilai  $f(n)$ , maka status dijamin akan diproses dengan urutan yang sama dengan metode BFS, yaitu dengan urutan seluruh status pada kedalaman yang sama sebelum memproses seluruh status pada kedalaman selanjutnya.

### 3.2.4 Perbandingan Efisiensi A\* dan UCS

Metode UCS dipastikan dapat mengembalikan solusi yang optimal pada permasalahan Rush Hour, karena pada kasus ini UCS bekerja serupa dengan BFS yang sudah dibuktikan dapat menemukan solusi optimal. Solusi dianggap optimal jika solusi memiliki langkah penyelesaian terpendek.

Algoritma A\* juga dapat dipastikan mengembalikan solusi yang optimal pada permasalahan Rush Hour dengan syarat model heuristik yang digunakan bersifat *admissible*. Ini berarti, tidak ada perbedaan pada optimalitas solusi yang dihasilkan antara metode UCS dan A\* dengan heuristik *admissible*.

Walaupun menghasilkan solusi yang sama-sama optimal, UCS perlu menjelajahi semua status yang ada dari status awal hingga pada kedalaman status akhir optimal. Berbeda dengan A\* yang mampu memotong jumlah penjelajahan status dengan memprioritaskan status yang terlihat lebih menjanjikan, yang terlihat dari nilai  $f(n)$ .

Secara teori, algoritma A\* dipastikan dapat menemukan solusi yang optimal dengan lebih efisien daripada UCS, kecuali jika biaya *overhead* evaluasi heuristik jauh

lebih besar dibanding biaya penjelajahan status, misal pada kasus kondisi puzzle yang dapat langsung diselesaikan dalam satu kali gerakan.

### **3.2.5 Optimalitas GBFS**

Algoritma GBFS bekerja dengan memilih status yang menurut fungsi heuristik  $h(n)$  paling dekat dengan tujuan. Fokus utama GBFS adalah mencari jalur ke tujuan dengan waktu (komputasi) yang singkat, bukan memastikan bahwa jalur yang diambil adalah yang paling murah. Meskipun heuristik GBFS *admissible*, GBFS tidak memperhitungkan biaya aktual dari status awal ke status saat ini. Akibatnya, algoritma ini bisa memilih jalur yang terlihat dekat secara heuristik, tetapi justru lebih mahal secara nyata.

## BAB IV

### IMPLEMENTASI

#### 4.1. Implementasi Program

Program diimplementasikan dengan bahasa Java 21 di dalam sebuah paket bernama `com.syafiqriza.rushhoursolver`. Pengguna bisa menjalankan program melalui CLI maupun GUI. GUI diimplementasikan menggunakan pustaka JavaFX versi 21. Untuk membantu proses *build* program dengan GUI, digunakan kakas Gradle. Namun, skrip *build* manual juga disediakan jika kakas Gradle tidak tersedia. File jar *executable* juga disediakan dalam *repository*.

#### 4.2. Kode Sumber

##### 4.2.1. Model

##### 4.2.1.1. Car.java

```
package com.syafiqriza.rushhoursolver.model;

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

/**
 * Merepresentasikan sebuah mobil dalam puzzle Rush Hour.
 * Setiap mobil memiliki ID, posisi (baris, kolom), panjang, dan
 * orientasi (horizontal/vertikal).
 */
public class Car {
    private final char id; // ID unik dari mobil (contoh: "A", "B", "P")
    private final int length; // Jumlah sel yang ditempati oleh mobil
    private final boolean isHorizontal; // Orientasi mobil: true jika
    horizontal
    private int row, col; // Posisi kepala mobil (sel paling kiri/atas)

    /**
     * Konstruktor untuk membuat objek mobil baru.
     * @param id ID unik mobil
     * @param length Panjang mobil (jumlah sel yang ditempati)
     * @param isHorizontal Orientasi mobil (true jika horizontal)
     * @param row Baris awal (kepala mobil)
     * @param col Kolom awal (kepala mobil)
     */
    public Car(char id, int length, boolean isHorizontal, int row, int
    col) {
        this.id = id;
        this.length = length;
        this.isHorizontal = isHorizontal;
        this.row = row;
        this.col = col;
    }

    public char getId() { return id; }
```

```

public int getLength() { return length; }
public boolean isHorizontal() { return isHorizontal; }
public int getRow() { return row; }
public int getCol() { return col; }

/**
 * Menggerakkan mobil sejauh offset dalam arah orientasinya.
 * @param offset Jumlah sel untuk bergerak (positif atau negatif)
 */
public void move(int offset) {
    if (isHorizontal) col += offset;
    else row += offset;
}

/**
 * Membatalkan gerakan sebelumnya dengan arah yang berlawanan.
 * @param offset Offset yang sama dengan yang digunakan di move()
 */
public void undoMove(int offset) {
    move(-offset);
}

/**
 * Mendapatkan daftar semua sel yang ditempati oleh mobil ini.
 * @return List berisi pasangan [baris, kolom] untuk setiap sel yang
ditempati
 */
public List<int[]> getOccupiedCells() {
    List<int[]> cells = new ArrayList<>();
    for (int i = 0; i < length; i++) {
        cells.add(isHorizontal ? new int[]{row, col + i} : new
int[]{row + i, col});
    }
    return cells;
}

/**
 * Membuat salinan baru dari mobil ini.
 * @return Objek Car baru dengan nilai properti yang sama
 */
public Car copy() {
    return new Car(id, length, isHorizontal, row, col);
}
}

```

#### 4.2.1.2. Board.java

```

package com.syafiqriza.rushhoursolver.model;

import java.util.ArrayList;

```



```

import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Merepresentasikan papan permainan Rush Hour.
 * Papan terdiri dari grid ukuran rows x cols, sejumlah mobil (termasuk
 mobil utama),
 * dan posisi tujuan (goal) yang harus dicapai oleh mobil utama.
 */
public class Board {
    private final int rows;           // Jumlah baris pada papan
    private final int cols;           // Jumlah kolom pada papan
    private final char goalCarId;      // ID mobil utama (biasanya 'P')
    private final int goalRow;         // Baris tujuan yang harus dicapai
 mobil utama
    private final int goalCol;         // Kolom tujuan yang harus dicapai
 mobil utama

    private final String detail;

    private final Map<Character, Car> cars; // Map dari ID mobil ke
 objek Car
    private char[][] grid;              // Representasi visual grid papan
 saat ini

    /**
     * Konstruktor papan.
     * @param rows jumlah baris papan
     * @param cols jumlah kolom papan
     * @param cars map ID mobil ke objek Car
     * @param goalCarId ID dari mobil utama (yang harus mencapai goal)
     * @param goalRow baris tujuan
     * @param goalCol kolom tujuan
     */
    public Board(int rows, int cols, Map<Character, Car> cars,
        char goalCarId, int goalRow, int goalCol, String
 detail) {
        this.rows = rows;
        this.cols = cols;
        this.goalCarId = goalCarId;
        this.goalRow = goalRow;
        this.goalCol = goalCol;
        this.cars = cars;
        this.detail = detail;
        buildGrid();
    }

    /**
     * Membangun ulang grid dari posisi mobil yang ada.

```

```

    */
    private void buildGrid() {
        grid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            Arrays.fill(grid[i], '.'); // '.' berarti sel kosong
        }
        for (Car car : cars.values()) {
            char idChar = car.getId();
            for (int[] cell : car.getOccupiedCells()) {
                grid[cell[0]][cell[1]] = idChar;
            }
        }
    }

    /**
     * Mengecek apakah mobil dengan ID tertentu bisa bergerak sebanyak
     * offset.
     * @param id ID mobil yang ingin digerakkan
     * @param offset jarak perpindahan
     * @return true jika gerakan valid, false jika tidak
     */
    public boolean canMove(char id, int offset) {
        Car car = cars.get(id);
        Car temp = car.copy();
        temp.move(offset);

        for (int[] cell : temp.getOccupiedCells()) {
            int r = cell[0], c = cell[1];
            if (r < 0 || r >= rows || c < 0 || c >= cols) return false;
            if (grid[r][c] != '.' && grid[r][c] != id) return false;
        }
        return true;
    }

    /**
     * Menerapkan gerakan mobil ke grid.
     * @param id ID mobil
     * @param offset langkah gerakan
     */
    public void applyMove(char id, int offset) {
        cars.get(id).move(offset);
        buildGrid();
    }

    /**
     * Membatalkan gerakan mobil.
     * @param id ID mobil
     * @param offset offset yang ingin dibatalkan
     */
    public void undoMove(char id, int offset) {
        cars.get(id).undoMove(offset);
    }

```

```

        buildGrid();
    }

    /**
     * Mengecek apakah mobil utama telah mencapai posisi goal.
     * @return true jika posisi tujuan tercapai
     */
    public boolean isSolved() {
        Car car = cars.get(goalCarId);
        for (int[] cell : car.getOccupiedCells()) {
            if(car.isHorizontal()) {
                cell[1] += goalCol < cell[1] ? -1 : 1;
            } else {
                cell[0] += goalRow < cell[0] ? -1 : 1;
            }
            if (cell[0] == goalRow && cell[1] == goalCol) {
                return true;
            }
        }
        return false;
    }

    /**
     * Mengembalikan salinan dari semua mobil pada papan.
     * @return map ID ke salinan Car
     */
    public Map<Character, Car> getCarsCopy() {
        Map<Character, Car> copy = new HashMap<>();
        for (Map.Entry<Character, Car> entry : cars.entrySet()) {
            copy.put(entry.getKey(), entry.getValue().copy());
        }
        return copy;
    }

    /**
     * Format ANSI untuk simbol Board
     * @param carID
     * @return string berisi ID kendaraan dengan warna
     */
    private String getFormattedBoardSymbol(char carID) {
        StringBuilder formatted = new StringBuilder();
        formatted.append("\u001B[");
        if(carID == getDetail().charAt(0)) {
            formatted.append("1;"); // bold
        }
        if (carID == 'P') {
            formatted.append("31"); // hijau
        } else if (carID == 'K') {
            formatted.append("32"); // hijau
        } else if (carID == getDetail().charAt(0)) {

```

```

        formatted.append("34"); // biru
    } else {
        formatted.append("0");
    }

    formatted.append("m").append(carID).append("\u001B[0m");
    return formatted.toString();
}

/**
 * Menampilkan papan ke console.
 */
public void printBoard() {
    var grid = getGrid();
    if(goalRow == -1) {
        for(int j = 0; j < cols; j++) {
            String symbol = getFormattedBoardSymbol(goalCol == j ?
'K' : ' ');
            System.out.print(symbol + " ");
        }
        System.out.println();
    }
    for (int i = 0; i < rows; i++) {
        if(goalCol == -1) {
            String symbol = getFormattedBoardSymbol(goalRow == i ?
'K' : ' ');
            System.out.print(symbol + " ");
        }
        for (int j = 0; j < cols; j++) {
            String symbol = getFormattedBoardSymbol(grid[i][j]);
            System.out.print(symbol + " ");
        }
        if(goalCol == cols) {
            String symbol = getFormattedBoardSymbol(goalRow == i ?
'K' : ' ');
            System.out.print(symbol + " ");
        }
        System.out.println();
    }
    if(goalRow == rows) {
        for(int j = 0; j < cols; j++) {
            String symbol = getFormattedBoardSymbol(goalCol == j ?
'K' : ' ');
            System.out.print(symbol + " ");
        }
        System.out.println();
    }
}

/**
 * Mendapatkan semua kemungkinan gerakan Car dari Board saat ini.

```

```

    * @return Array berisi kemungkinan kondisi Board selanjutnya
    */
    public Board[] getAllPossibleMovement() {
        List<Board> possibleBoards = new ArrayList<>();

        for (char carId : cars.keySet()) {
            Car car = cars.get(carId);

            // coba gerakan ke arah negatif (mundur)
            int offset = -1;
            while (canMove(carId, offset)) {
                String movement = carId + (car.isHorizontal() ? "-kiri"
: "-atas");
                Board newBoard = new Board(rows, cols, getCarsCopy(),
goalCarId, goalRow, goalCol, movement);
                newBoard.applyMove(carId, offset);
                possibleBoards.add(newBoard);
                offset--; // coba gerakan lebih jauh
            }

            // coba gerakan ke arah positif (maju)
            offset = 1;
            while (canMove(carId, offset)) {
                String movement = carId + (car.isHorizontal() ? "-kanan"
: "-bawah");
                Board newBoard = new Board(rows, cols, getCarsCopy(),
goalCarId, goalRow, goalCol, movement);
                newBoard.applyMove(carId, offset);
                possibleBoards.add(newBoard);
                offset++; // coba gerakan lebih jauh
            }
        }

        return possibleBoards.toArray(new Board[0]);
    }

    // Getter standar
    public int getRows() { return rows; }
    public int getCols() { return cols; }
    public Map<Character, Car> getCars() { return cars; }
    public char[][] getGrid() { return grid; }
    public char getGoalCarId() { return goalCarId; }
    public int getGoalRow() { return goalRow; }
    public int getGoalCol() { return goalCol; }
    public int getGoalRowClamped() {
        return Math.max(0, Math.min(goalRow, rows - 1));
    }
    public int getGoalColClamped() {
        return Math.max(0, Math.min(goalCol, cols - 1));
    }
    public String getDetail() { return detail; }

```

```
}
```

#### 4.2.1.3. State.java

```
package com.syafiqgriza.rushhoursolver.model;

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

public class State {
    private final int cumCost;           // g(n)
    private final int estimatedCost;     // h(n)

    private final Board board;
    private final List<String> carStates;

    public State(Board board, int cumCost, int estimatedCost) {
        this.board = board;
        this.cumCost = cumCost;
        this.estimatedCost = estimatedCost;

        carStates = new ArrayList<>();
        for (char carID : board.getCars().keySet()) {
            Car car = board.getCars().get(carID);
            carStates.add(carID + "-" + car.getRow() + "-" +
car.getCol() + "-" + (car.isHorizontal() ? 1 : 0));
        }
        Collections.sort(carStates); // normalize order
    }

    public Board getBoard() {
        return board;
    }

    public int getCumulativeCost() {
        return cumCost;
    }

    public int getEstimatedCost() {
        return estimatedCost;
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof State)) return false;
        State other = (State) obj;
        return carStates.equals(other.carStates);
    }
}
```

```

    }

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

```

#### 4.2.1.3. State.java

```

package com.syafiqriza.rushhoursolver.model;

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

public class State {
    private final int cumCost;           //  $g(n)$ 
    private final int estimatedCost;     //  $h(n)$ 

    private final Board board;
    private final List<String> carStates;

    public State(Board board, int cumCost, int estimatedCost) {
        this.board = board;
        this.cumCost = cumCost;
        this.estimatedCost = estimatedCost;

        carStates = new ArrayList<>();
        for (char carID : board.getCars().keySet()) {
            Car car = board.getCars().get(carID);
            carStates.add(carID + "-" + car.getRow() + "-" +
car.getCol() + "-" + (car.isHorizontal() ? 1 : 0));
        }
        Collections.sort(carStates); // normalize order
    }

    public Board getBoard() {
        return board;
    }

    public int getCumulativeCost() {
        return cumCost;
    }

    public int getEstimatedCost() {
        return estimatedCost;
    }

    public State copy() {

```

```

        return new State(
            new Board(
                board.getRows(),
                board.getCols(),
                board.getCarsCopy(),
                board.getGoalCarId(),
                board.getGoalRow(),
                board.getGoalCol(),
                board.getDetail()
            ),
            cumCost,
            estimatedCost
        );
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof State)) return false;
        State other = (State) obj;
        return carStates.equals(other.carStates);
    }

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

    public void printState() {
        for (String string : carStates) {
            System.out.println(string);
        }
    }
}

```

#### 4.2.1.4. Utils.java

```

package com.syafiqriza.rushhoursolver.model;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

public class Utils {
    public static Board readRushHourPuzzleFromFile(String fileName)
        throws FileNotFoundException {
        Scanner sc = new Scanner(new File(fileName));

        if (!sc.hasNextLine())
            throw new IllegalArgumentException("Format file tidak valid!
            File kosong.");

        String[] dimLine = sc.nextLine().trim().split("\\s+");
    }
}

```



```

        if (dimLine.length != 2)
            throw new IllegalArgumentException("Format file tidak valid!
Baris pertama harus memiliki 2 nilai: rows dan cols. Diberikan: " +
Arrays.toString(dimLine));

        int rows, cols;
        try {
            rows = Integer.parseInt(dimLine[0]);
            cols = Integer.parseInt(dimLine[1]);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Format file tidak valid!
Baris pertama harus berupa angka. Diberikan: " +
Arrays.toString(dimLine));
        }

        if (!sc.hasNextLine())
            throw new IllegalArgumentException("Format file tidak valid!
Baris kedua (jumlah kendaraan) tidak ditemukan.");

        String jumlahLine = sc.nextLine().trim();
        int jumlahNonP;
        try {
            jumlahNonP = Integer.parseInt(jumlahLine);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Format file tidak valid!
Baris kedua harus berupa angka. Diberikan: " + jumlahLine);
        }

        List<String> gridLines = new ArrayList<>();
        while (sc.hasNextLine()) {
            String line = sc.nextLine().stripTrailing();
            if (!line.isEmpty()) gridLines.add(line);
        }

        if (gridLines.size() < rows)
            throw new IllegalArgumentException("Format file tidak valid!
Expected " + rows + " baris papan, diberikan: " + gridLines.size());

        boolean hasExtraKLine = gridLines.size() == rows + 1;
        int goalRow = -2, goalCol = -2;

        Map<Character, List<int[]>> charToCells = new HashMap<>();
        Set<String> occupied = new HashSet<>();

        boolean leftK = false;

        for (int i = 0, finali = 0; i < rows; i++, finali++) {
            String rowLine = gridLines.get(finali);
            if (rowLine.length() > cols + 1)
                throw new IllegalArgumentException("Format file tidak
valid! Baris ke-" + i + " terlalu panjang. Expected max " + (cols + 1) +

```

```

", diberikan: " + rowLine.length());

    if(hasExtraKLine) {
        if(i == 0) {
            int idx = rowLine.indexOf('K');
            if (idx != -1) {
                goalRow = -1;
                goalCol = idx;
                finali++;
                rowLine = gridLines.get(finali);
            }
        }
    }

    for (int j = 0; j < rowLine.length(); j++) {
        char ch = rowLine.charAt(j);
        if(!hasExtraKLine && j == 0 && ch == 'K') {
            leftK = true;
            if (goalRow != -2)
                throw new IllegalArgumentException("Format file
tidak valid! Duplikat 'K' ditemukan.");
            goalRow = i;
            goalCol = j-1;
            continue;
        } else if (!hasExtraKLine && j == 0 && ch == ' ') {
            leftK = true;
            continue;
        } else if (ch == '.') {
            continue;
        }

        int finalj = leftK ? j-1 : j;
        if (finalj >= cols) {
            if (ch != 'K')
                throw new IllegalArgumentException("Format file
tidak valid! Karakter di luar grid hanya boleh 'K'. Diberikan: '" + ch +
"'" + finalj);
            if (goalRow != -2)
                throw new IllegalArgumentException("Format file
tidak valid! Duplikat 'K' ditemukan.");
            goalRow = i;
            goalCol = finalj;
            continue;
        }

        String key = i + "," + finalj;
        System.out.println(ch+key);

        if(ch == 'K') {
            throw new IllegalArgumentException("Format file

```

```

tidak valid! Mobil tidak boleh memiliki simbol K");
    }

    if (occupied.contains(key))
        throw new IllegalArgumentException("Format file
tidak valid! Tumpang tindih kendaraan di posisi (" + i + "," + finalj +
")");

    occupied.add(key);
    charToCells.computeIfAbsent(ch, k -> new
ArrayList<>()).add(new int[]{i, finalj});
    }
}

if (goalRow == -2 && hasExtraKLine) {
    String extra = gridLines.get(rows);
    int idx = extra.indexOf('K');
    if ((idx == -1 && goalRow == -2) || (idx != -1 && goalRow !=
-2))
        throw new IllegalArgumentException("Format file tidak
valid! Baris tambahan tidak valid atau duplikat 'K'.");
    goalRow = rows;
    goalCol = idx;
}

if (goalRow == -2 || goalCol == -2)
    throw new IllegalArgumentException("Format file tidak valid!
Posisi goal 'K' tidak ditemukan.");

Map<Character, Car> cars = new HashMap<>();
char goalCarId = 'P';

for (Map.Entry<Character, List<int[]>> entry :
charToCells.entrySet()) {
    char id = entry.getKey();
    List<int[]> posList = entry.getValue();
    if (posList.size() < 2)
        throw new IllegalArgumentException("Format file tidak
valid! Mobil '" + id + "' panjangnya kurang dari 2. Diberikan: " +
posList.size());

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

    int[] head = posList.get(0);
    int[] second = posList.get(1);

    boolean isHorizontal = head[0] == second[0];
    if (!isHorizontal && head[1] != second[1])
        throw new IllegalArgumentException("Format file tidak

```

```

valid! Mobil '"' + id + '"' tidak lurus horizontal atau vertikal.");

        // cek gap
        if (isHorizontal) {
            for (int i = 1; i < posList.size(); i++) {
                if (posList.get(i)[1] != posList.get(i - 1)[1] + 1
|| posList.get(i)[0] != posList.get(i-1)[0]) {
                    throw new IllegalArgumentException("Format file
tidak valid! Mobil '"' + id + '"' memiliki celah pada kolom.");
                }
            }
        } else {
            for (int i = 1; i < posList.size(); i++) {
                if (posList.get(i)[0] != posList.get(i - 1)[0] + 1
|| posList.get(i)[1] != posList.get(i-1)[1]) {
                    throw new IllegalArgumentException("Format file
tidak valid! Mobil '"' + id + '"' memiliki celah pada baris.");
                }
            }
        }

        cars.put(id, new Car(id, posList.size(), isHorizontal,
head[0], head[1]));
    }

    if(jumlahNonP != cars.size() - 1) {
        throw new IllegalArgumentException("Format file tidak valid!
Jumlah mobil non-utama berjumlah " + (cars.size() - 1) + " dari " +
jumlahNonP);
    }

    if (!cars.containsKey(goalCarId))
        throw new IllegalArgumentException("Format file tidak valid!
Mobil utama 'P' tidak ditemukan.");

    Car goalCar = cars.get(goalCarId);
    if ((goalCar.isHorizontal() && goalCar.getRow() != goalRow) ||
        (!goalCar.isHorizontal() && goalCar.getCol() !=
goalCol)) {
        throw new IllegalArgumentException("Format file tidak valid!
Posisi goal (K) tidak searah dengan mobil 'P'. Diharapkan baris: " +
goalCar.getRow() + " atau kolom: " + goalCar.getCol() + ", diberikan: ("
+ goalRow + "," + goalCol + ")");
    }

    return new Board(rows, cols, cars, goalCarId, goalRow, goalCol,
"Start state");
}
}

```

#### 4.2.1.5. Algorithm

##### 4.2.1.5.1. Algorithm.java

```
package com.syafiqriza.rushhoursolver.model.algorithm;

import com.syafiqriza.rushhoursolver.model.State;

public abstract class Algorithm {

    public class SolutionData {
        public int nodeCount;
        public State[] states = null;
        public double timeElapsedMs;
    }

    SolutionData solutionData = new SolutionData();

    public abstract void solve(State initialState);

    public SolutionData getSolution() {
        return this.solutionData;
    }
}
```

##### 4.2.1.5.2. InformedSearch.java

```
package com.syafiqriza.rushhoursolver.model.algorithm;

import com.syafiqriza.rushhoursolver.model.heuristic.Heuristic;

public abstract class InformedSearch extends Algorithm {
    protected Heuristic heuristicModel = null;

    public void setHeuristicModel(Heuristic heuristicModel) {
        this.heuristicModel = heuristicModel;
    }
}
```

##### 4.2.1.5.3. Node.java

```
package com.syafiqriza.rushhoursolver.model.algorithm;

import com.syafiqriza.rushhoursolver.model.*;

public class Node {

    private final State state;
    private final int depth;
    private final Node parent;

    public Node(State state, int depth, Node parent) {
        this.state = state;
    }
}
```

```

        this.depth = depth;
        this.parent = parent;
    }

    public State getState() { return state; }

    public Node getParent() { return parent; }

    public int getDepth() { return depth; }
}

```

#### 4.2.1.5.4. Astar.java

```

package com.syafiqriza.rushhoursolver.model.algorithm;

import java.util.Comparator;
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Set;

import com.syafiqriza.rushhoursolver.model.Board;
import com.syafiqriza.rushhoursolver.model.State;

public class AStar extends InformedSearch {

    public AStar() {}

    @Override
    public void solve(State initialState) {
        Set<State> visited = new HashSet<>();
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(n ->
n.getState().getEstimatedCost() + n.getState().getCumulativeCost()));

        long startTime = System.nanoTime();
        queue.add(new Node(initialState, 0, null));

        while(!queue.isEmpty()) {
            Node currentNode = queue.poll();
            State currentState = currentNode.getState();

            solutionData.nodeCount++;

            // berhenti jika board sudah solved
            if(currentState.getBoard().isSolved()) {
                // resize array solusi
                this.solutionData.states = new
State[currentNode.getDepth() + 1];

                Node pathNode = currentNode;

                // isi array solusi dari indeks akhir ke awal menuju

```

```

root node
        while (pathNode != null) {
            this.solutionData.states[pathNode.getDepth()] =
pathNode.getState();
            pathNode = pathNode.getParent();
        }
        break;
    }

    // hanya proses state jika belum visited
    if (!visited.contains(currentState)) {
        visited.add(currentState);

        // enqueue semua possible state
        for (Board board :
currentState.getBoard().getAllPossibleMovement()) {
            State s = new State(board,
currentState.getCumulativeCost() + 1, heuristicModel.getValue(board));
            queue.add(new Node(s, currentNode.getDepth() + 1,
currentNode));
        }
    }

    long timeElapsedNs = System.nanoTime() - startTime;
    double timeElapsedMs = timeElapsedNs / 1_000_000.0;
    solutionData.timeElapsedMs = timeElapsedMs;
}
}

```

#### 4.2.1.5.5. GreedyBestFirstSearch.java

```

package com.syafiqrizarushhoursolver.model.algorithm;

import com.syafiqrizarushhoursolver.model.State;
import com.syafiqrizarushhoursolver.model.Board;

import java.util.Comparator;
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Set;

public class GreedyBestFirstSearch extends InformedSearch {

    public GreedyBestFirstSearch() {}

    @Override
    public void solve(State initialState) {
        Set<State> visited = new HashSet<>();
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(n ->
n.getState().getEstimatedCost()));
    }
}

```

```

        long startTime = System.nanoTime();
        queue.add(new Node(initialState, 0, null));

        while(!queue.isEmpty()) {
            Node currentNode = queue.poll();
            State currentState = currentNode.getState();

            solutionData.nodeCount++;

            // berhenti jika board sudah solved
            if(currentState.getBoard().isSolved()) {
                // resize array solusi
                this.solutionData.states = new
State[currentNode.getDepth() + 1];

                Node pathNode = currentNode;

                // isi array solusi dari indeks akhir ke awal menuju
root node
                while (pathNode != null) {
                    this.solutionData.states[pathNode.getDepth()] =
pathNode.getState();
                    pathNode = pathNode.getParent();
                }
                break;
            }

            // hanya proses state jika belum visited
            if (!visited.contains(currentState)) {
                visited.add(currentState);

                // enqueue semua possible state
                for(Board board :
currentState.getBoard().getAllPossibleMovement()) {
                    State s = new State(board,
currentState.getCumulativeCost() + 1, heuristicModel.getValue(board));
                    queue.add(new Node(s, currentNode.getDepth() + 1,
currentNode));
                }
            }
        }

        long timeElapsedNs = System.nanoTime() - startTime;
        double timeElapsedMs = timeElapsedNs / 1_000_000.0;
        solutionData.timeElapsedMs = timeElapsedMs;
    }
}

```

#### 4.2.1.5.6. UniformCostSearch.java

```

package com.syafigriza.rushhoursolver.model.algorithm;

```



```

import com.syafigriza.rushhoursolver.model.State;
import com.syafigriza.rushhoursolver.model.Board;

import java.util.Comparator;
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Set;

public class UniformCostSearch extends Algorithm {

    public UniformCostSearch() {}

    @Override
    public void solve(State initialState) {
        Set<State> visited = new HashSet<>();
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(n ->
n.getState().getCumulativeCost()));

        long startTime = System.nanoTime();
        queue.add(new Node(initialState, 0, null));

        while(!queue.isEmpty()) {
            Node currentNode = queue.poll();
            State currentState = currentNode.getState();

            solutionData.nodeCount++;

            // berhenti jika board sudah solved
            if(currentState.getBoard().isSolved()) {
                // resize array solusi
                this.solutionData.states = new
State[currentNode.getDepth() + 1];

                Node pathNode = currentNode;

                // isi array solusi dari indeks akhir ke awal menuju
root node
                while (pathNode != null) {
                    this.solutionData.states[pathNode.getDepth()] =
pathNode.getState();
                    pathNode = pathNode.getParent();
                }
                break;
            }

            // hanya proses state jika belum visited
            if (!visited.contains(currentState)) {
                visited.add(currentState);
            }
        }
    }
}

```

```

        // enqueue semua possible state
        for(Board board :
currentState.getBoard().getAllPossibleMovement()) {
            State s = new State(board,
currentState.getCumulativeCost() + 1, currentState.getCumulativeCost() +
1);
            queue.add(new Node(s, currentNode.getDepth() + 1,
currentNode));
        }
    }
}

long timeElapsedNs = System.nanoTime() - startTime;
double timeElapsedMs = timeElapsedNs / 1_000_000.0;
solutionData.timeElapsedMs = timeElapsedMs;
}
}

```

#### 4.2.1.6. Heuristic

##### 4.2.1.6.1. Heuristic.java

```

package com.syafiqriza.rushhoursolver.model.heuristic;

import com.syafiqriza.rushhoursolver.model.Board;

public abstract class Heuristic {
    public abstract int getValue(Board board);
}

```

##### 4.2.1.6.2. BlockingHeuristic.java

```

package com.syafiqriza.rushhoursolver.model.heuristic;

import com.syafiqriza.rushhoursolver.model.Board;
import com.syafiqriza.rushhoursolver.model.Car;

public class BlockingHeuristic extends Heuristic {

    public BlockingHeuristic() {}

    @Override
    public int getValue(Board board) {
        if(board.isSolved()) {
            return 0;
        }

        Car goalCar = board.getCars().get(board.getGoalCarId());
        int dirToFinish;
        if(goalCar.isHorizontal()) {
            if(board.getGoalCol() > goalCar.getCol()) {
                dirToFinish = 1;
            } else {

```

```

        dirToFinish = -1;
    }
    else {
        if(board.getGoalRow() > goalCar.getRow()) {
            dirToFinish = 1;
        } else {
            dirToFinish = -1;
        }
    }

    char[][] grid = board.getGrid();

    int blockingCarCounter = 0;
    if(goalCar.isHorizontal()) {
        int carRow = goalCar.getRow();
        for(int j = goalCar.getCol() + dirToFinish; j !=
board.getGoalCol(); j += dirToFinish) {
            char carId = grid[carRow][j];
            if(carId != 'P' && carId != '.') {
                blockingCarCounter++;
            }
        }
    } else {
        int carCol = goalCar.getCol();
        for(int i = goalCar.getRow() + dirToFinish; i !=
board.getGoalRow(); i += dirToFinish) {
            char carId = grid[i][carCol];
            if(carId != 'P' && carId != '.') {
                blockingCarCounter++;
            }
        }
    }

    return blockingCarCounter;
}
}

```

#### 4.2.1.6.3. DistanceHeuristic.java

```

package com.syafiqrizal.rushhoursolver.model.heuristic;

import com.syafiqrizal.rushhoursolver.model.Board;
import com.syafiqrizal.rushhoursolver.model.Car;

public class DistanceHeuristic extends Heuristic {

    public DistanceHeuristic() {}

    @Override

```

```

public int getValue(Board board) {
    if(board.isSolved()) {
        return 0;
    }

    Car goalCar = board.getCars().get(board.getGoalCarId());
    int dirToFinish;
    if(goalCar.isHorizontal()) {
        if(board.getGoalCol() > goalCar.getCol()) {
            dirToFinish = 1;
        } else {
            dirToFinish = -1;
        }
    } else {
        if(board.getGoalRow() > goalCar.getRow()) {
            dirToFinish = 1;
        } else {
            dirToFinish = -1;
        }
    }

    char[][] grid = board.getGrid();

    int distance = 0;
    if(goalCar.isHorizontal()) {
        int carRow = goalCar.getRow();
        for(int j = goalCar.getCol() + dirToFinish; j !=
board.getGoalCol(); j += dirToFinish) {
            char carId = grid[carRow][j];
            if(carId != 'P') {
                distance++;
            }
        }
    } else {
        int carCol = goalCar.getCol();
        for(int i = goalCar.getRow() + dirToFinish; i !=
board.getGoalRow(); i += dirToFinish) {
            char carId = grid[i][carCol];
            if(carId != 'P') {
                distance++;
            }
        }
    }

    return distance;
}
}

```

#### 4.2.1.5.3. Astar.java

## 4.2.2. View

### 4.2.2.1. CLI.java

```
package com.syafiqriza.rushhoursolver.view;

import java.io.IOException;
import java.util.Scanner;

import com.syafiqriza.rushhoursolver.model.Board;
import com.syafiqriza.rushhoursolver.model.State;
import com.syafiqriza.rushhoursolver.model.Utills;
import com.syafiqriza.rushhoursolver.model.algorithm.*;
import com.syafiqriza.rushhoursolver.model.heuristic.*;

public class CLI {
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(System.in)) {
            String filePath;

            if (args.length == 0) {
                System.out.print("Masukkan path ke file puzzle (.txt): ");

                filePath = sc.nextLine();
            } else {
                filePath = args[0];
            }

            Board board;
            try {
                board = Utills.readRushHourPuzzleFromFile(filePath);
                board.printBoard();
            } catch (IOException e) {
                System.err.println("File tidak ditemukan: " + e.getMessage());
                return;
            } catch (IllegalArgumentException e) {
                System.err.println("Puzzle tidak valid: " + e.getMessage());
                return;
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
                return;
            }

            System.out.println();

            int chosenAlgorithm = 0;
            System.out.println("""
                Pilih algoritma:
                1. Uniform Cost Search
                2. Greedy Best First Search
                3. A*
            """);
        }
    }
}
```

```

        """);

while (true) {
    System.out.print(">>");
    if (!sc.hasNextInt()) {
        System.out.println("Masukkan pilihan angka yang
valid");

        sc.nextLine();
        continue;
    }
    chosenAlgorithm = sc.nextInt();
    if (chosenAlgorithm > 0 && chosenAlgorithm < 4) {
        break;
    }
    System.out.println("Masukkan pilihan angka yang valid");
}

Algorithm algorithm = null;
switch (chosenAlgorithm) {
    case 1:
        algorithm = new UniformCostSearch();
        break;
    case 2:
        algorithm = new GreedyBestFirstSearch();
        break;
    case 3:
        algorithm = new AStar();
        break;

    default:
        assert false;
}

if (algorithm instanceof InformedSearch alg) {
    int chosenHeuristic = 0;
    System.out.println("""
        Pilih heuristik:
        1. Blocking Cars
        2. Distance to Goal
        """);

    while (true) {
        System.out.print(">>");
        if (!sc.hasNextInt()) {
            System.out.println("Masukkan pilihan angka yang
valid");

            sc.nextLine();
            continue;
        }
        chosenHeuristic = sc.nextInt();
        if (chosenHeuristic > 0 && chosenHeuristic < 3) {

```

```

        break;
    }

    System.out.println("Masukkan pilihan angka yang
valid");
}

Heuristic heuristic = null;
switch (chosenHeuristic) {
    case 1:
        heuristic = new BlockingHeuristic();
        break;
    case 2:
        heuristic = new DistanceHeuristic();
        break;

    default:
        assert false;
}
alg.setHeuristicModel(heuristic);
}

algorithm.solve(new State(board, 0, 0));

Algorithm.SolutionData solutionData =
algorithm.getSolution();

System.out.println();

if (solutionData.states != null) {
    System.out.println("Solusi: ");
    for (State state : solutionData.states) {
        System.out.println(state.getBoard().getDetail());
        state.getBoard().printBoard();
        System.out.println();
    }

    System.out.println("Solusi ditemukan");
} else {
    System.out.println("Solusi tidak ditemukan");
}
System.out.println();

System.out.println("Waktu (ms) : " +
solutionData.timeElapsedMs);
System.out.println("Node dikunjungi : " +
solutionData.nodeCount);

if (solutionData.states != null)
    System.out.println("Jumlah langkah : " +
solutionData.states.length);

```

```

        System.out.println();
    }
}

```

#### 4.2.2.2. GUI.java

```

package com.syafiqriza.rushhoursolver.view;
import com.syafiqriza.rushhoursolver.model.Board;
import com.syafiqriza.rushhoursolver.model.Utills;
import com.syafiqriza.rushhoursolver.model.algorithm.*;
import com.syafiqriza.rushhoursolver.model.heuristic.BlockingHeuristic;
import com.syafiqriza.rushhoursolver.model.heuristic.DistanceHeuristic;
import com.syafiqriza.rushhoursolver.model.heuristic.Heuristic;
import javafx.animation.FadeTransition;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.concurrent.Task;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.image.Image;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.Stop;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.FileChooser;
import javafx.stage.Stage;
import javafx.util.Duration;

import java.io.File;
import java.io.IOException;

public class GUI extends Application {
    private Board board;
    private Algorithm algorithm;
    private Font customFont;
    private Text resultText;
    private Text errorMessageText;
    private VBox boardViewContainer;
    private ComboBox<String> heuristicSelector;

    private Heuristic getSelectedHeuristic() {
        String selected =

```



```

    heuristicSelector.getSelectionModel().getSelectedItem();
    return switch (selected) {
        case "Distance" -> new DistanceHeuristic();
        case "Blocking" -> new BlockingHeuristic();
        default -> new BlockingHeuristic();
    };
}

@Override
public void start(Stage stage) {
    customFont =
Font.loadFont(getClass().getResourceAsStream("/RacingEngine-Regular.otf"
), 16);

    BorderPane root = new BorderPane();
    root.setBackground(new Background(new BackgroundFill(
        new LinearGradient(0, 0, 1, 1, true,
CycleMethod.NO_CYCLE,
        new Stop(0, Color.web("#0f0f0f")), new Stop(1,
Color.web("#1c1c1c"))
    ), CornerRadii.EMPTY, Insets.EMPTY)));

    VBox titleBox = createTitleBox();
    root.setTop(titleBox);

    VBox menuBox = createMenuBox(stage);
    root.setCenter(menuBox);

    Scene scene = new Scene(root, 800, 600);
    stage.setTitle("Rush Hour Solver - Menu Utama");
    stage.setScene(scene);
    stage.show();
}

private VBox createTitleBox() {
    VBox titleBox = new VBox();
    titleBox.setAlignment(Pos.CENTER);
    titleBox.setBackground(new Background(new BackgroundFill(
        new LinearGradient(0, 0, 0, 1, true,
CycleMethod.NO_CYCLE,
        new Stop(0, Color.web("#ffff00")), new Stop(1,
Color.web("#ffff00"))
    ), CornerRadii.EMPTY, Insets.EMPTY)));

    BackgroundImage checkeredBg = new BackgroundImage(
        new
Image(getClass().getResourceAsStream("/checkered.jpg")),
        BackgroundRepeat.REPEAT, BackgroundRepeat.NO_REPEAT,
        BackgroundPosition.DEFAULT,
        new BackgroundSize(BackgroundSize.AUTO, 20, false,
false, false, false)

```

```

);

Region topBar = new Region();
topBar.setPrefHeight(20);
topBar.setBackground(new Background(checkeredBg));

Region bottomBar = new Region();
bottomBar.setPrefHeight(20);
bottomBar.setBackground(new Background(checkeredBg));

Text title = new Text("RUSH HOUR SOLVER");

title.setFont(Font.loadFont(getClass().getResourceAsStream("/RacingEngine-Sharp.otf"), 36));
title.setFill(Color.BLACK);

FadeTransition fadeIn = new
FadeTransition(Duration.seconds(1.5), title);
fadeIn.setFromValue(0);
fadeIn.setToValue(1);
fadeIn.play();

VBox titleTextBox = new VBox(title);
titleTextBox.setAlignment(Pos.CENTER);
titleTextBox.setPadding(new Insets(10));

titleBox.getChildren().addAll(topBar, titleTextBox, bottomBar);
return titleBox;
}

private VBox createMenuBox(Stage stage) {
    VBox menuBox = new VBox(20);
    menuBox.setAlignment(Pos.CENTER);
    menuBox.setPadding(new Insets(40));
    menuBox.setMaxWidth(350);
    menuBox.setStyle("-fx-background-color: rgba(255, 255, 255, 0.05); -fx-background-radius: 15;");

    Text subtitle = styledLabel("Pilih algoritma dan puzzle:");
    subtitle.setFill(Color.LIGHTYELLOW);

    errorMessageText = styledLabel("");
    errorMessageText.setFill(Color.ORANGERED);

    ComboBox<String> algoSelector = new ComboBox<>();
    heuristicSelector = new ComboBox<>();
    heuristicSelector.getItems().addAll("Blocking", "Distance");
    heuristicSelector.getSelectionModel().selectFirst();
    heuristicSelector.setPrefWidth(250);
    heuristicSelector.setStyle(
        "-fx-font-size: 14px;" +

```

```

        "-fx-background-color: #2c2c2c;" +
        "-fx-text-fill: #ffff00;" +
        "-fx-prompt-text-fill: #ffff00;" +
        "-fx-mark-color: yellow;" +
        "-fx-border-color: yellow;" +
        "-fx-border-radius: 5;" +
        "-fx-background-radius: 5;"
    );
    heuristicSelector.setButtonCell(new
javafx.scene.control.ListCell<>() {
    @Override
    protected void updateItem(String item, boolean empty) {
        super.updateItem(item, empty);
        setText(item);
        setTextFill(Color.web("#ffff00"));
        setFont(customFont);
    }
});
    heuristicSelector.setVisible(false); // hanya terlihat jika GBFS
atau A* dipilih

    algoSelector.getItems().addAll("Uniform Cost Search", "Greedy
Best First Search", "A*");
    algoSelector.getSelectionModel().selectFirst();
    algoSelector.setPrefWidth(250);
    algoSelector.setStyle(
        "-fx-font-size: 14px;" +
        "-fx-background-color: #2c2c2c;" +
        "-fx-text-fill: #ffff00;" +
        "-fx-prompt-text-fill: #ffff00;" +
        "-fx-mark-color: yellow;" +
        "-fx-border-color: yellow;" +
        "-fx-border-radius: 5;" +
        "-fx-background-radius: 5;"
    );
    algoSelector.setButtonCell(new javafx.scene.control.ListCell<>()
{
    @Override
    protected void updateItem(String item, boolean empty) {
        super.updateItem(item, empty);
        setText(item);
        setTextFill(Color.web("#ffff00"));
        setFont(customFont);
    }
});
    algoSelector.setOnAction(event -> {
        int selected =
algoSelector.getSelectionModel().getSelectedIndex();
        heuristicSelector.setVisible(selected == 1 || selected ==
2);
    });

```

```

        Button loadButton = createStyledButton("\uD83D\uDCC2 Load
Puzzle");
        Button solveButton = createStyledButton("\uD83D\uDE80 Cari
Solusi");

        loadButton.setOnAction(e -> {
            FileChooser fileChooser = new FileChooser();
            fileChooser.setTitle("Open Puzzle File");
            fileChooser.setInitialDirectory(new
File(System.getProperty("user.dir")));
            File file = fileChooser.showOpenDialog(stage);
            if (file != null) {
                try {
                    board =
Utils.readRushHourPuzzleFromFile(file.getAbsolutePath());
                    errorMessageText.setFill(Color.LIMEGREEN);
                    errorMessageText.setText("Puzzle berhasil dimuat");
                } catch (IOException | IllegalArgumentException ex) {
                    errorMessageText.setFill(Color.ORANGERED);

                    // split messages into lines
                    String message = ex.getMessage();
                    StringBuilder formattedMessage = new
StringBuilder();

                    int lineLength = 0;
                    final int maxLength = 35;

                    for (String word : message.split(" ")) {
                        if (lineLength + word.length() > maxLength) {
                            formattedMessage.append("\n");
                            lineLength = 0;
                        }
                        formattedMessage.append(word).append(" ");
                        lineLength += word.length() + 1;
                    }

                    errorMessageText.setText(formattedMessage.toString().trim());
                }
            }
        });

        solveButton.setOnAction(e -> {
            errorMessageText.setFill(Color.ORANGERED);
            if (board == null) {
                errorMessageText.setText("Silakan load puzzle terlebih
dahulu");
                return;
            }
            int choice =

```

```

algoSelector.getSelectionModel().getSelectedIndex();
    switch (choice) {
        case 0 -> algorithm = new UniformCostSearch();
        case 1 -> {
            algorithm = new GreedyBestFirstSearch();
            if (algorithm instanceof InformedSearch alg) {
                alg.setHeuristicModel(getSelectedHeuristic());
            }
        }
        case 2 -> {
            algorithm = new AStar();
            if (algorithm instanceof InformedSearch alg) {
                alg.setHeuristicModel(getSelectedHeuristic());
            }
        }
        default -> algorithm = null;
    }

    if (algorithm == null) {
        errorMessageText.setText("Algoritma tidak dikenali");
        return;
    }

    showProcessWindow(stage);

    Task<Void> solveTask = new Task<>() {
        private long duration;
        private boolean found;

        @Override
        protected Void call() {
            long start = System.nanoTime();
            com.syafigriza.rushhoursolver.model.State initial =
new com.syafigriza.rushhoursolver.model.State(board, 0, 0);
            algorithm.solve(initial);
            found = algorithm.getSolution() != null &&
algorithm.getSolution().states.length != 0;
            duration = (System.nanoTime() - start) / 1_000_000;
            return null;
        }

        @Override
        protected void succeeded() {
            showSolutionWindow(stage, found);
        }
    };
    new Thread(solveTask).start();
});

menuBox.getChildren().addAll(subtitle, errorMessageText,
algoSelector, heuristicSelector, loadButton, solveButton);

```

```

        return menuBox;
    }

    private void showProcessWindow(Stage stage) {
        BorderPane processLayout = new BorderPane();
        processLayout.setBackground(new Background(new BackgroundFill(
            new LinearGradient(0, 0, 1, 1, true,
CycleMethod.NO_CYCLE,
            new Stop(0, Color.web("#0f0f0f")), new Stop(1,
Color.web("#1c1c1c"))
            ), CornerRadii.EMPTY, Insets.EMPTY)));

        resultText = styledLabel("Mencari solusi...");
        resultText.setStyle("-fx-font-size: 22px; -fx-font-weight:
bold;");
        resultText.setFill(Color.YELLOW);

        boardViewContainer = new VBox();
        boardViewContainer.setAlignment(Pos.CENTER);

        VBox card = new VBox(20, resultText, boardViewContainer);
        card.setAlignment(Pos.CENTER);
        card.setPadding(new Insets(30));
        card.setMaxWidth(400);
        card.setStyle("-fx-background-color: rgba(255, 255, 255, 0.05);
-fx-background-radius: 15;");

        BorderPane.setAlignment(card, Pos.CENTER);
        processLayout.setCenter(card);

        Scene processScene = new Scene(processLayout, 800, 600);
        stage.setScene(processScene);
    }

    private void showSolutionWindow(Stage stage, boolean found) {
        BorderPane layout = new BorderPane();
        layout.setBackground(new Background(new BackgroundFill(
            new LinearGradient(0, 0, 1, 1, true,
CycleMethod.NO_CYCLE,
            new Stop(0, Color.web("#0f0f0f")), new Stop(1,
Color.web("#1c1c1c"))
            ), CornerRadii.EMPTY, Insets.EMPTY)));

        Algorithm.SolutionData solutionData = algorithm.getSolution();

        Text titleText = styledLabel(found ? "Solusi ditemukan" :
"Solusi tidak ditemukan");
        titleText.setFill(Color.WHITE);
        titleText.setStyle("-fx-font-size: 22px; -fx-font-weight:
bold;");
    }

```

```

        Text infoText = styledLabel2(
            "Waktu Pencarian: " + solutionData.timeElapsedMs + "
ms\n" +
            "Jumlah Simpul: " + solutionData.nodeCount +
            "\n" +
            "Jumlah Langkah: " +
solutionData.states.length);
        infoText.setStyle("-fx-font-size: 14px;");
        infoText.setFill(Color.LIGHTGRAY);

        BoardView boardView = new BoardView();
        double scale = Math.min(1.0, 600.0 / (board.getCols() * 60));
        boardView.setScaleX(scale);
        boardView.setScaleY(scale);

        HBox contentBox = new HBox(40, boardView, infoText);
        contentBox.setAlignment(Pos.CENTER);

        Button replayButton = createStyledButton("🔄 Replay Solusi");
        replayButton.setDisable(!found || solutionData.states == null ||
solutionData.states.length == 0);

        Button backButton = createStyledButton("← Kembali");
        backButton.setOnAction(e -> start(stage));

        VBox card = new VBox(20, titleText, contentBox, replayButton,
backButton);
        card.setAlignment(Pos.CENTER);
        card.setPadding(new Insets(30));
        card.setMaxWidth(600);
        card.setStyle("-fx-background-color: rgba(255, 255, 255, 0.05);
-fx-background-radius: 15;");

        layout.setCenter(card);
        Scene solutionScene = new Scene(layout, 800, 600);
        stage.setScene(solutionScene);

        Runnable runReplay = () -> {
            Task<Void> animationTask = new Task<>() {
                @Override
                protected Void call() throws Exception {
                    for (var state : solutionData.states) {
                        Platform.runLater(() -> boardView.draw(state));
                        Thread.sleep(500);
                    }
                    return null;
                }
            };
            new Thread(animationTask).start();
        };
    };
}

```

```

        if (found && solutionData.states != null &&
solutionData.states.length != 0) {
            runReplay.run();
            replayButton.setAction(e -> runReplay.run());
        }
    }

    private void showAlert(String title, String message) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }

    private Button createStyledButton(String text) {
        Button button = new Button(text);
        button.setFont(customFont);
        button.setStyle("-fx-background-color: linear-gradient(to
bottom, #fff200, #ffd000); -fx-text-fill: black; " +
            "-fx-font-size: 16px; -fx-border-color: black;
-fx-border-width: 2px; " +
            "-fx-border-radius: 8; -fx-background-radius: 8;
-fx-padding: 8px 16px;");
        button.setOnMouseEntered(e ->
button.setStyle("-fx-background-color: black; -fx-text-fill: white; " +
            "-fx-font-size: 16px; -fx-border-color: white;
-fx-border-width: 2px; " +
            "-fx-border-radius: 8; -fx-background-radius: 8;
-fx-padding: 8px 16px;"));
        button.setOnMouseExited(e ->
button.setStyle("-fx-background-color: linear-gradient(to bottom,
#fff200, #ffd000); -fx-text-fill: black; " +
            "-fx-font-size: 16px; -fx-border-color: black;
-fx-border-width: 2px; " +
            "-fx-border-radius: 8; -fx-background-radius: 8;
-fx-padding: 8px 16px;"));
        return button;
    }

    private Text styledLabel(String content) {
        Text label = new Text(content);
        label.setFont(customFont);
        label.setFill(Color.WHITE);
        return label;
    }

    private Text styledLabel2(String content) {
        Font vorcasFont =

```



```

Font.loadFont(getClass().getResourceAsStream("/ShockSurgent.otf"), 14);
    Text label = new Text(content);
    label.setFont(vorcasFont);
    label.setFill(Color.WHITE);
    return label;
}

    public static void main(String[] args) {
        launch(args);
    }
}

```

#### 4.2.2.1. BoardView.java

```

package com.syafiqriza.rushhoursolver.view;

import com.syafiqriza.rushhoursolver.model.Board;
import com.syafiqriza.rushhoursolver.model.State;
import javafx.geometry.Insets;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

import java.util.HashMap;
import java.util.Map;
import java.util.Random;

/**
 * Komponen visual untuk menampilkan papan permainan Rush Hour.
 */
public class BoardView extends GridPane {
    private static final int TILE_SIZE = 60;
    private final Map<Character, Color> colorMap = new HashMap<>();
    private final Random random = new Random();
    private char[][] currentGrid;

    public BoardView() {
        setHgap(3);
        setVgap(3);
        setPadding(new Insets(15));
        setStyle("-fx-background-color: linear-gradient(to bottom right, #1c1c1c, #2a2a2a);"
            + "-fx-border-color: yellow; -fx-border-width: 3;"
            + "-fx-border-radius: 10;");
    }

    /**
     * Menggambar ulang papan berdasarkan objek Board.
     * @param board Objek Board yang akan digambarkan.
     */
    public void draw(Board board) {

```

```

        this.currentGrid = board.getGrid();
        getChildren().clear();

        int rows = board.getRows();
        int cols = board.getCols();
        int goalRow = board.getGoalRowClamped();
        int goalCol = board.getGoalColClamped();

        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                Rectangle tile = new Rectangle(TILE_SIZE, TILE_SIZE);
                char c = currentGrid[row][col];

                tile.setFill(getColorForCar(c));
                tile.setStroke(Color.web("#cccccc"));
                tile.setStrokeWidth(1.5);
                tile.setArcWidth(16);
                tile.setArcHeight(16);

                if (row==goalRow && col == goalCol) {
                    tile.setStroke(Color.YELLOW);
                    tile.setStrokeWidth(3.0);
                }

                add(tile, col, row);
            }
        }
    }

    /**
     * Menggambar ulang papan berdasarkan grid.
     * @param grid Matriks karakter keadaan papan.
     */
    public void draw(char[][] grid) {
        getChildren().clear();
        int rows = grid.length;
        int cols = grid[0].length;

        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                Rectangle tile = new Rectangle(TILE_SIZE, TILE_SIZE);
                char c = grid[row][col];

                tile.setFill(getColorForCar(c));
                tile.setStroke(Color.web("#cccccc"));
                tile.setStrokeWidth(1.5);
                tile.setArcWidth(16);
                tile.setArcHeight(16);

                add(tile, col, row);
            }
        }
    }

```

```

    }
}

/**
 * Menggambar ulang papan berdasarkan State.
 * @param state State yang akan digambarkan.
 */
public void draw(State state) {
    draw(state.getBoard());
}

private static final Color[] PRESET_COLORS = {
    Color.hsb(150, 0.8, 0.85), // Spring Green
    Color.hsb(210, 0.9, 0.9),  // Azure
    Color.hsb(330, 0.8, 0.8),  // Pink
    Color.hsb(270, 0.9, 0.85), // Purple
    Color.hsb(75, 0.8, 0.85),  // Lime
    Color.hsb(285, 0.8, 0.85), // Orchid
    Color.hsb(180, 0.9, 0.9),  // Cyan
    Color.hsb(255, 0.8, 0.9),  // Violet
    Color.hsb(60, 0.8, 0.9),   // Yellow
    Color.hsb(135, 0.8, 0.8),   // Forest Green
    Color.hsb(300, 0.9, 0.85),  // Magenta
    Color.hsb(195, 0.8, 0.85),  // Sky Blue
    Color.hsb(30, 0.9, 0.9),    // Orange
    Color.hsb(120, 0.9, 0.9),   // Green
    Color.hsb(165, 0.7, 0.85),  // Aqua Green
    Color.hsb(90, 0.7, 0.8),    // Yellow-Green
    Color.hsb(225, 0.8, 0.85),  // Blue
    Color.hsb(45, 0.8, 0.85),   // Yellow-Orange
    Color.hsb(240, 0.9, 0.85),  // Indigo
    Color.hsb(315, 0.7, 0.85)   // Rose
};

/**
 * Menghasilkan warna berdasarkan ID mobil.
 * @param c Karakter ID mobil.
 * @return Warna tile.
 */
private int colorIndex = 0;

private Color getColorForCar(char c) {
    if (c == '.') return Color.web("#444444");
    if (c == 'P') return Color.web("#ff3b3b");

    if (!colorMap.containsKey(c)) {
        Color color = PRESET_COLORS[colorIndex %
PRESET_COLORS.length];

```

```

        colorMap.put(c, color);
        colorIndex++;
    }
    return colorMap.get(c);
}

/**
 * Mendapatkan grid terakhir yang ditampilkan.
 * @return Matriks grid terakhir.
 */
public char[][] getCurrentGrid() {
    return currentGrid;
}
}

```

### 4.2.3 Main.java

```

package com.syafigriza.rushhoursolver;

import com.syafigriza.rushhoursolver.view.GUI;
import com.syafigriza.rushhoursolver.view.CLI;

import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        if (args.length > 0 && args[0].equalsIgnoreCase("cli")) {
            CLI.main(Arrays.copyOfRange(args, 1, args.length));
        } else {
            GUI.main(args);
        }
    }
}

```

## BAB V

### IMPLEMENTASI BONUS

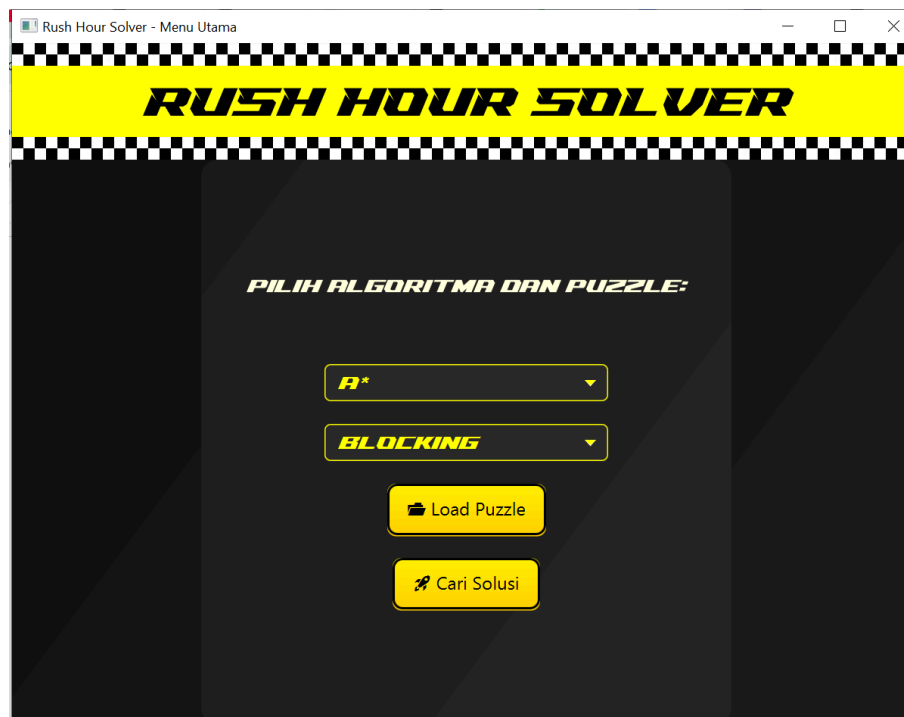
#### 5.1. Graphical User Interface

Dalam tugas kecil ini, penulis berhasil mengimplementasikan bonus Graphical User Interface. Bonus ini dikerjakan dengan pustaka JavaFX.

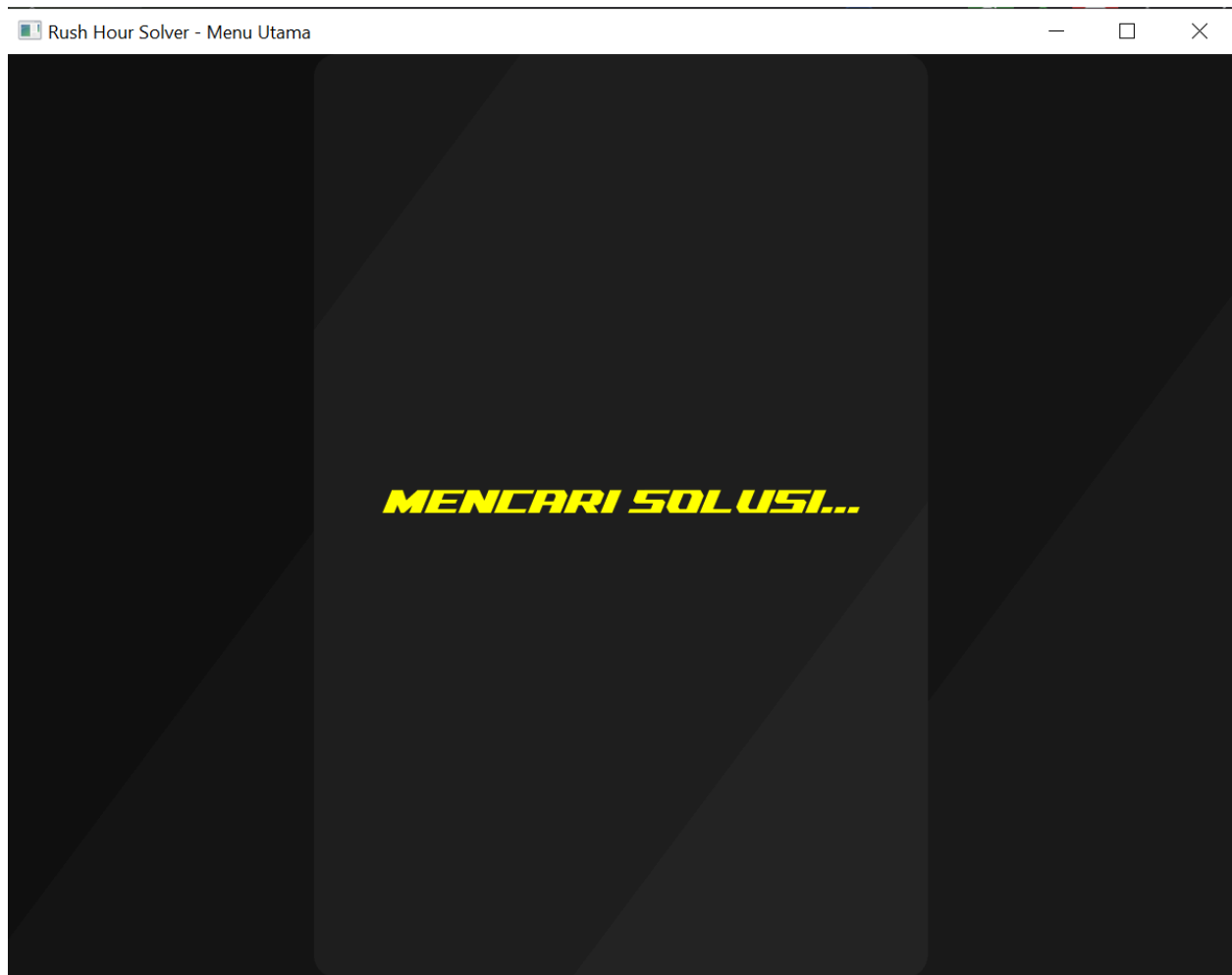
GUI memiliki tiga window, yaitu menu utama, tampilan pencarian solusi, dan presentasi solusi. Pada menu utama, pengguna dapat memilih konfigurasi puzzle dari file .txt. Konfigurasi yang tidak valid akan memunculkan pesan error. Pengguna juga dapat memilih algoritma yang akan digunakan (UCS/GBFS/A\*) dan heuristiknya (Blocking/Distance) jika memilih GBFS/A\*. Setelah memilih algoritma dan puzzle, pengguna dapat memulai pencarian solusi.

Pencarian solusi akan dilakukan sementara GUI menampilkan pesan “Mencari Solusi” pada window tampilan pencarian solusi. Setelah selesai, GUI akan menampilkan window presentasi solusi.

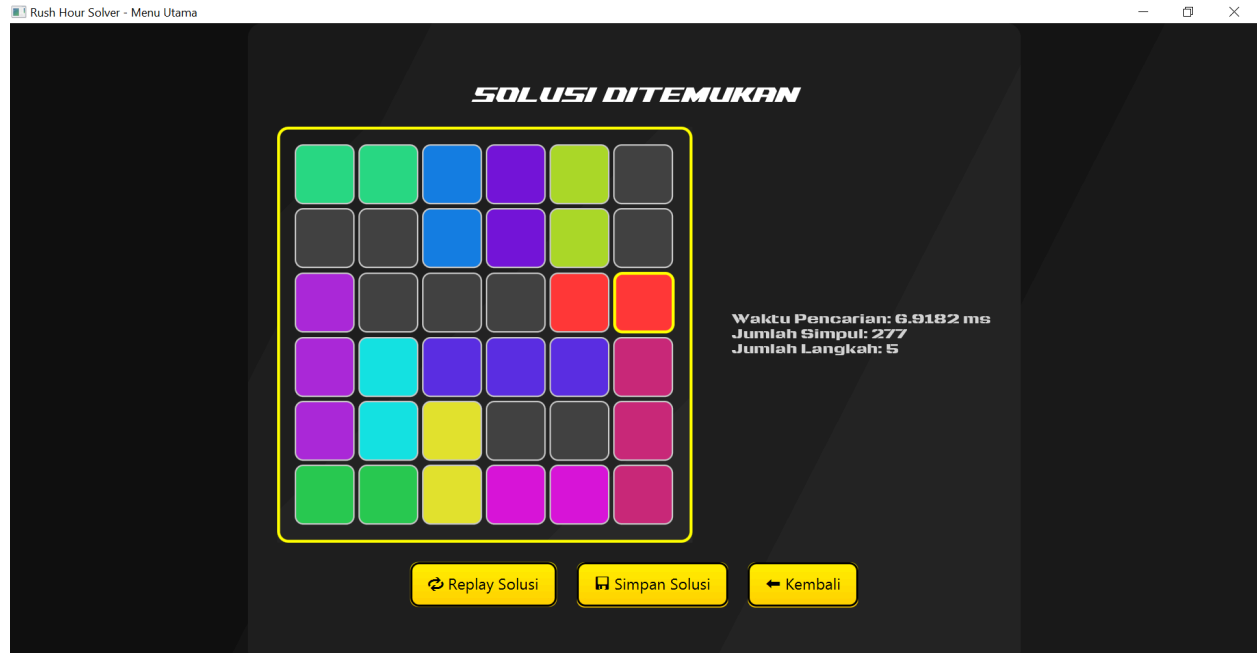
Hasil pencarian akan dipresentasikan dengan animasi sekuens status awal hingga status akhir. Animasi tersebut dapat diputar ulang menggunakan tombol replay. Presentasi hasil pencarian juga memberikan informasi berupa waktu pencarian, jumlah simpul, dan jumlah langkah. Untuk kembali ke menu utama, tombol “Kembali” dapat ditekan.



Gambar 5.1.1 Tampilan Halaman Utama Rush Hour Solver



Gambar 5.1.2 Tampilan proses pencarian solusi



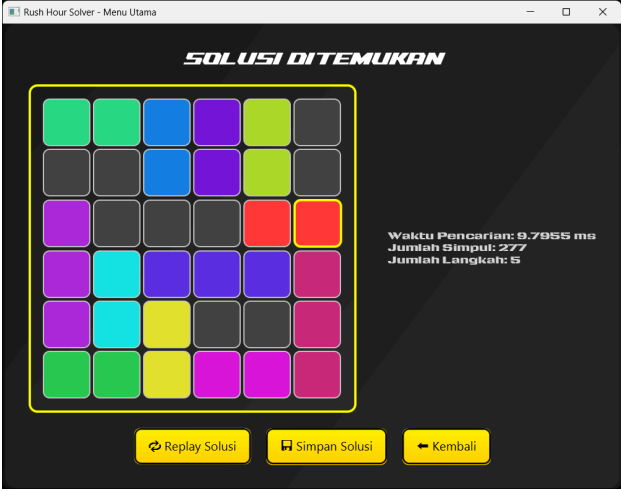
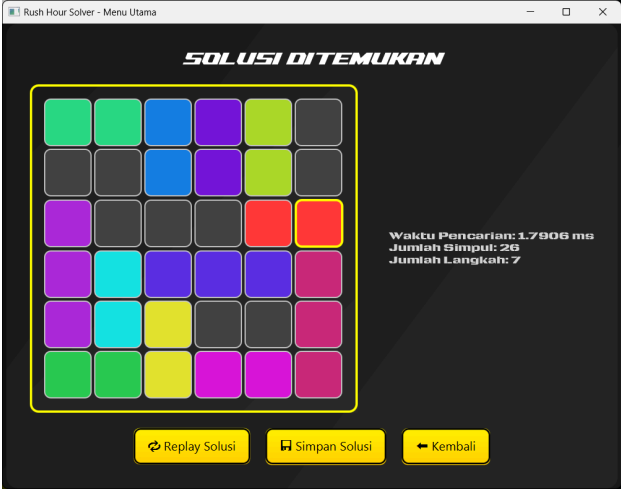
Gambar 5.1.3 Tampilan hasil pencarian

## 5.2. Heuristik Alternatif

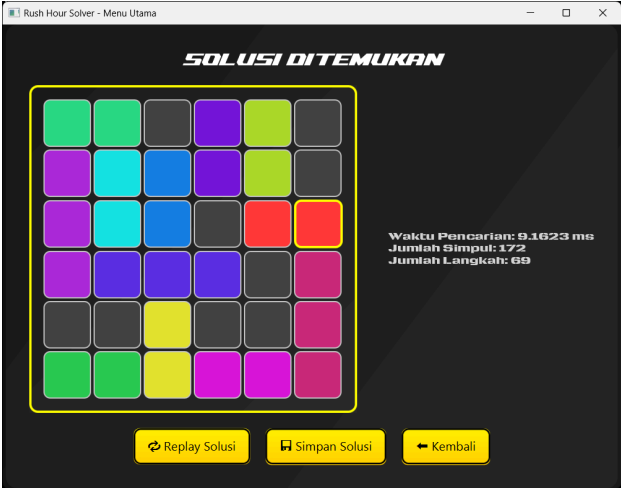
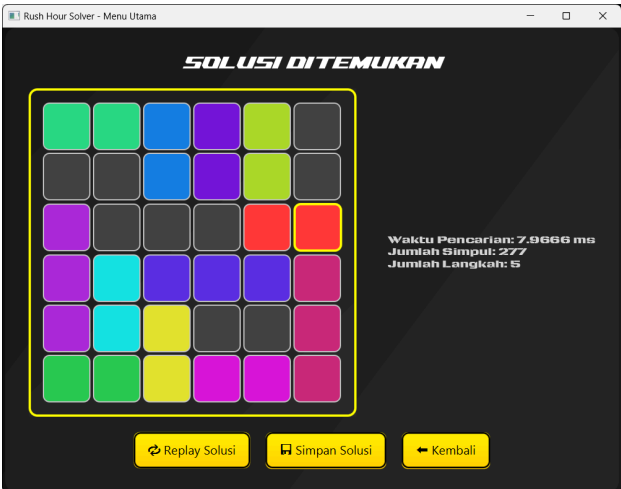
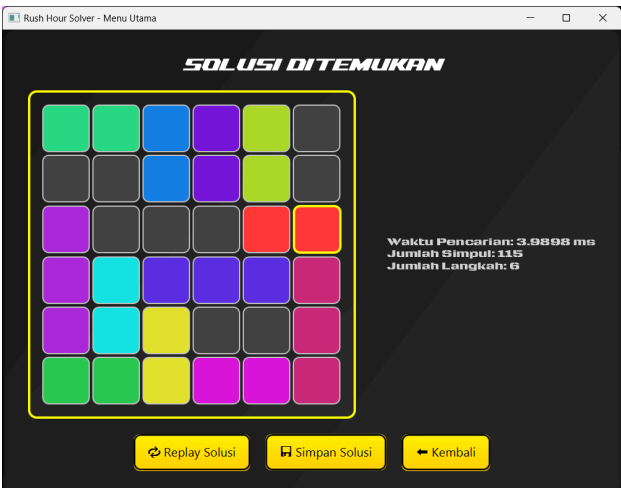
Dalam tugas kecil ini, penulis berhasil mengimplementasikan bonus heuristik alternatif dengan mengimplementasikan dua model heuristik yang dapat dipilih pengguna ketika menggunakan algoritma GBFS atau A\*, yaitu jumlah mobil yang menghalangi goal dan jarak mobil utama ke goal.

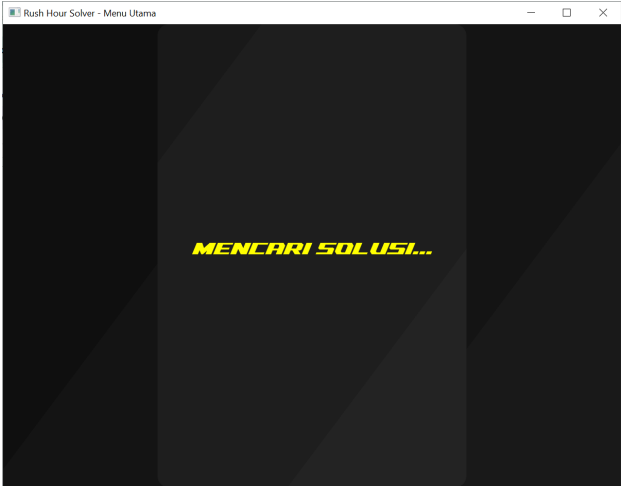
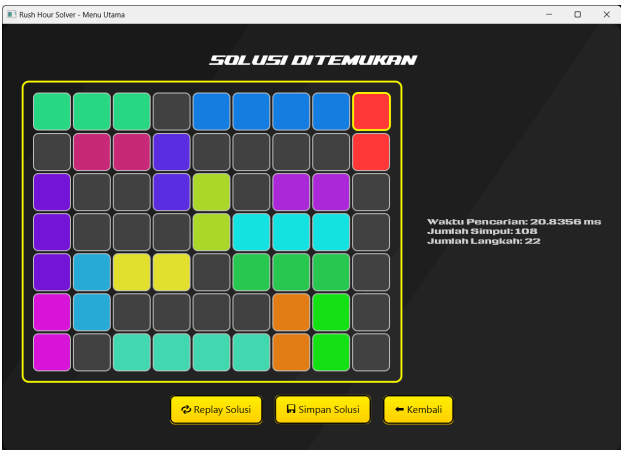
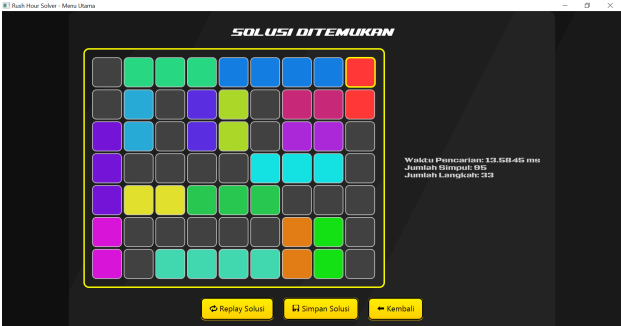
## BAB VI EKSPERIMEN DAN ANALISIS

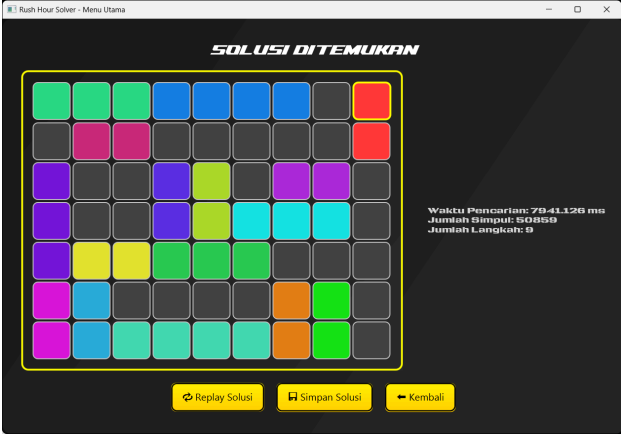
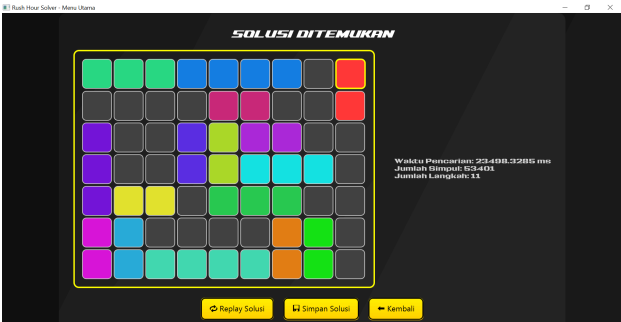
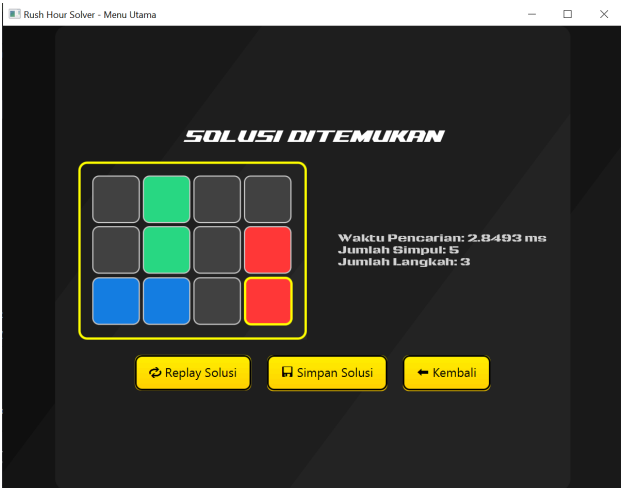
### 6.1. Pengujian Program

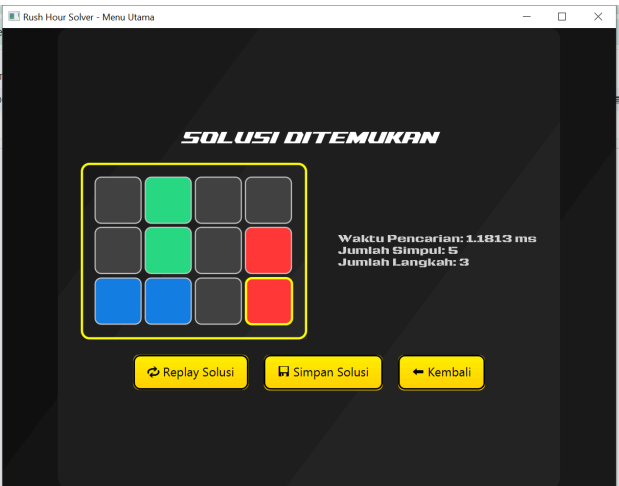
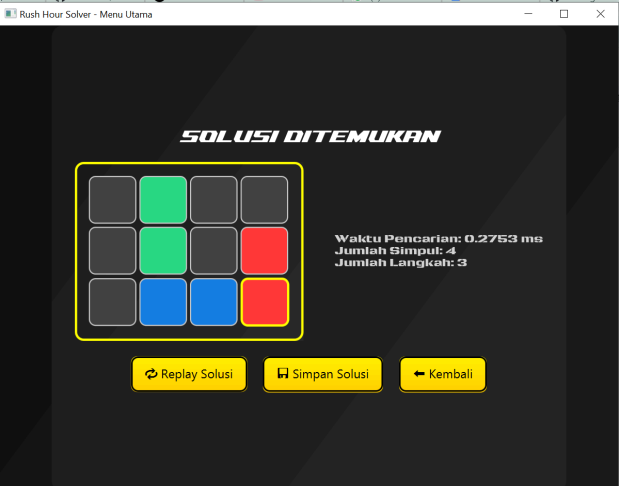
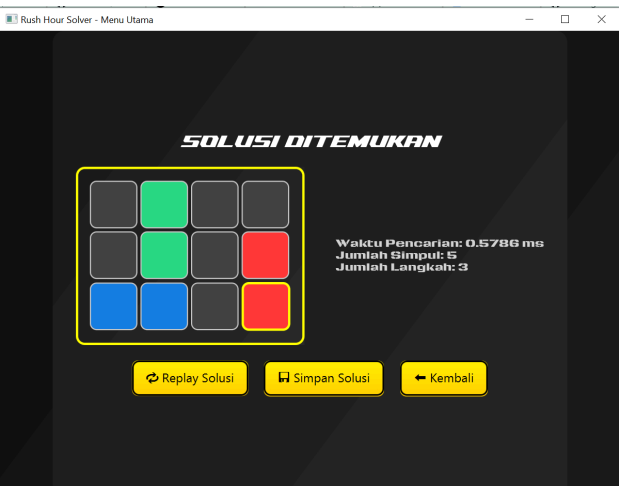
No	Input	Algoritma	Heuristik	Screenshot
1.	6 6 11 AAB...F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	UCS	-	
2.	6 6 11 AAB...F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	GBFS	Blocking	

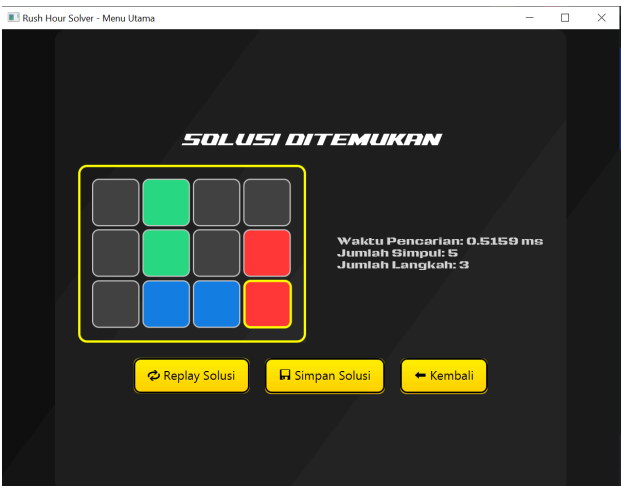
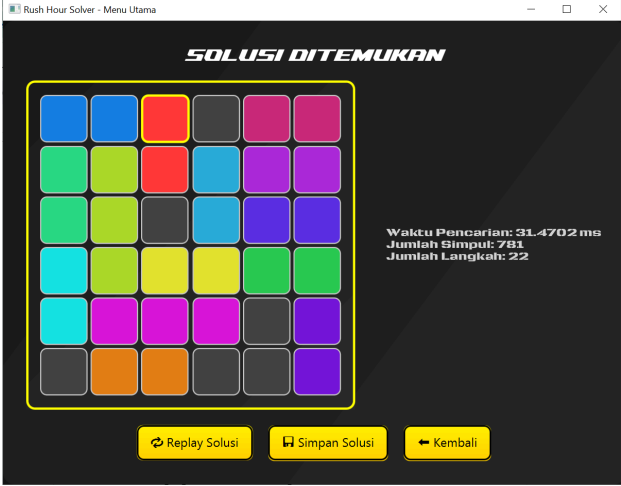
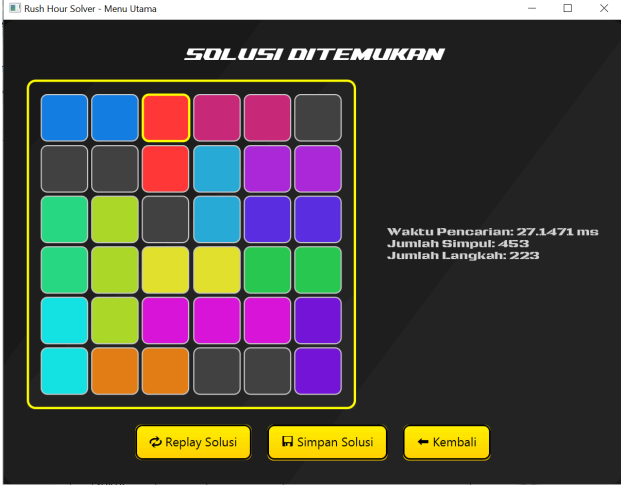


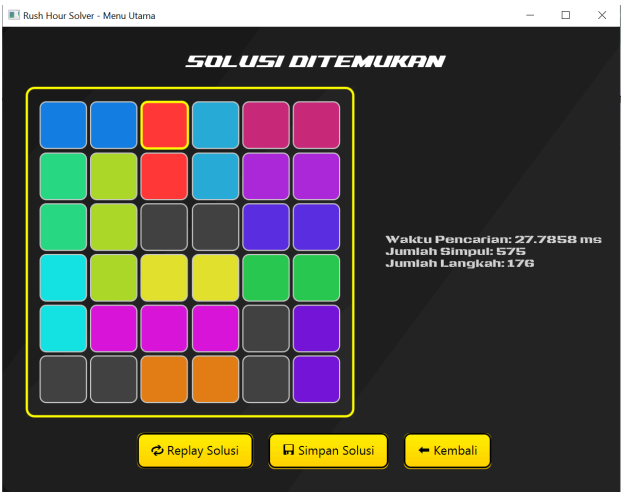
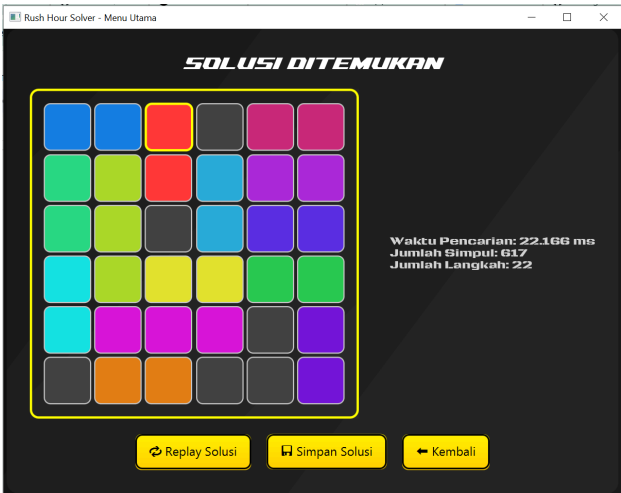
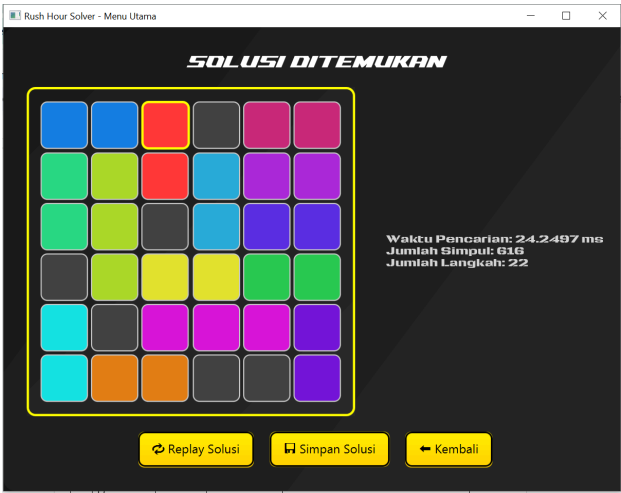
3.	6 6 11 AAB...F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	GBFS	Distance	
4.	6 6 11 AAB...F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	A*	Blocking	
5.	6 6 11 AAB...F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	A*	Distance	


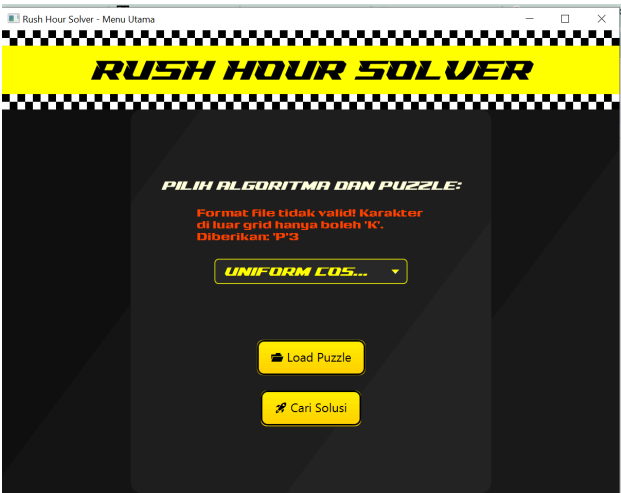
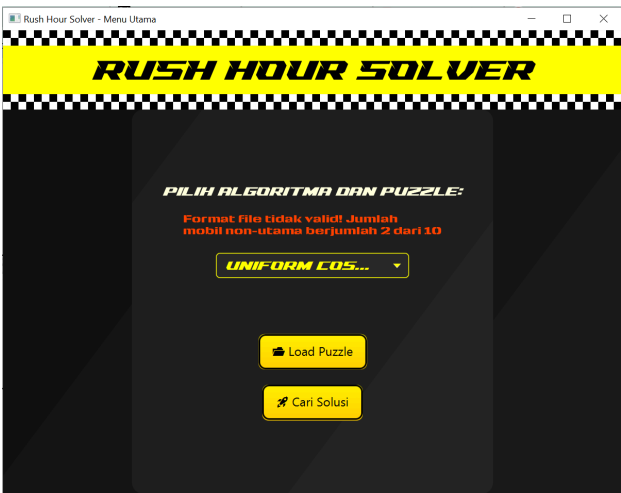
6.	7 9 15  K ..AAABBBB .....CC D...E...FF D...E.GGG D..HIIJJJ LM.H..NOP LMQQQQNOP	UCS	-	 <p>setelah 1 jam..</p>
7.	7 9 15  K ..AAABBBB .....CC D...E...FF D...E.GGG D..HIIJJJ LM.H..NOP LMQQQQNOP	GBFS	Blocking	
8.	7 9 15  K ..AAABBBB .....CC D...E...FF D...E.GGG D..HIIJJJ LM.H..NOP LMQQQQNOP	GBFS	Distance	


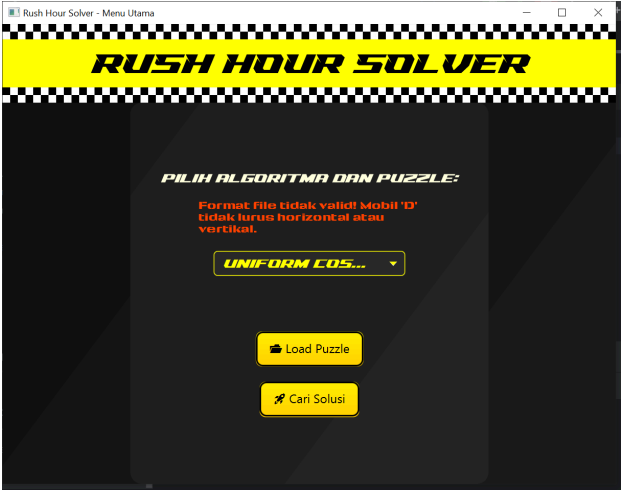
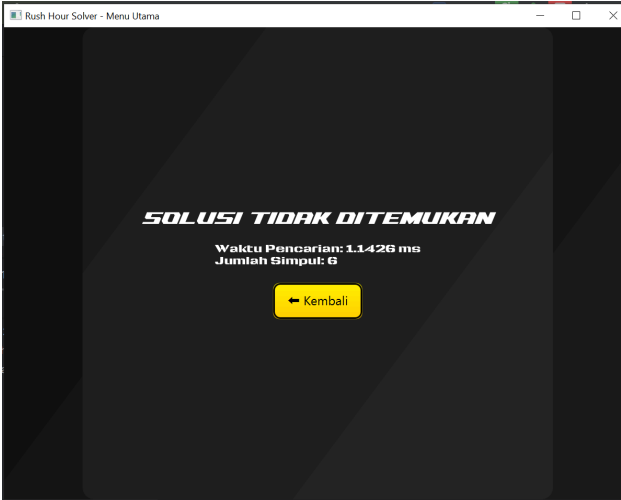
9.	7 9 15  K ..AAABBBB .....CC D...E...FF D...E.GGG D..HIIJJJ LM.H..NOP LMQQQQNOP	A*	Blocking	
10.	7 9 15  K ..AAABBBB .....CC D...E...FF D...E.GGG D..HIIJJJ LM.H..NOP LMQQQQNOP	A*	Distance	
11.	3 4 2 ...P .A.P .ABB K	UCS	-	

12.	3 4 2 ...P .A.P .ABB K	GBFS	Blocking	
13.	3 4 2 ...P .A.P .ABB K	GBFS	Distance	
14.	3 4 2 ...P .A.P .ABB K	A*	Blocking	

15.	3 4 2 ...P .A.P .ABB K	A*	Distance	
16.	6 6 13 K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM	UCS	-	
17.	6 6 13 K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM	GBFS	Blocking	

18.	6 6 13 K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM	GBFS	Distance	
19.	6 6 13 K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM	A*	Blocking	
20.	6 6 13 K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM	A*	Distance	

21.	6 6 11 AAB...F ..BCDF GPPCDF GH.IIIK GHJ... LLJMM.	-	-	
22.	4 3 2 ...P .A.P .ABB K	-	-	
23.	3 4 10 ...P .A.P .ABB K	-	-	

24.	6 6 9 ..AABB .CCD.. .E.DFF K.E.PPG .E..HG .IIPHG	-	-	
25.	6 6 9 AA.P.. BCCP.. BD.EEE DDDDFF .GGG.. .HHII. K			
26.	6 6 12 N.AABB NCCDQQ NEODFF KNEOPPG NEO.HG NIIIHG			



## 6.2. Analisis Hasil

Hasil percobaan menunjukkan bahwa A\* dengan heuristik *admissible* (blocking cars) secara konsisten menghasilkan solusi optimal (jumlah langkah minimum) dengan waktu dan ruang komputasi yang relatif efisien. Hal ini karena A\* mampu menyeimbangkan eksplorasi antara langkah yang sudah diambil ( $g(n)$ ) dan estimasi langkah ke depan ( $h(n)$ ), sambil tetap menjaga optimalitas karena  $h(n)$  tidak pernah melebihi estimasi. A\* menunjukkan keunggulan dalam kasus puzzle yang kompleks, di mana ruang pencarian sangat besar dan solusi optimal sulit ditemukan secara brute-force.

Sebaliknya, UCS, meskipun juga menjamin solusi optimal karena hanya bergantung pada  $g(n)$ , membutuhkan waktu dan memori jauh lebih besar dibanding A\*, terutama pada puzzle dengan banyak cabang atau solusi yang letaknya dalam. Hal ini karena UCS tidak memiliki arah dalam pencarian dan mengevaluasi semua kemungkinan secara menyeluruh berdasarkan kedalaman. Sudah dijelaskan juga sebelumnya bahwa pada permasalahan puzzle Rush Hour, UCS bekerja serupa dengan BFS. Dalam praktiknya, UCS cenderung mengunjungi lebih banyak simpul yang sebenarnya tidak relevan terhadap solusi.

Sementara itu, GBFS menawarkan waktu pencarian tercepat, namun dengan kualitas solusi yang tidak optimal. Ini karena GBFS hanya mempertimbangkan  $h(n)$  tanpa memperhitungkan langkah-langkah yang sudah ditempuh ( $g(n) = 0$ , dianggap sama untuk semua simpul). Ketika menggunakan heuristik jumlah pemblokir, GBFS memang lebih terarah, tetapi tetap rentan memilih jalur pendek yang tampak menjanjikan namun mungkin buntu. Ketika menggunakan heuristik tidak *admissible* (jarak ke goal), kecepatan meningkat tetapi sering kali solusi yang ditemukan memiliki langkah lebih panjang dibanding A\* dan UCS. Ini mengonfirmasi bahwa GBFS mengorbankan optimalitas demi efisiensi waktu.

Secara keseluruhan, percobaan menunjukkan bahwa A\* dengan heuristik *admissible* adalah pendekatan paling seimbang, menghasilkan solusi optimal dengan efisiensi yang baik. UCS tetap berguna sebagai referensi pencarian untuk solusi optimal, tetapi mahal secara komputasi. Sedangkan GBFS cocok untuk kasus di mana waktu pencarian jauh lebih penting daripada kualitas solusi.

## 6.3. Analisis Kompleksitas Algoritma

### 6.3.1. Kompleksitas Algoritma UCS

UCS bekerja dengan menjelajahi simpul berdasarkan biaya kumulatif dari status awal ( $g(n)$ ), sehingga dapat dikatakan serupa dengan BFS, di mana setiap langkah memiliki biaya yang sama. UCS akan mengunjungi semua simpul pada kedalaman  $d$  sebelum melanjutkan ke  $d + 1$ .

Kompleksitas waktu dan ruangnya adalah eksponensial terhadap kedalaman solusi  $d$ , dengan  $b$  sebagai branching factor (rata-rata jumlah gerakan dari satu posisi). Ini menjelaskan mengapa dalam percobaan, UCS membutuhkan memori dan waktu yang besar, terutama ketika solusi berada pada kedalaman tinggi. UCS memiliki kompleksitas waktu  $O(b^d)$  dan kompleksitas ruang  $O(b^d)$ .

### 6.3.2. Kompleksitas Algoritma GBFS

GBFS hanya mempertimbangkan fungsi heuristik  $h(n)$  untuk memilih simpul berikutnya, tanpa memperhatikan berapa langkah yang sudah diambil. Hal ini membuat pencarian sangat cepat pada banyak kasus, terutama ketika  $h(n)$  mengarahkan pencarian secara efektif. Namun, karena tidak menyimpan riwayat biaya, GBFS seringkali salah arah atau terjebak pada jalur yang tampak menjanjikan tapi tidak benar-benar mendekatkan ke solusi, terutama jika heuristik tidak *admissible*.

Kompleksitas GBFS eksponensial dalam kasus terburuk, tetapi dalam praktik bisa jauh lebih rendah, seperti yang terlihat dari percobaan di mana GBFS menyelesaikan puzzle jauh lebih cepat (meski tidak optimal). GBFS memiliki kompleksitas waktu  $O(b^m)$  dengan  $m$  adalah kedalaman status dari solusi yang ditemukan dan kompleksitas ruang  $O(b^m)$ .

### 6.3.3. Kompleksitas Algoritma A\*

A\* menggabungkan keunggulan algoritma UCS dan GBFS dengan mengevaluasi status berdasarkan  $f(n) = g(n) + h(n)$ . Kompleksitas terburuknya tetap eksponensial, tetapi jumlah simpul yang dikunjungi bisa sangat berkurang jika heuristik  $h(n)$  *admissible*.

Dalam percobaan, A\* berhasil menemukan solusi optimal seperti UCS, tetapi dengan waktu dan memori yang jauh lebih hemat, membuktikan bahwa pemanfaatan heuristik yang baik secara signifikan meningkatkan efisiensi. A\* memiliki kompleksitas waktu  $O(b^d)$  pada kasus terburuk dan kompleksitas ruang  $O(b^d)$ .

## LAMPIRAN

### Lampiran Checklist Spesifikasi Program

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	✓	

### Lampiran Repository Program

*Repository* program dapat dibuka lewat tautan berikut:

[https://github.com/L4mbads/Tucil3\\_13523135\\_13523162](https://github.com/L4mbads/Tucil3_13523135_13523162)