

IF2211 Strategi Algoritma
Laporan Tugas Kecil 3



Disusun oleh:

Rafael Marchel Darma Wijaya (13523146)
Muhammad Kinan Arkansyaddad (13523152)

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG 40132
2025

Daftar Isi

Daftar Isi	2
BAB I	
DESKRIPSI MASALAH DAN ALGORITMA	5
1.1. Penyelesain Rush Hour dengan Algoritma Pathfinding	5
1.2. Algoritma Uniform Cost Search (UCS)	7
1.3. Fungsi Heuristik	9
1.4. Algoritma Greedy Best First Search	10
1.5. Algoritma A*	11
1.6. [BONUS] Algoritma IDA*	13
BAB II	
IMPLEMENTASI ALGORITMA DALAM BAHASA JAVA	15
2.1. Implementasi Fungsi Heuristik Jarak	15
2.2. Implementasi Fungsi Heuristik Blocker	16
2.3. Implementasi Fungsi Heuristik Kombinasi	19
2.4. Implementasi UCS, GBFS, A*	19
2.5. Implementasi IDA*	22
BAB III	
SOURCE CODE PROGRAM	25
3.1. Repository Program	25
3.2. Source Code Algoritma Pathfinding	26
3.2.1. InputReader.java	26
3.2.2. GameSolverGUI.java	32
3.2.3. GameBoard.java	45
3.2.4. GameEnums.java	50
3.2.5. GamePiece.java	51
3.2.6. GameState.java	57
3.2.7. Algorithm	62

3.2.8. Comparator	70
3.2.9. Heuristic	72
3.2.10. GameSolver.java	76
BAB IV	
EKSPERIMEN	80
4.1 Test Case 1-1: UCS	80
4.2 Test Case 1-2A: Greedy Best + Combine Heuristic	82
4.3 Test Case 1-3A: A* + Combine Heuristic	85
4.4 Test Case 1-2B: Greedy Best + Manhattan Distance Heuristic	88
4.5 Test Case 1-2C: Greedy Best + Blocker Count Heuristic	91
4.6 Test Case 1-3B: A* + Manhattan Distance Heuristic	94
4.7 Test Case 1-3C: A*+ Blocker Count Heuristic	97
4.8 Test Case 2-1: UCS	100
4.9 Test Case 2-2A: Greedy Best + Combine Heuristic	103
4.10 Test Case 2-3A: A* + Combine Heuristic	106
4.11 Test Case 2-2B: Greedy Best + Manhattan Distance Heuristic	109
4.12 Test Case 2-2C: Greedy Best + Blocker Count Heuristic	112
4.13 Test Case 2-3B: A* + Manhattan Distance Heuristic	115
4.14 Test Case 2-3C: A*+ Blocker Count Heuristic	118
4.15 Test Case 1-4A: IDA*+ Combine Heuristic	121
4.16 Test Case 1-4B: IDA*+ Manhattan Distance Heuristic	124
4.17 Test Case 1-4C: IDA*+ Blocker Count Heuristic	127
4.18 Test Case 2-4A: IDA*+ Combine Heuristic	130
4.19 Test Case 2-4B: IDA*+ Manhattan Distance Heuristic	133
4.20 Test Case 2-4C: IDA*+ Blocker Count Heuristic	136
4.21 Test Case 3-1: UCS	139
4.22 Test Case 3-2A: Greedy + Combined Heuristic	142
4.23 Test Case 3-3A: A* + Combined Heuristic	144
4.24 Test Case 3-4A: IDA* + Combined Heuristic	147

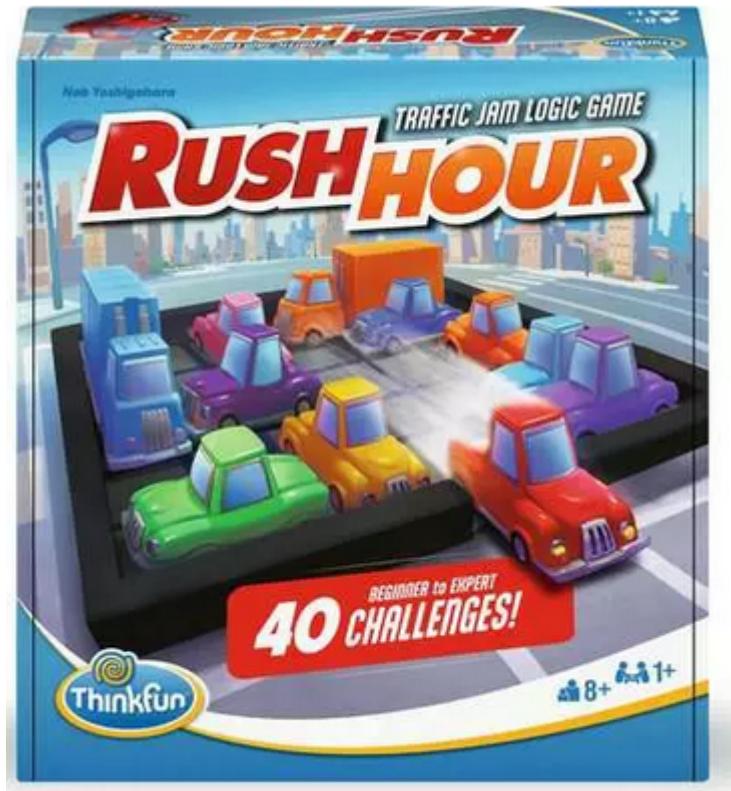
BAB V

ANALISIS DAN PEMBAHASAN	151
5.1 Analisis Hasil Percobaan	151
5.2 Analisis Kompleksitas Waktu Program	157
BAB VI	
LAMPIRAN	160
Daftar Pustaka	162

BAB I

DESKRIPSI MASALAH DAN ALGORITMA

1.1. Penyelesain Rush Hour dengan Algoritma Pathfinding



Gambar 1. Rush Hour Puzzle

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

Komponen penting dari permainan Rush Hour terdiri dari:

1. **Papan** – Papan merupakan tempat permainan dimainkan.

Papan terdiri atas cell, yaitu sebuah singular point dari papan. Sebuah piece akan menempati cell-cell pada papan. Ketika permainan dimulai, semua piece telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi piece dan orientasi, antara horizontal atau vertikal.

Hanya primary piece yang dapat digerakkan keluar papan melewati pintu keluar. Piece yang bukan primary piece tidak dapat digerakkan keluar papan. Papan memiliki satu pintu keluar yang pasti berada di dinding papan dan sejajar dengan orientasi primary piece.

2. **Piece** – Piece adalah sebuah kendaraan di dalam papan. Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece. Secara standar, variasi ukuran sebuah piece adalah 2-piece (menempati 2 cell) atau 3-piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.
3. **Primary Piece** – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.
4. **Pintu Keluar** – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan
5. **Gerakan** – Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

1.2. Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search (UCS) adalah salah satu algoritma *uninformed search*. UCS bekerja dengan cara mengeksplorasi graf berdasarkan *total cost* dari simpul awal ke simpul yang sedang dievaluasi. Tujuan utama UCS adalah menemukan jalur dengan *cost minimum* menuju simpul tujuan.

UCS menggunakan *priority queue* untuk menyimpan daftar simpul, di mana setiap simpul diberi prioritas berdasarkan fungsi:

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

dengan $g(n)$ adalah *total cost* dari simpul awal ke simpul n. Karena UCS hanya mempertimbangkan $g(n)$ tanpa memperhitungkan estimasi ke tujuan, ia tergolong algoritma *blind* (tidak menggunakan heuristik).

UCS bersifat lengkap, artinya algoritma ini akan selalu menemukan solusi jika solusi tersebut ada, dengan syarat:

- Faktor percabangan (*branching factor*) terbatas.
- *Cost* dari setiap aksi bernilai positif dan lebih besar dari nol.

UCS juga dijamin optimal, karena algoritma ini memilih simpul berdasarkan *cost* terkecil. Dengan demikian, simpul tujuan pertama yang ditemukan pasti merupakan bagian dari jalur dengan *total cost minimum*.

Kompleksitas waktu UCS sangat bergantung pada beberapa faktor yakni, jumlah maksimal simpul anak yang dapat dihasilkan dari suatu simpul (*branching factor b*), *total cost minimum* dari simpul awal ke simpul tujuan (C), dan *cost* terkecil dari semua aksi yang mungkin dilakukan (ϵ , $\epsilon > 0$). UCS akan membuat semua simpul yang memiliki *total cost* $\leq C$. Sehingga UCS akan menghasilkan *tree* dengan level sebanyak $\frac{C}{\epsilon}$. Sehingga total simpul yang harus dikunjungi dalam kasus terburuk adalah:

$$b^0 + b^1 + b^2 + \cdots + b^{\lfloor \frac{C}{\epsilon} \rfloor}$$

$$O(b^{(1+\lfloor \frac{C}{\epsilon} \rfloor)})$$

Dalam konteks Rush Hour, UCS bekerja dengan cara mengevaluasi semua kemungkinan gerakan dari suatu konfigurasi papan, lalu menyimpan konfigurasi-konfigurasi baru dalam *priority queue* berdasarkan jumlah total langkah yang telah dilakukan. Karena semua aksi (gerakan mobil satu langkah) memiliki *cost* seragam (*cost*-nya adalah satu), fungsi UCS menjadi sama dengan jumlah langkah dari keadaan awal ke keadaan saat ini. Kompleksitas waktu terburuk menjadi,

$$O(b^{(1+C)})$$

Kompleksitas ruang untuk algoritma UCS sama dengan kompleksitas waktu karena algoritma ini harus menyimpan semua simpul (konfigurasi papan) yang dihasilkan UCS.

Dalam konteks Rush Hour, UCS sama dengan BFS. Hal ini karena semua aksi memiliki *cost* yang sama. Setiap simpul pada pohon pencarian memiliki *cost* yang sama dengan level mereka. UCS akan menjelajahi semua simpul pada level n sebelum lanjut ke simpul $n+1$ karena *cost* pada level n lebih kecil daripada level $n+1$. Perilaku ini sama dengan perilaku BFS yang menjelajahi seluruh simpul pada satu level sebelum lanjut ke simpul pada level selanjutnya.

UCS memiliki beberapa keunggulan. UCS menjamin bahwa solusi yang ditemukan adalah solusi dengan jumlah gerakan minimum dan optimal. UCS juga tidak memerlukan fungsi heuristik, sehingga implementasinya mudah.

Namun, UCS juga memiliki beberapa kelemahan. Karena tidak menggunakan heuristik (*blind*), UCS cenderung mengeksplorasi banyak simpul dalam satu level sebelum mencapai solusi. Performa UCS memburuk secara eksponensial terhadap faktor percabangan. UCS lemah ketika menangani kasus dengan faktor percabangan tinggi.

1.3. Fungsi Heuristik

Fungsi heuristik $h(n)$ adalah estimasi cost minimum dari suatu simpul n ke tujuan. Fungsi ini berbeda dengan $g(n)$. $g(n)$ adalah cost yang sebenarnya untuk menuju simpul saat ini. Sedangkan, $h(n)$ adalah estimasi cost dari simpul saat ini ke tujuan. Fungsi heuristik yang sempurna akan sama dengan cost optimal sebenarnya dari n ke tujuan ($h^*(n)$). Namun, menghitung $h^*(n)$ biasanya sama sulitnya dengan menyelesaikan masalah asli.

Dua hal penting dalam fungsi heuristik adalah *admissibility* dan *consistency*. Sebuah fungsi heuristik dikatakan *admissible* jika estimasi cost tidak pernah melebihi cost optimal sebenarnya:

$$\forall n, h(n) \leq h^*(n)$$

Fungsi heuristik dikatakan *consistent* jika memenuhi ketidaksetaraan segitiga:

$$\forall n, \forall n', h(n) \leq c(n, n') + h(n')$$

n' adalah suksesor dari n

$c(n, n')$ adalah cost dari n ke n'

Semua heuristik yang *consistent* juga *admissible*, namun tidak sebaliknya.

Dalam konteks pemecahan masalah Rush Hour, digunakan tiga fungsi heuristik. Pertama, heuristik jarak menghitung jarak Manhattan antara posisi *primary piece* ke tujuan.

$$h_d(n) = |x_{\text{goal}} - x_{\text{primary}}|$$

atau

$$h_d(n) = |y_{\text{goal}} - y_{\text{primary}}|$$

Admissibility dan *consistency* heuristik ini terjamin. *Primary piece* harus bergerak sejauh $h_d(n)$, tidak ada gerakan yang bisa mengurangi jarak itu secara langsung, dan setiap aksi menggeser *piece* sejauh satu petak.

Kedua, heuristik *blocker* menghitung banyaknya *piece* unik yang menghalangi jalannya *primary piece*. *Admissibility* dan *consistency* heuristik ini tidak terjamin. *Piece* yang menghalangi tidak dapat dihilangkan dalam satu aksi (biasanya perlu menggerakkan *piece* lain terlebih dahulu) dan setiap aksi bisa saja mengurangi penghalang atau tidak sama sekali.

Ketiga, Heuristik gabungan menggabungkan heuristik jarak dan *blocker*, dengan harapan meningkatkan *informedness*. *Admissibility* dan *consistency* heuristik ini tidak terjamin karena heuristik ini juga menggunakan heuristik *blocker* yang tidak *admissible* dan *consistent*.

1.4. Algoritma Greedy Best First Search

Greedy Best-First Search (GBFS) adalah algoritma *informed search*. GBFS tidak mempertimbangkan *cost* sebenarnya untuk mencapai suatu simpul, melainkan hanya mempertimbangkan fungsi heuristik $h(n)$. Fungsi Evaluasi:

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

Karena mengabaikan $g(n)$ (*cost* sebenarnya dari awal ke n), GBFS bersifat serakah, memilih simpul yang tampak paling dekat ke tujuan, meskipun bisa jadi jalurnya lebih mahal secara total.

GBFS bersifat tidak lengkap karena algoritma ini bisa saja terjebak dalam traversal yang berujung pada siklus atau memilih jalur yang terlihat murah padahal mungkin saja jalur tersebut mahal. Karena GBFS tidak dapat melakukan *backtracking*, algoritma ini dapat terjebak pada jalur yang tidak mengarah ke tujuan karena mungkin saja jalur ini terlihat dekat karena fungsi heuristik.

GBFS juga tidak optimal karena algoritma ini bisa saja tertipu oleh *cost* dari fungsi heuristik. Karena tidak memperhitungkan *cost* aslinya, GBFS awalnya akan mengikuti jalur dengan nilai heuristik yang rendah, walau pada kenyataannya *total cost* bisa saja tinggi. GBFS terjebak dalam minimum lokal.

Kompleksitas waktu dari GBFS bergantung pada *branching factor* (b) dan kedalaman pohon pencarian (m). Semakin dalam pohon pencarian, simpul yang dibuat akan meningkat secara eksponensial terhadap *branching factor*. Sehingga, kompleksitas waktu terburuk GBFS adalah,

$$O(b^m)$$

Kompleksitas ruang untuk algoritma GBFS sama dengan kompleksitas waktunya karena algoritma ini harus menyimpan semua simpul yang dihasilkan.

Di dalam penyelesaian Rush Hour, setiap status atau konfigurasi papan merupakan simpul dalam ruang pencarian. GBFS bekerja dengan: menyimpan daftar status dalam *priority queue* berdasarkan $h(n)$, memilih status dengan nilai heuristik terkecil, mengekspansi semua kemungkinan langkah dari status tersebut. Dengan menggunakan heuristik jarak, GBFS lebih sederhana dan efisien, tapi bisa saja keliru karena tidak mempertimbangkan *piece* yang menghambat. Dengan menggunakan heuristik *blocker*, GBFS lebih dapat menggambarkan hambatan dan cenderung akan menyingkirkan hambatan, tapi GBFS tidak dapat *blocker* mana yang lebih mahal (*blocker* yang perlu langkah banyak untuk disingkirkan). Dengan heuristik kombinasi, GBFS berusaha menyeimbangkan informasi jarak dan *blocker*, tapi tetap saja algoritma ini tidak *complete* dan tidak optimal.

1.5. Algoritma A*

A* adalah algoritma *informed search*. Algoritma ini mengevaluasi simpul berdasarkan kombinasi antara *total cost* dari

simpul awal ke simpul yang sedang dievaluasi dan estimasi *cost* dari fungsi heuristik. Fungsi evaluasi yang digunakan oleh A* adalah:

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

A* bersifat lengkap, artinya algoritma ini akan menemukan solusi jika solusi tersebut ada, dengan syarat: *branching factor* terbatas, *cost* dari setiap aksi lebih besar dari nol, dan fungsi heuristik *admissible*.

A* menjamin solusi optimal jika fungsi heuristiknya *admissible*. Hal ini karena fungsi heuristik yang *admissible* memiliki properti $h(n) \leq h^*(n)$, sehingga $g(n) + h(n) \leq g(n) + h^*(n)$, $f(n) \leq f^*(n)$. Sehingga, A* pasti akan menghasilkan solusi optimal.

Kompleksitas waktu dari A* juga bergantung pada *branching factor* (b) dan kedalaman pohon pencarian (m). Semakin dalam pohon pencarian, simpul yang dibuat akan meningkat secara eksponensial terhadap *branching factor*. Sehingga, kompleksitas waktu terburuk A* adalah,

$$O(b^m)$$

Kompleksitas ruang untuk algoritma A* sama dengan kompleksitas waktunya karena algoritma ini harus menyimpan semua simpul yang dihasilkan.

Dalam penyelesaian permainan Rush Hour, A* menyeimbangkan dua aspek: meminimalkan jumlah langkah yang telah diambil dan mengestimasi seberapa dekat status saat ini dengan tujuan. A* bekerja dengan: menyimpan daftar status dalam *priority queue* berdasarkan $g(n) + h(n)$, memilih status dengan nilai heuristik terkecil, mengekspansi semua kemungkinan langkah dari status tersebut. Dengan menggunakan heuristik jarak, A* menjamin *completeness* dan *optimality* karena heuristik ini *admissible*. Dengan menggunakan heuristik *blocker* atau heuristik kombinasi, A* tidak menjamin *completeness* dan *optimality* karena heuristik ini tidak *admissible*.

Keunggulan dari A* adalah: menjamin solusi optimal jika heuristik *admissible* dan tidak dapat terjebak seperti GBFS. Secara teoritis, A* lebih efisien daripada UCS. UCS melakukan pencarian hanya dengan *total cost* sehingga dapat melakukan eksplorasi yang sangat luas, sedangkan A* menggunakan fungsi heuristik juga yang membuat algoritma A* cenderung melakukan eksplorasi yang mengarah ke jalur dengan heuristik yang rendah (lebih dekat dengan *goal*).

1.6. [BONUS] Algoritma IDA*

Algoritma Iterative Deepening A* (IDA*) adalah algoritma pathfinding yang menggabungkan prinsip Iterative Deepening Depth-First Search (IDDFS) dengan fungsi heuristik dari algoritma A*. Algoritma IDA* akan menemukan solusi optimal (jalur terpendek) dengan penggunaan memori yang lebih efisien dibandingkan dengan A*. Inti dari algoritma IDA* adalah penggunaan fungsi evaluasi $f(n)$ untuk setiap node dalam graf yang didefinisikan sebagai:

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

dimana $g(n)$ adalah biaya (cost) dari node awal hingga node n dan $h(n)$ adalah perkiraan biaya dari node n hingga node tujuan.

IDA* bekerja dengan melakukan pencarian Depth-First Search (DFS). Setiap iterasi DFS dibatasi oleh sebuah *threshold*. Pada iterasi pertama, *threshold* biasanya diinisialisasi dengan nilai fungsi evaluasi dari node awal, yaitu perkiraan jarak ke node tujuan ($h(n)$). Dalam setiap iterasi DFS, algoritma akan menjelajahi node-node selama nilai fungsi evaluasinya tidak melebihi *threshold* saat ini. Jika sebuah node memiliki nilai fungsi evaluasi yang melebihi *threshold*, cabang tersebut akan dipangkas (*prune*) untuk iterasi saat ini dan nilai fungsi evaluasi terkecil yang dipangkas akan digunakan sebagai *threshold* untuk iterasi berikutnya. Proses ini berlanjut hingga node tujuan ditemukan dengan nilai fungsi evaluasi yang sama atau kurang dari *threshold* saat ini.

Dengan cara ini, IDA* secara iteratif meningkatkan *threshold*, memastikan bahwa jalur dengan biaya terendah akan ditemukan terlebih dahulu. Karena menggunakan prinsip DFS, IDA* memiliki

kebutuhan memori yang jauh lebih rendah dibandingkan A*, karena hanya perlu menyimpan jalur saat ini dalam tumpukan (stack).

Dalam notasi Big O, kompleksitas waktu algoritma IDA* sangat dipengaruhi oleh kualitas fungsi heuristik. Pada kasus terburuk, tanpa heuristik yang informatif, IDA* dapat mereduksi menjadi pencarian mendalam iteratif yang memiliki kompleksitas waktu $O(b^d)$, di mana b adalah faktor percabangan rata-rata dalam ruang pencarian dan d adalah kedalaman solusi optimal. Ini karena pada setiap iterasi dengan threshold yang meningkat, algoritma mungkin harus mengunjungi kembali banyak node. Namun, jika fungsi heuristik $h(n)$ cukup baik dalam memangkas ruang pencarian, kinerja IDA* bisa jauh lebih baik. Idealnya, jika heuristik sempurna, IDA* hanya akan menjelajahi jalur terpendek. Secara umum, kompleksitas waktu IDA* sulit untuk dinyatakan dalam notasi Big O yang sederhana yang berlaku untuk semua kasus, karena sangat bergantung pada seberapa efektif heuristik memangkas pencarian pada setiap threshold. Meskipun demikian, penting untuk menyadari bahwa dalam skenario terburuk, kompleksitasnya tetap eksponensial terhadap kedalaman solusi.

BAB II

IMPLEMENTASI ALGORITMA DALAM BAHASA JAVA

2.1. Implementasi Fungsi Heuristik Jarak

Heuristik jarak bertujuan untuk memberikan estimasi minimum jumlah langkah yang diperlukan oleh primary piece (mobil berlabel 'P') untuk mencapai goal dalam permainan Rush Hour. Hal ini dilakukan tanpa memperhitungkan kendaraan yang menghalangi jalan, hanya fokus pada jarak geometris langsung ke goal di sepanjang sumbu pergerakan utama dari primary piece.

Algoritma ini dapat dipahami dalam beberapa langkah utama:

1. Identifikasi Primary Piece

Setiap keadaan permainan (state) memuat informasi tentang posisi seluruh kendaraan di grid. Algoritma mengecek daftar kendaraan untuk menemukan primary piece dengan ID khusus ('P'). Hanya ada satu primary piece.



2. Tentukan Lokasi Goal

Rush Hour mengizinkan konfigurasi goal pada salah satu dari empat sisi papan: kanan, kiri, atas, atau bawah. Posisi goal ini menjadi acuan utama dalam menentukan arah gerak yang dibutuhkan oleh primary piece.



3. Estimasi Jarak

Karena kendaraan hanya dapat bergerak di sepanjang orientasinya (horizontal atau vertikal), estimasi jarak dilakukan hanya pada satu sumbu. Untuk kendaraan horizontal dan goal di kiri/kanan, hanya sumbu-x yang relevan. Untuk kendaraan

vertikal dan goal di atas/bawah, hanya sumbu-y yang relevan. Secara umum, algoritma mencari: RIGHT goal → jumlah sel dari sisi kanan kendaraan ke ujung kanan papan. LEFT goal → jumlah sel dari sisi kiri kendaraan ke ujung kiri papan. BOTTOM goal → jumlah sel dari sisi bawah kendaraan ke ujung bawah papan. TOP goal → jumlah sel dari sisi atas kendaraan ke ujung atas papan.

```
● ○ ●  
1 switch (goalSide) {  
2     case RIGHT → distanceToGoal = grid[0].length - 1 - x;  
3     case LEFT → distanceToGoal = x;  
4     case BOTTOM → distanceToGoal = grid.length - 1 - y;  
5     case TOP → distanceToGoal = y;  
6 }
```

2.2. Implementasi Fungsi Heuristik Blocker

Heuristik *Blocker* bertujuan memberikan estimasi jumlah kendaraan unik yang menghalangi secara langsung jalur dari *primary piece* menuju *goal*. Fokus heuristik *blocker* bukan pada jumlah langkah, melainkan pada jumlah *piece* yang harus dimanipulasi, menjadikannya sangat relevan untuk papan yang padat (*congested*).

Algoritma ini dapat dipahami dalam beberapa langkah utama:

1. Identifikasi Primary Piece

Sama seperti heuristik jarak, langkah pertama adalah mengidentifikasi kendaraan utama 'P' dalam keadaan saat ini. Posisi kendaraan ini menentukan titik awal jalur yang akan dianalisis.

```
● ○ ●  
1 GamePiece primaryPiece = getPrimaryPiece(state);
```

2. Tentukan Arah Jalur ke Goal

Posisi goal bisa berada di salah satu dari empat sisi papan: kanan, kiri, atas, atau bawah. Berdasarkan ini, ditentukan arah

iterasi. Horizontal (kiri-kanan) untuk kendaraan horizontal. Vertikal (atas-bawah) untuk kendaraan vertikal.



```
1 GoalPlacement goalSide = state.getBoard().getGoalPlacement();
```

3. Iterasi Sel-Sel pada Jalur Menuju Goal

Setelah mengetahui orientasi dan arah goal, algoritma melakukan iterasi linear dari ujung kendaraan utama ke arah goal. Setiap sel akan diperiksa apakah sel tersebut: kosong (bukan penghalang), Berisi primary piece (diabaikan), atau berisi

kendaraan lain (*blocker*).

```
● ○ ●
1 switch (goalSide) {
2     case RIGHT → {
3         for (int col = x + 1; col < grid[0].length; col++) {
4             char cell = grid[y][col];
5             if (cell ≠ '.' && cell ≠ 'P') {
6                 blockers.add(cell);
7             }
8         }
9     }
10    case LEFT → {
11        for (int col = x - 1; col ≥ 0; col--) {
12            char cell = grid[y][col];
13            if (cell ≠ '.' && cell ≠ 'P') {
14                blockers.add(cell);
15            }
16        }
17    }
18    case BOTTOM → {
19        for (int row = y + 1; row < grid.length; row++) {
20            char cell = grid[row][x];
21            if (cell ≠ '.' && cell ≠ 'P') {
22                blockers.add(cell);
23            }
24        }
25    }
26    case TOP → {
27        for (int row = y - 1; row ≥ 0; row--) {
28            char cell = grid[row][x];
29            if (cell ≠ '.' && cell ≠ 'P') {
30                blockers.add(cell);
31            }
32        }
33    }
34 }
```

4. Menyimpan Penghalang

Kendaraan yang terdeteksi sebagai penghalang dicatat dalam sebuah set koleksi unik agar satu kendaraan hanya dihitung satu

kali, meskipun menghalangi dalam lebih dari satu sel.



```
1 Set<Character> blockers = new HashSet<>();
```

2.3. Implementasi Fungsi Heuristik Kombinasi

Fungsi heuristik kombinasi merupakan fungsi heuristik yang menjumlahkan perhitungan heuristik jarak dan *blocker*. Implementasinya sangat sederhana karena cukup menggabungkan kedua fungsi heuristik di atas.



```
1 public int calculate(GameState state) {  
2     return distanceHeuristic.calculate(state) + blockerCountHeuristic.calculate(state);  
3 }
```

2.4. Implementasi UCS, GBFS, A*

Algoritma StandardSearch adalah kerangka (*framework*) untuk pencarian berbasis *priority queue*, dan dapat dikonfigurasi menjadi UCS, GBFS, atau A* hanya dengan mengganti strategi komparator (*state ordering function*).

Secara umum, algoritma ini bekerja sebagai berikut:

1. Inisialisasi

Mulai dari simpul awal (*initial state*). Buat *priority queue* yang menyimpan state yang akan diperluas, dengan urutan ditentukan oleh fungsi perbandingan (*comparator*). Simpan himpunan visited untuk menghindari eksplorasi ulang pada *state*.

yang sama.

```
● ● ●  
1 PriorityQueue<GameState> queue = new PriorityQueue<>(comparator);  
2 Set<String> visited = new HashSet<>();  
3 nodesExplored = 0;  
4  
5 queue.add(initialState);
```

2. Perulangan Ekspansi

Ambil state dengan prioritas tertinggi dari *queue*. Periksa apakah ini adalah state goal. Jika ya, lakukan rekonstruksi jalur solusi (*path reconstruction*) menggunakan jejak parent. Jika tidak, hasilkan semua state anak (*successors*), dan tambahkan ke *queue* jika belum pernah dikunjungi.

```
● ● ●  
1 while (!queue.isEmpty()) {  
2     nodesExplored++;  
3     GameState current = queue.poll();  
4  
5     if (visited.contains(current.getBoard().toString())) {  
6         continue;  
7     }  
8  
9     visited.add(current.getBoard().toString());  
10    if (current.isGoal()) {  
11        return reconstructPath(current);  
12    }  
13  
14    for (GameState neighbor : current.generateSuccessors()) {  
15        if (!visited.contains(neighbor.getBoard().toString())) {  
16            queue.add(neighbor);  
17        }  
18    }  
19}  
20 }
```

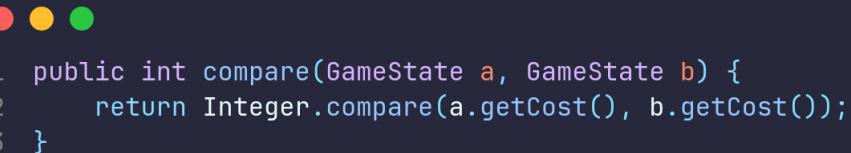
3. Terminasi

Jika *queue* kosong dan *goal* belum ditemukan, berarti tidak ada solusi.



Setiap algoritma pencarian didefinisikan oleh bagaimana algoritma tersebut mengurutkan simpul-simpul dalam *queue*. Hal ini dikontrol oleh fungsi komparator.

Implementasi UCS menggunakan komparator berdasarkan biaya kumulatif aktual dari start ke suatu state (*cost*). UCS akan selalu memilih state dengan biaya minimum dari start.



Fungsi *getCost* sendiri dihitung dengan *cost parent + 1* (Karena setiap *cost* bertambah 1 untuk simpul anak).



Implementasi GBFS menggunakan hanya nilai heuristik $h(n)$, yaitu estimasi biaya ke *goal* dari state sekarang. Perhitungan nilai heuristik berdasarkan fungsi heuristik yang dipilih.

```
● ● ●  
1 public int compare(GameState a, GameState b) {  
2     return Integer.compare(  
3         heuristic.calculate(a),  
4         heuristic.calculate(b)  
5     );  
6 }
```

Implementasi A* menggunakan komparator nilai heuristik $h(n)$ dan biaya kumulatif aktual. Perhitungan nilai heuristik berdasarkan fungsi heuristik yang dipilih.

```
● ● ●  
1 public int compare(GameState a, GameState b) {  
2     return Integer.compare(  
3         a.getCost() + heuristic.calculate(a),  
4         b.getCost() + heuristic.calculate(b)  
5     );  
6 }
```

2.5. Implementasi IDA*

Dalam implementasi algoritma IDA* pada permasalahan Rush Hour, penulis mengadopsi pendekatan iteratif mendalam (DFS) yang dipandu oleh fungsi heuristik untuk menemukan urutan gerakan minimal yang mengarah pada solusi, sesuai dengan ketentuan algoritma IDA*. Algoritma bekerja sebagai berikut:

Algoritma dimulai dengan menetapkan threshold awal berdasarkan nilai heuristik dari kondisi awal papan permainan.

```
● ● ●  
1 public List<GameState> solve(GameState initialState) {  
2     nodesExplored = 0;  
3     int threshold = heuristic.calculate(initialState);  
4  
5     while (true) {  
6         visited = new HashMap<>();  
7         List<GameState> initialPath = new ArrayList<>();  
8         initialPath.add(initialState);  
9         Result result = search(initialState, threshold, initialPath);
```

Kemudian, serangkaian pencarian Depth-First Search (DFS) dilakukan, setiap pencarian dibatasi oleh threshold saat ini. Dalam setiap iterasi DFS, algoritma mengeksplorasi state-state permainan dengan menghitung nilai fungsi evaluasi ($f(n)$).

```
● ● ●  
1 // Recursive DFS  
2 int nextThreshold = Integer.MAX_VALUE;  
3 List<GameState> successors = state.generateSuccessors();  
4  
5 for (GameState successor : successors) {  
6     path.add(successor);  
7     Result result = search(successor, threshold, path);  
8  
9     if (result.found) {  
10        return result;  
11    }  
12  
13    path.remove(path.size() - 1);  
14    nextThreshold = Math.min(nextThreshold, result.nextThreshold);  
15 }
```

Jika nilai $f(n)$ dari suatu state melebihi threshold saat ini, cabang pencarian tersebut dipangkas, dan nilai $f(n)$ terkecil yang melampaui threshold dicatat untuk digunakan sebagai threshold pada iterasi berikutnya.

```
● ● ●  
1 private Result search(GameState state, int threshold, List<GameState> path) {  
2     nodesExplored++;  
3  
4     int f = state.getCost() + heuristic.calculate(state);  
5  
6     if (f > threshold) {  
7         return new Result(false, null, f);  
8     }  
9 }
```

Proses ini berlanjut dengan peningkatan threshold secara bertahap hingga solusi ditemukan, yaitu ketika sebuah state tujuan tercapai dengan nilai f yang tidak melebihi threshold saat ini. Untuk menghindari eksplorasi state yang berulang dalam satu iterasi DFS, sebuah mekanisme pelacakan state yang telah dikunjungi (visited) dengan biaya yang lebih rendah atau sama diimplementasikan.

```
● ● ●  
1 String boardKey = state.getBoard().toString();  
2 Integer previousCost = visited.get(boardKey);  
3 if (previousCost != null && previousCost <= state.getCost()) {  
4     return new Result(false, null, threshold + 1);  
5 }
```

Algoritma mengembalikan urutan state (`List<GameState>`) yang mengarah dari kondisi awal ke kondisi tujuan setelah solusi ditemukan, atau list kosong jika tidak ada solusi yang ditemukan setelah semua kemungkinan threshold dieksplorasi. Jumlah total node yang dieksplorasi selama proses pencarian juga dilacak.

BAB III

SOURCE CODE PROGRAM

3.1. Repository Program

Repository Program dapat diakses melalui tautan Github berikut:
https://github.com/kin-ark/Tucil3_13523146_13523152

3.2. Source Code Algoritma Pathfinding

3.2.1. InputReader.java

Kelas ini berperan dalam menangani pembacaan input dari file konfigurasi permainan.

Attributes & Constructor

```
● ● ●

1 public class InputReader {
2     public int A, B, N;
3     public char[][] board;
4     public char primaryPiece = 'P';
5     public char goalPiece = 'K';
6     public List<List<int[]>> puzzlePieces;
7     public char[] ids;
8
9     private final List<String> rawBoardLines;
10    public GameEnums.GoalPlacement goalPlacement;
11    public int goalIndex;
12    private boolean isGoalFound;
13    private int startA, endA, startB, endB;
14
15    public InputReader(String fileName) {
16        puzzlePieces = new ArrayList<>();
17        rawBoardLines = new ArrayList<>();
18        readInputFile(fileName);
19    }
```

Method void readInputFile(String fileName)

```
1 private void readInputFile(String fileName) {
2     try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
3         String[] firstLine = br.readLine().split(" ");
4         if (firstLine.length != 2) {
5             throw new Error("Invalid first line: Expected format 'A B'");
6         }
7
8         // Read A and B (Rows and Columns)
9         A = parsePositiveInt(firstLine[0], "Invalid row count (A)");
10        B = parsePositiveInt(firstLine[1], "Invalid column count (B)");
11
12        // Read N (Number of Pieces)
13        String pLine = br.readLine();
14        if (pLine == null) throw new Error("Missing piece count (N)");
15        N = parsePositiveInt(pLine.trim(), "Invalid piece count (N)");
16
17        // Scan for goal position while get the raw board lines
18        for (int i = 0; i <= A; i++) {
19            String line = br.readLine();
20            if (line == null) {
21                if (!isGoalFound && i != A) {
22                    throw new Error("Unexpected end of file when reading board.");
23                }
24                break;
25            }
26
27            if (line.length() < B || line.length() > B + 1) {
28                throw new Error("Invalid line length at row " + i + ": must be " + B + " or " + (B + 1) + " characters.");
29            }
30
31            rawBoardLines.add(line);

```

```

● ● ●
1  for (int j = 0; j < line.length(); j++) {
2      char c = line.charAt(j);
3      if (c == goalPiece) {
4          if (isGoalFound) {
5              throw new Error("Multiple goal positions ('K') found.");
6          }
7
8          if (i == 0 && j < B) { // TOP border
9              goalPlacement = GameEnums.GoalPlacement.TOP;
10             goalIndex = j;
11             isGoalFound = true;
12         }
13         else if (j == 0 && i < A) { // LEFT border
14             goalPlacement = GameEnums.GoalPlacement.LEFT;
15             goalIndex = i;
16             isGoalFound = true;
17         }
18         else if (j == B && i < A) { // RIGHT border
19             goalPlacement = GameEnums.GoalPlacement.RIGHT;
20             goalIndex = i;
21             isGoalFound = true;
22         }
23         else if (i == A && j < B) { // BOTTOM border
24             goalPlacement = GameEnums.GoalPlacement.BOTTOM;
25             goalIndex = j;
26             isGoalFound = true;
27         }
28         else {
29             throw new Error("Goal piece ('K') must be on the border.");
30         }
31     }
32 }
33
34
35 if (goalPlacement == GameEnums.GoalPlacement.TOP && rawBoardLines.size() != A + 1) {
36     goalPlacement = GameEnums.GoalPlacement.LEFT;
37 }
38
39 if (!isGoalFound) {
40     throw new Error("No goal position ('K') found.");
41 }
42
43 switch (goalPlacement) {
44     case LEFT:
45         startA = 0;
46         endA = A;
47         startB = 1;
48         endB = B + 1;
49         break;
50     case TOP:
51         startA = 1;
52         endA = A + 1;
53         startB = 0;
54         endB = B;
55         break;
56     case RIGHT:
57     case BOTTOM:
58     default:
59         startA = 0;
60         endA = A;
61         startB = 0;
62         endB = B;
63         break;
64 }

```

```

1 board = new char[A][B];
2         for (int i = 0; i < A; i++) {
3             for (int j = 0; j < B; j++) {
4                 board[i][j] = '.';
5             }
6         }
7
8     Map<Character, List<int[]>> pieceMap = new LinkedHashMap<>();
9
10    for (int i = startA; i < endA; i++) {
11        String line = rawBoardLines.get(i);
12        for (int j = startB; j < endB && j < line.length(); j++) {
13            char c = line.charAt(j);
14
15            if (c == '.' || c == goalPiece) continue;
16
17            int boardRow = i;
18            int boardCol = j;
19
20            switch (goalPlacement) {
21                case LEFT:
22                    boardCol = j - 1;
23                    break;
24                case TOP:
25                    boardRow = i - 1;
26                    break;
27                case RIGHT:
28                case BOTTOM:
29                default:
30                    break;
31            }
32
33            if (boardRow < 0 || boardRow >= A || boardCol < 0 || boardCol >= B) continue;
34
35            board[boardRow][boardCol] = c;
36
37            pieceMap.putIfAbsent(c, new ArrayList<>());
38            pieceMap.get(c).add(new int[]{boardCol, boardRow});
39        }
40    }
41
42    ids = new char[pieceMap.size()];
43    puzzlePieces = new ArrayList<>();
44
45    int idx = 0;
46    for (Map.Entry<Character, List<int[]>> entry : pieceMap.entrySet()) {
47        ids[idx] = entry.getKey();
48        puzzlePieces.add(entry.getValue());
49        idx++;
50    }
51
52    boolean foundPrimary = false;
53    for (char id : ids) {
54        if (id == primaryPiece) {
55            foundPrimary = true;
56            break;
57        }
58    }
59    if (!foundPrimary && ids.length > 0) {
60        throw new Error("No Primary Piece ('P') detected!");
61    }
62    checkPrimaryPieceAlignment();
63 } catch (IOException e) {
64     throw new Error("Error reading file: " + e.getMessage());
65 }
66 }
```

Method int parsePositiveInt(String value, String errorMessage)

```
 1 private int parsePositiveInt(String value, String errorMessage) {  
 2     try {  
 3         int number = Integer.parseInt(value);  
 4         if (number ≤ 0) {  
 5             throw new Error(errorMessage + " must be a positive integer.");  
 6         }  
 7         return number;  
 8     } catch (NumberFormatException e) {  
 9         throw new Error(errorMessage + " is not a valid integer.");  
10    }  
11 }
```

Method void checkPrimaryPieceAlignment()

```
● ● ●
```

```
1 private void checkPrimaryPieceAlignment() {
2     if (!isGoalFound || ids == null) return;
3
4     List<int[]> primaryCoords = null;
5     for (int i = 0; i < ids.length; i++) {
6         if (ids[i] == primaryPiece) {
7             primaryCoords = puzzlePieces.get(i);
8             break;
9         }
10    }
11
12    if (primaryCoords == null || primaryCoords.isEmpty()) {
13        throw new Error("Primary piece '" + primaryPiece + "' has no coordinates");
14    }
15
16    boolean isAligned = true;
17    switch (goalPlacement) {
18        case LEFT, RIGHT → {
19            for (int[] coord : primaryCoords) {
20                if (coord[1] ≠ goalIndex) {
21                    isAligned = false;
22                    break;
23                }
24            }
25        }
26        case TOP, BOTTOM → {
27            for (int[] coord : primaryCoords) {
28                if (coord[0] ≠ goalIndex) {
29                    isAligned = false;
30                    break;
31                }
32            }
33        }
34    }
35
36    if (!isAligned) {
37        throw new Error("Primary piece '" + primaryPiece + "' is not aligned with goal position");
38    }
39 }
```

3.2.2. GameSolverGUI.java

Kelas ini berperan dalam tampilan GUI dari program. Kelas ini menggunakan library JavaFX.

Attributes & Constructor



```
1 public class GameSolverGUI extends Application {  
2     // Color palette  
3     private static final Color BACKGROUND_COLOR = Color.web("#2E3440");  
4     private static final Color PRIMARY_COLOR = Color.web("#5E81AC");  
5     private static final Color PRIMARY_LIGHT = Color.web("#88C0D0");  
6     private static final Color TEXT_COLOR = Color.web("#ECEFF4");  
7     private static final Color EMPTY_CELL = Color.web("#3B4252");  
8     private static final Color PLAYER_CELL = Color.web("#A3BE8C");  
9     private static final Color OTHER_CELL = Color.web("#BF616A");  
10  
11     private Stage primaryStage;  
12     private File selectedFile;  
13     private ComboBox<String> algorithmComboBox;  
14     private GridPane boardDisplay;  
15     private List<GameState> solutionPath;  
16     private long solveTime;  
17     private int rows, cols;  
18     private int nodesExplored = 0;  
19     private Label loadingLabel;  
20     private Timeline loadingAnimation;  
21     private Timeline solutionAnimation;  
22     private final AtomicInteger currentStepIndex = new AtomicInteger(0);  
23     private Slider speedSlider;
```

Method void start(Stage primaryStage) (override)

```
1 @Override
2 public void start(Stage primaryStage) {
3     this.primaryStage = primaryStage;
4     primaryStage.setTitle("Rush Hour Solver");
5     primaryStage.setWidth(800);
6     primaryStage.setHeight(600);
7
8     showMainMenu();
9
10    primaryStage.show();
11 }
```

Method void showMainMenu()

Method untuk menu utama dan landing page user. Digunakan untuk memilih algoritma dan input file konfigurasi permainan.



```
1 private void showMainMenu() {
2     VBox root = new VBox(30);
3     root.setPadding(new Insets(40));
4     root.setAlignment(Pos.CENTER);
5     root.setStyle("-fx-background-color: " + toHexString(BACKGROUND_COLOR) + ";");
6
7     Label titleLabel = new Label("Rush Hour Solver");
8     titleLabel.setFont(Font.font("Poppins", FontWeight.BOLD, 32));
9     titleLabel.setTextFill(PRIMARY_COLOR);
10
11    // Ini kalo mau ada deskripsi aja sih
12    Label descLabel = new Label("Rush Hour Solver with Pathfinding Algorithm");
13    descLabel.setFont(Font.font("Poppins", FontWeight.NORMAL, 14));
14    descLabel.setTextFill(TEXT_COLOR);
15
16    // Milih algoritma ganti dropdown lah ya
17    HBox algorithmBox = new HBox(15);
18    algorithmBox.setAlignment(Pos.CENTER);
19
20    Label algoLabel = new Label("Algorithm:");
21    algoLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 16));
22    algoLabel.setTextFill(TEXT_COLOR);
23
24    algorithmComboBox = new ComboBox();
25    algorithmComboBox.getItems().addAll("UCS", "Greedy Best First", "A*");
26    algorithmComboBox.setValue("UCS");
27    algorithmComboBox.setStyle("-fx-font-family: 'Poppins'; -fx-font-size: 14px;");
28    algorithmComboBox.setPrefWidth(200);
29
30    algorithmBox.getChildren().addAll(algoLabel, algorithmComboBox);
31
32    // File selection button
33    Button selectFileButton = new Button("Select Puzzle File");
34    styleButton(selectFileButton);
35    selectFileButton.setOnAction(e → openFileChooser());
36
37    root.getChildren().addAll(titleLabel, descLabel, algorithmBox, selectFileButton);
38
39    Scene scene = new Scene(root);
40    primaryStage.setScene(scene);
41 }
```

Method void openFileChooser()

```
● ● ●  
1 private void openFileChooser() {  
2     FileChooser fileChooser = new FileChooser();  
3     fileChooser.setTitle("Open Puzzle File");  
4     fileChooser.getExtensionFilters().add(  
5         new FileChooser.ExtensionFilter("Text Files", "*.txt")  
6     );  
7  
8     fileChooser.setInitialDirectory(new File(System.getProperty("user.dir")));  
9  
10    selectedFile = fileChooser.showOpenDialog(primaryStage);  
11  
12    if (selectedFile != null) {  
13        showLoadingScreen();  
14    }  
15 }
```

Method void showLoadingScreen()

Method untuk menampilkan tampilan loading ketika program sedang menjalankan algoritma solver.

```
● ● ●

1 private void showLoadingScreen() {
2     BorderPane root = new BorderPane();
3     root.setPadding(new Insets(40));
4     root.setStyle("-fx-background-color: " + toHexString(BACKGROUND_COLOR) + ";");
5
6     Label statusLabel = new Label("Solving puzzle...");
7     statusLabel.setFont(Font.font("Poppins", FontWeight.BOLD, 20));
8     statusLabel.setTextFill(TEXT_COLOR);
9     statusLabel.setAlignment(Pos.CENTER);
10
11    VBox topBox = new VBox(10);
12    topBox.setAlignment(Pos.CENTER);
13    Label fileNameLabel = new Label("File: " + selectedFile.getName());
14    fileNameLabel.setFont(Font.font("Poppins", FontWeight.NORMAL, 14));
15    fileNameLabel.setTextFill(TEXT_COLOR);
16
17    String selectedAlgorithm = algorithmComboBox.getValue();
18    Label algoLabel = new Label("Algorithm: " + selectedAlgorithm);
19    algoLabel.setFont(Font.font("Poppins", FontWeight.NORMAL, 14));
20    algoLabel.setTextFill(TEXT_COLOR);
21
22    topBox.getChildren().addAll(statusLabel, fileNameLabel, algoLabel);
23
24    loadingLabel = new Label(":");
25    loadingLabel.setFont(Font.font("Monospaced", FontWeight.BOLD, 60));
26    loadingLabel.setTextFill(PRIMARY_COLOR);
27
28    StackPane centerPane = new StackPane(loadingLabel);
29    centerPane.setAlignment(Pos.CENTER);
30
31    Button cancelButton = new Button("Cancel");
32    styleButton(cancelButton);
33    cancelButton.setOnAction(e → showMainMenu());
34
35    StackPane bottomPane = new StackPane(cancelButton);
36    bottomPane.setAlignment(Pos.CENTER);
37    bottomPane.setPadding(new Insets(20, 0, 0, 0));
38
39    root.setTop(topBox);
40    root.setCenter(centerPane);
41    root.setBottom(bottomPane);
42    BorderPane.setAlignment(topBox, Pos.CENTER);
43
44    Scene scene = new Scene(root);
45    primaryStage.setScene(scene);
46
47    startLoadingAnimation();
48    CompletableFuture.runAsync(this::runSolver);
49 }
```



```
1 private void runSolver() {
2     try {
3         InputReader reader = new InputReader(selectedFile.getAbsolutePath());
4         rows = reader.A;
5         cols = reader.B;
6
7         String selectedAlgorithm = algorithmComboBox.getValue();
8
9         List<GamePiece> pieces = new ArrayList<>();
10        for (int i = 0; i < reader.ids.length; i++) {
11            pieces.add(new GamePiece(reader.ids[i], reader.puzzlePieces.get(i)));
12        }
13
14        GameBoard initialBoard = new GameBoard(reader.A, reader.B, reader.goalPlacement, reader.goalIndex);
15        initialBoard.placePieces(pieces);
16
17        GameState initialState = new GameState(initialBoard, pieces, null, "Start");
18
19        long startTime = System.currentTimeMillis();
20        GameSolver.SolverResult result = GameSolver.solve(initialState, selectedAlgorithm);
21        solutionPath = result.getPath();
22        long endTime = System.currentTimeMillis();
23        solveTime = endTime - startTime;
24
25        nodesExplored = result.getNodesExplored();
26
27        Platform.runLater(() -> {
28            if (solutionPath.isEmpty()) {
29                showNoSolutionScreen();
30            } else {
31                showSolutionScreen();
32            }
33        });
34    } catch (Exception e) {
35        Platform.runLater(() -> showErrorDialog("Exception: " + e.getMessage()));
36    } catch (Error e) {
37        Platform.runLater(() -> showErrorDialog("Error: " + e.getMessage()));
38    }
39 }
```

Method void showNoSolutionScreen()

Tampilan ketika konfigurasi permainan tidak memiliki solusi.

```
● ● ●
1 private void showNoSolutionScreen() {
2     if (loadingAnimation != null) {
3         loadingAnimation.stop();
4     }
5
6     BorderPane root = new BorderPane();
7     root.setPadding(new Insets(30));
8     root.setStyle("-fx-background-color: " + toHexString(BACKGROUND_COLOR) + ";");
9
10    VBox centerBox = new VBox(20);
11    centerBox.setAlignment(Pos.CENTER);
12
13    Label resultLabel = new Label("No Solution Found!");
14    resultLabel.setFont(Font.font("Poppins", FontWeight.BOLD, 24));
15    resultLabel.setTextFill(PRIMARY_COLOR);
16
17    Label explainLabel = new Label("This puzzle has no valid solution.");
18    explainLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 16));
19    explainLabel.setTextFill(TEXT_COLOR);
20
21    Label nodesExploredLabel = new Label("Nodes explored: " + nodesExplored);
22    nodesExploredLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 14));
23    nodesExploredLabel.setTextFill(TEXT_COLOR);
24
25    Label timeLabel = new Label("Time spent: " + solveTime + " ms");
26    timeLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 14));
27    timeLabel.setTextFill(TEXT_COLOR);
28
29    Button backButton = new Button("Back to Menu");
30    styleButton(backButton);
31    backButton.setOnAction(e → {
32        showMainMenu();
33        nodesExplored = 0;
34    });
35
36    centerBox.getChildren().addAll(
37        resultLabel,
38        explainLabel,
39        nodesExploredLabel,
40        timeLabel,
41        backButton
42    );
43
44    root.setCenter(centerBox);
45
46    Scene scene = new Scene(root);
47    primaryStage.setScene(scene);
48 }
```

Method void showSolutionScreen()

```

1  private void showSolutionScreen() {
2      if (loadingAnimation != null) {
3          loadingAnimation.stop();
4      }
5
6      BorderPane root = new BorderPane();
7      root.setPadding(new Insets(30));
8      root.setStyle("-fx-background-color: " + toHexString(BACKGROUND_COLOR) + ";");
9
10     VBox statsBox = new VBox(12);
11     statsBox.setAlignment(Pos.CENTER);
12
13     Label resultLabel = new Label("Solution Found!");
14     resultLabel.setFont(Font.font("Poppins", FontWeight.BOLD, 24));
15     resultLabel.setTextFill(PRIMARY_COLOR);
16
17     HBox statsPane = new HBox(30);
18     statsPane.setAlignment(Pos.CENTER);
19
20     Label timeLabel = new Label("Time: " + solveTime + " ms");
21     timeLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 14));
22     timeLabel.setTextFill(TEXT_COLOR);
23
24     Label nodesLabel = new Label("Nodes: " + nodesExplored);
25     nodesLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 14));
26     nodesLabel.setTextFill(TEXT_COLOR);
27
28     Label stepsLabel = new Label("Steps: " + (solutionPath.size() - 1));
29     stepsLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 14));
30     stepsLabel.setTextFill(TEXT_COLOR);
31
32     statsPane.getChildren().addAll(timeLabel, nodesLabel, stepsLabel);
33     statsBox.getChildren().addAll(resultLabel, statsPane);
34
35     StackPane boardContainer = new StackPane();
36     boardContainer.setPadding(new Insets(20));
37     boardContainer.setStyle(
38         "-fx-background-color: " + toHexString(PRIMARY_COLOR) + ";" +
39         "-fx-border-color: " + toHexString(PRIMARY_LIGHT) + ";" +
40         "-fx-border-width: 2;" +
41         "-fx-border-radius: 5;" +
42     );
43
44     boardDisplay = new GridPane();
45     boardDisplay.setAlignment(Pos.CENTER);
46     boardDisplay.setHgap(2);
47     boardDisplay.setVgap(2);
48
49     currentStepIndex.set(0);
50     updateBoard(solutionPath.get(0));
51     boardContainer.getChildren().add(boardDisplay);
52
53     HBox controlsBox = new HBox(15);
54     controlsBox.setAlignment(Pos.CENTER);
55     controlsBox.setPadding(new Insets(20, 0, 0, 0));
56
57     Label stepIndicator = new Label("Step: 0/" + (solutionPath.size() - 1));
58     stepIndicator.setFont(Font.font("Poppins", FontWeight.MEDIUM, 14));
59     stepIndicator.setTextFill(TEXT_COLOR);
60
61     Label moveDescriptionLabel = new Label("Initial position");
62     moveDescriptionLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 14));
63     moveDescriptionLabel.setTextFill(TEXT_COLOR);
64     moveDescriptionLabel.setPrefWidth(200);
65     moveDescriptionLabel.setAlignment(Pos.CENTER);

```

```

1   Button playPauseButton = new Button("▶ Play");
2   styleButton(playPauseButton);
3   playPauseButton.setUserData(false);
4   playPauseButton.setOnAction(e → {
5     boolean isPlaying = (boolean) playPauseButton.getUserData();
6     if (isPlaying) {
7       stopAnimation();
8       playPauseButton.setText("▶ Play");
9       playPauseButton.setUserData(false);
10    } else {
11      playAnimation(stepIndicator, moveDescriptionLabel, playPauseButton);
12      playPauseButton.setText("⏸ Pause");
13      playPauseButton.setUserData(true);
14    }
15  });
16
17 Button prevButton = new Button("◀");
18 styleButton(prevButton);
19 prevButton.setOnAction(e → {
20   stopAnimation();
21   int index = currentStepIndex.decrementAndGet();
22   if (index < 0) {
23     currentStepIndex.set(0);
24     index = 0;
25   }
26   updateBoard(solutionPath.get(index));
27   updateStepInfo(stepIndicator, moveDescriptionLabel, index);
28   playPauseButton.setText("▶ Play");
29   playPauseButton.setUserData(false);
30   playPauseButton.setDisable(false);
31 });
32
33 Button nextButton = new Button("▶");
34 styleButton(nextButton);
35 nextButton.setOnAction(e → {
36   stopAnimation();
37   int index = currentStepIndex.incrementAndGet();
38   if (index ≥ solutionPath.size()) {
39     currentStepIndex.set(solutionPath.size() - 1);
40     index = solutionPath.size() - 1;
41   }
42   updateBoard(solutionPath.get(index));
43   updateStepInfo(stepIndicator, moveDescriptionLabel, index);
44   playPauseButton.setText("▶ Play");
45   playPauseButton.setUserData(false);
46   playPauseButton.setDisable(false);
47 });
48
49 Label speedLabel = new Label("Speed:");
50 speedLabel.setFont(Font.font("Poppins", FontWeight.MEDIUM, 14));
51 speedLabel.setTextFill(TEXT_COLOR);
52
53 Slider speedSlider = new Slider(0.5, 3, 1);
54 speedSlider.setPrefWidth(100);
55 speedSlider.setShowTickMarks(true);
56 speedSlider.setShowTickLabels(true);
57 speedSlider.setMajorTickUnit(0.5);
58
59 Button resetButton = new Button("↺ Reset");
60 styleButton(resetButton);
61 resetButton.setOnAction(e → {
62   stopAnimation();
63   currentStepIndex.set(0);
64   updateBoard(solutionPath.get(0));
65   updateStepInfo(stepIndicator, moveDescriptionLabel, 0);
66   playPauseButton.setText("▶ Play");
67   playPauseButton.setUserData(false);
68   playPauseButton.setDisable(false);
69 });
70
71 Button backButton = new Button("← Back");
72 styleButton(backButton);
73 backButton.setOnAction(e → {
74   stopAnimation();
75   showMainMenu();
76   nodesExplored = 0;
77 });
78
79 HBox navigationBox = new HBox(10, prevButton, playPauseButton, nextButton);
80 navigationBox.setAlignment(Pos.CENTER);
81
82 controlsBox.getChildren().addAll(
83   new VBox(5, speedLabel, speedSlider),
84   new VBox(5, stepIndicator, moveDescriptionLabel),
85   navigationBox,
86   resetButton,
87   backButton
88 );
89
90 root.setTop(statsBox);
91 root.setCenter(boardContainer);
92 root.setBottom(controlsBox);
93
94 Scene scene = new Scene(root);
95 primaryStage.setScene(scene);
96 }

```

Method playAnimation & stopAnimation

Kedua method ini digunakan untuk tampilan animasi board state dari awal hingga mencapai solusi.

```
1 private void playAnimation(Label stepIndicator, Label moveDescLabel, Button playPauseButton) {  
2     stopAnimation();  
3  
4     solutionAnimation = new Timeline();  
5     double speed = speedSlider.getValue();  
6  
7     KeyFrame keyFrame = new KeyFrame(Duration.millis(500 / speed), e → {  
8         int index = currentStepIndex.getAndIncrement();  
9         if (index < solutionPath.size()) {  
10             updateBoard(solutionPath.get(index));  
11             updateStepInfo(stepIndicator, moveDescLabel, index);  
12         } else {  
13             stopAnimation();  
14             playPauseButton.setText("▶ Play");  
15             playPauseButton.setUserData(false);  
16             playPauseButton.setDisable(true);  
17         }  
18     });  
19  
20     solutionAnimation.getKeyFrames().add(keyFrame);  
21     solutionAnimation.setCycleCount(Timeline.INDEFINITE);  
22     solutionAnimation.play();  
23 }  
24  
25 private void stopAnimation() {  
26     if (solutionAnimation != null) {  
27         solutionAnimation.stop();  
28     }  
29 }
```

Method Update

Method-method ini digunakan untuk update tampilan setiap kali piece bergerak.

```
1 private void updateStepInfo(Label stepIndicator, Label moveDescriptionLabel, int index) {
2     stepIndicator.setText("Step: " + index + "/" + (solutionPath.size() - 1));
3
4     if (index == 0) {
5         moveDescriptionLabel.setText("Initial position");
6     } else {
7         GameState state = solutionPath.get(index);
8         String moveDescription = state.getMoveDescription();
9         moveDescriptionLabel.setText(moveDescription);
10    }
11 }
12
13 private void updateBoard(GameState state) {
14     boardDisplay.getChildren().clear();
15
16     char[][] board = state.getBoard().getGridCopy();
17
18     for (int row = 0; row < rows; row++) {
19         for (int col = 0; col < cols; col++) {
20             char cell = board[row][col];
21
22             Label cellLabel = new Label(String.valueOf(cell));
23             cellLabel.setAlignment(Pos.CENTER);
24             cellLabel.setPrefSize(50, 50);
25             cellLabel.setFont(Font.font("Monospaced", FontWeight.BOLD, 16));
26             cellLabel.setTextFill(TEXT_COLOR);
27
28             StackPane cellPane = new StackPane(cellLabel);
29             cellPane.setStyle(
30                 "-fx-border-color: " + toHexString(PRIMARY_LIGHT) + ";" +
31                 "-fx-border-width: 1;" +
32                 "-fx-border-radius: 3;" +
33             );
34
35             switch (cell) {
36                 case '.' -> cellPane.setBackground(new Background(new BackgroundFill(EMPTY_CELL, new CornerRadii(3), Insets.EMPTY)));
37                 case 'P' -> cellPane.setBackground(new Background(new BackgroundFill(PLAYER_CELL, new CornerRadii(3), Insets.EMPTY)));
38                 default -> cellPane.setBackground(new Background(new BackgroundFill(OTHER_CELL, new CornerRadii(3), Insets.EMPTY)));
39             }
40             boardDisplay.add(cellPane, col, row);
41         }
42     }
43 }
```

Method void showErrorDialog(String message)

Method untuk menampilkan tampilan error jika terjadi exception.

```
 1 private void showErrorDialog(String message) {  
 2     if (loadingAnimation != null) {  
 3         loadingAnimation.stop();  
 4     }  
 5  
 6     Alert alert = new Alert(Alert.AlertType.ERROR);  
 7     alert.setTitle("Error");  
 8     alert.setHeaderText("Something is wrong with the configuration file!");  
 9     alert.setContentText(message);  
10  
11    DialogPane dialogPane = alert.getDialogPane();  
12    dialogPane.setStyle("-fx-background-color: " + toHexString(BACKGROUND_COLOR) + ";");  
13    dialogPane.lookup(".label.content").setStyle("-fx-font-size: 14px; -fx-font-family: 'Poppins';");  
14    dialogPane.lookup(".header-panel").setStyle("-fx-background-color: " + toHexString(PRIMARY_COLOR) + ";");  
15    dialogPane.lookup(".header-panel .label").setStyle("-fx-text-fill: " + toHexString(TEXT_COLOR) + ";" + "-fx-font-size: 16px; -fx-font-family: 'Poppins';");  
16  
17    alert.showAndWait();  
18    showMainMenu();  
19 }
```

3.2.3. GameBoard.java

Kelas ini berperan dalam membuat objek yang menjadi representasi dari board yang digunakan untuk meletakkan pieces.

Attributes & Constructor

```
● ● ●
1 public class GameBoard {
2     private final int rows, cols;
3     private final char[][] grid;
4     private final GameEnums.GoalPlacement goalPlacement;
5     private final int goalIndex;
6
7     public GameBoard(int rows, int cols, GameEnums.GoalPlacement goalPlacement, int goalIndex) {
8         this.rows = rows;
9         this.cols = cols;
10        this.goalPlacement = goalPlacement;
11        this.goalIndex = goalIndex;
12        this.grid = new char[rows][cols];
13        for (char[] row : grid) {
14            Arrays.fill(row, '.');
15        }
16    }
}
```

Method void placePieces(List<GamePiece> pieces)

Method ini berguna untuk menaruh pieces ke dalam board.



```
1 public void placePieces(List<GamePiece> pieces) {  
2     for (GamePiece piece : pieces) {  
3         for (int[] pos : piece.getPositions()) {  
4             int col = pos[0];  
5             int row = pos[1];  
6             if (!isInBounds(col, row)) {  
7                 throw new IllegalArgumentException("Piece out of bounds");  
8             }  
9             if (grid[row][col] != '.') {  
10                 throw new IllegalArgumentException("Overlapping pieces detected");  
11             }  
12             grid[row][col] = piece.getId();  
13         }  
14     }  
15 }
```

Method boolean isGoalReached(GamePiece primary)

Method ini menentukan apakah primary piece sudah mencapai sisi goal atau belum.



```
1 public boolean isGoalReached(GamePiece primary) {  
2     for (int[] pos : primary.getPositions()) {  
3         int col = pos[0];  
4         int row = pos[1];  
5         switch (goalPlacement) {  
6             case LEFT → {  
7                 if (col == 0 && row == goalIndex) return true;  
8             }  
9             case RIGHT → {  
10                if (col == cols - 1 && row == goalIndex) return true;  
11            }  
12            case TOP → {  
13                if (col == goalIndex && row == 0) return true;  
14            }  
15            case BOTTOM → {  
16                if (col == goalIndex && row == rows - 1) return true;  
17            }  
18        }  
19    }  
20    return false;  
21 }
```

Method boolean isInBounds(int col, int row)

Method ini memastikan suatu koordinat berada di dalam board atau tidak.



```
1 public boolean isInBounds(int col, int row) {  
2     return col ≥ 0 && col < cols && row ≥ 0 && row < rows;  
3 }
```

Method Getter



```
1 public char[][] getGridCopy() {  
2     char[][] copy = new char[rows][cols];  
3     for (int row = 0; row < rows; row++) {  
4         System.arraycopy(grid[row], 0, copy[row], 0, cols);  
5     }  
6     return copy;  
7 }  
8  
9 public int getGoalIndex() {  
10    return goalIndex;  
11 }  
12  
13 public GameEnums.GoalPlacement getGoalPlacement() {  
14    return goalPlacement;  
15 }
```

3.2.4. GameEnums.java

Kelas ini berisi kelas enum untuk menyatakan orientasi piece dan sisi goal.



```
1 public class GameEnums {
2     public enum GoalPlacement {
3         TOP,
4         LEFT,
5         RIGHT,
6         BOTTOM
7     }
8
9     public enum Orientation {
10        VERTICAL,
11        HORIZONTAL
12    }
13 }
```

3.2.5. GamePiece.java

Kelas ini berperan dalam membuat objek yang merepresentasikan pieces.

Attributes & Constructor

```
● ● ●
1 public final class GamePiece {
2     private final char id;
3     private final List<int[]> positions;
4     private final GameEnums.Orientation orientation;
5
6     public GamePiece(char id, List<int[]> positions) {
7         if (positions == null || positions.size() < 2) {
8             throw new IllegalArgumentException("Positions must contain at least two points");
9         }
10        this.id = id;
11        this.positions = new ArrayList<>(positions);
12        this.orientation = determineOrientation();
13        validatePositions();
14    }
15
16    public GamePiece(GamePiece other) {
17        this.id = other.id;
18        this.orientation = other.orientation;
19        this.positions = new ArrayList<>();
20        for (int[] pos : other.positions) {
21            this.positions.add(new int[] { pos[0], pos[1] });
22        }
23    }
}
```

Method GameEnums.Orientation determineOrientation()

Method ini menentukan orientasi dari piece.

```
● ● ●  
1 private GameEnums.Orientation determineOrientation() {  
2     int[] first = positions.get(0);  
3     int[] second = positions.get(1);  
4     return (first[1] == second[1]) ? GameEnums.Orientation.HORIZONTAL : GameEnums.Orientation.VERTICAL;  
5 }
```

Method void validatePositions()

Method ini menentukan apakah piece memiliki bentuk yang valid atau tidak berdasarkan orientasinya.

```
● ● ●  
1 private void validatePositions() {  
2     if (orientation == GameEnums.Orientation.HORIZONTAL) {  
3         int row = positions.get(0)[1];  
4         for (int[] pos : positions) {  
5             if (pos[1] != row) {  
6                 throw new IllegalArgumentException("All positions must be in the same row for horizontal pieces");  
7             }  
8         }  
9     }  
10    else {  
11        int col = positions.get(0)[0];  
12        for (int[] pos : positions) {  
13            if (pos[0] != col) {  
14                throw new IllegalArgumentException("All positions must be in the same column for vertical pieces");  
15            }  
16        }  
17    }  
18 }
```

Method boolean canMove(int distance, char[][] board)

Method ini menentukan apakah piece dapat bergerak sejauh distance berdasarkan

orientasinya.

```
● ● ●  
1 public boolean canMove(int distance, char[][] board) {  
2     for (int[] pos : positions) {  
3         int newX = pos[0];  
4         int newY = pos[1];  
5  
6         if (orientation == GameEnums.Orientation.HORIZONTAL) {  
7             newX += distance;  
8             if (newX < 0 || newX ≥ board[0].length) return false; // cols  
9             if (!occupiesPosition(newX, pos[1]) && board[pos[1]][newX] ≠ '.') return false; // [row][col]  
10        } else {  
11            newY += distance;  
12            if (newY < 0 || newY ≥ board.length) return false; // rows  
13            if (!occupiesPosition(pos[0], newY) && board[newY][pos[0]] ≠ '.') return false;  
14        }  
15    }  
16    return true;  
17 }
```

Method void move(int distance)

Method untuk menggerakkan piece berdasarkan orientasinya.



```
1 public void move(int distance) {  
2     for (int[] pos : positions) {  
3         if (orientation == GameEnums.Orientation.HORIZONTAL) {  
4             pos[0] += distance;  
5         } else {  
6             pos[1] += distance;  
7         }  
8     }  
9 }
```

Method boolean occupiesPosition(int x, int y)

Method ini menentukan apakah piece menempati suatu koordinat.



```
1 private boolean occupiesPosition(int x, int y) {  
2     for (int[] pos : positions) {  
3         if (pos[0] == x && pos[1] == y) return true;  
4     }  
5     return false;  
6 }
```

Method Getter

```
 1 public char getId() {
 2     return id;
 3 }
 4
 5 public List<int[]> getPositions() {
 6     return Collections.unmodifiableList(positions);
 7 }
 8
 9 public GameEnums.Orientation getOrientation() {
10     return orientation;
11 }
```

3.2.6. GameState.java

Kelas ini berperan dalam membuat objek yang merepresentasikan keadaan board dan pieces saat ini.

Attributes & Constructor

```
● ● ●
1 public class GameState {
2     private final GameBoard board;
3     private final List<GamePiece> pieces;
4     private final GameState parent;
5     private final int cost;
6     private final String moveDescription;
7
8     public GameState(GameBoard board, List<GamePiece> pieces, GameState parent, String moveDescription) {
9         this.board = board;
10        this.pieces = Collections.unmodifiableList(new ArrayList<>(pieces));
11        this.parent = parent;
12        this.moveDescription = moveDescription;
13        this.cost = parent == null ? 0 : parent.cost + 1;
14    }
}
```

Method GamePiece getPrimaryPiece()

Method untuk mencari primary piece.

```
● ● ●
1 private GamePiece getPrimaryPiece() {
2     for (GamePiece piece : pieces) {
3         if (piece.getId() == 'P') return piece;
4     }
5     throw new IllegalStateException("Red car (id='P') not found");
6 }
```

Method boolean isGoal()

Method untuk menentukan apakah primary piece sudah mencapai goal.



```
1 public boolean isGoal() {  
2     return board.isGoalReached(getPrimaryPiece());  
3 }
```

Method List<GameState> generateSuccessors()

Method untuk menciptakan semua GameState dari seluruh kemungkinan pergerakan pieces berdasarkan state saat ini.



```
1 public List<GameState> generateSuccessors() {
2     List<GameState> successors = new ArrayList<>();
3     for (int i = 0; i < pieces.size(); i++) {
4         GamePiece piece = pieces.get(i);
5         for (int dir : new int[]{-1, 1}) {
6             GamePiece movedPiece = new GamePiece(piece);
7             try {
8                 if (movedPiece.canMove(dir, board.getGridCopy())) {
9                     movedPiece.move(dir);
10                    List<GamePiece> newPieces = new ArrayList<>(pieces);
11                    newPieces.set(i, movedPiece);
12
13                    GameBoard newBoard = new GameBoard(
14                        board.getGridCopy().length,
15                        board.getGridCopy()[0].length,
16                        board.getGoalPlacement(),
17                        board.getGoalIndex()
18                    );
19                    newBoard.placePieces(newPieces);
20
21                    String direction;
22                    if (piece.getOrientation() == GameEnums.Orientation.HORIZONTAL) {
23                        direction = (dir == 1 ? "Right" : "Left");
24                    } else {
25                        direction = (dir == 1 ? "Down" : "Up");
26                    }
27
28                    successors.add(new GameState(newBoard, newPieces, this,
29                        "Move " + piece.getId() + " " + direction));
30                }
31            } catch (IllegalArgumentException ignored) {}
32        }
33    }
34    return successors;
35 }
```

Method Getter



```
1 public List<GamePiece> getPieces() {  
2     return pieces;  
3 }  
4  
5 public int getCost() {  
6     return cost;  
7 }  
8  
9 public String getMoveDescription() {  
10    return moveDescription;  
11 }  
12  
13 public GameState getParent() {  
14    return parent;  
15 }  
16  
17 public GameBoard getBoard() {  
18    return board;  
19 }
```

3.2.7. Algorithm

Algorithm merupakan package yang berisi interface dan implementasi metode search.

Interface SearchAlgorithm



```
1 public interface SearchAlgorithm {  
2     List<GameState> solve(GameState initialState);  
3     int getNodesExplored();  
4 }
```

Class StandardSearch

Kelas yang menggunakan priority queue sebagai struktur data penjelajahan tree.

Attributes & Constructor



```
1 public class StandardSearch implements SearchAlgorithm{  
2     private final Comparator<GameState> comparator;  
3     private int nodesExplored = 0;  
4  
5     public StandardSearch(Comparator<GameState> comparator) {  
6         this.comparator = comparator;  
7     }
```

Method `List<GameState> solve(GameState initialState)`



```
1  @Override
2  public List<GameState> solve(GameState initialState) {
3      PriorityQueue<GameState> queue = new PriorityQueue<>(comparator);
4      Set<String> visited = new HashSet<>();
5      nodesExplored = 0;
6
7      queue.add(initialState);
8
9      while (!queue.isEmpty()) {
10         nodesExplored++;
11         GameState current = queue.poll();
12
13         if (visited.contains(current.getBoard().toString())) {
14             continue;
15         }
16
17         visited.add(current.getBoard().toString());
18
19         if (current.isGoal()) {
20             return reconstructPath(current);
21         }
22
23         for (GameState neighbor : current.generateSuccessors()) {
24             if (!visited.contains(neighbor.getBoard().toString())) {
25                 queue.add(neighbor);
26             }
27         }
28     }
}
```

Method `List<GameState> reconstructPath(GameState goalState)`

Method ini membuat List of GameState dari initial state hingga goal state.



```
1 private List<GameState> reconstructPath(GameState goalState) {  
2     List<GameState> path = new ArrayList<>();  
3     GameState current = goalState;  
4     while (current != null) {  
5         path.add(current);  
6         current = current.getParent();  
7     }  
8     Collections.reverse(path);  
9     return path;  
10 }
```

Method int getNodesExplored()



```
1 @Override
2 public int getNodesExplored() {
3     return nodesExplored;
4 }
```

Class IDAStarSearch

Kelas yang mengimplementasi algoritma IDA*.

Attributes & Constructor



```
1 public class IDAStarSearch implements SearchAlgorithm {  
2     private final HeuristicFunction heuristic;  
3     private int nodesExplored = 0;  
4     private Map<String, Integer> visited; // String board, Integer cost  
5  
6     public IDAStarSearch(HeuristicFunction heuristic) {  
7         this.heuristic = heuristic;  
8     }
```

Method List<GameState> solve(GameState initialState)



```
1  @Override
2  public List<GameState> solve(GameState initialState) {
3      nodesExplored = 0;
4      int threshold = heuristic.calculate(initialState);
5
6      while (true) {
7          visited = new HashMap<>();
8          List<GameState> initialPath = new ArrayList<>();
9          initialPath.add(initialState);
10         Result result = search(initialState, threshold, initialPath);
11
12         if (result.found) {
13             return result.path;
14         }
15
16         if (result.nextThreshold == Integer.MAX_VALUE) {
17             return Collections.emptyList();
18         }
19
20         threshold = result.nextThreshold;
21     }
22 }
```

Method search()



```
1  private Result search(GameState state, int threshold, List<GameState> path) {
2      nodesExplored++;
3
4      int f = state.getCost() + heuristic.calculate(state);
5
6      if (f > threshold) {
7          return new Result(false, null, f);
8      }
9
10     String boardKey = state.getBoard().toString();
11     Integer previousCost = visited.get(boardKey);
12     if (previousCost != null && previousCost <= state.getCost()) {
13         return new Result(false, null, threshold + 1);
14     }
15     visited.put(boardKey, state.getCost());
16
17     if (state.isGoal()) {
18         GameState lastState = state.lastMove();
19         path.add(lastState);
20         nodesExplored += state.getPrimaryPiece().getPositions().size();
21         return new Result(true, path, threshold);
22     }
23
24     // Recursive DFS
25     int nextThreshold = Integer.MAX_VALUE;
26     List<GameState> successors = state.generateSuccessors();
27
28     for (GameState successor : successors) {
29         path.add(successor);
30         Result result = search(successor, threshold, path);
31
32         if (result.found) {
33             return result;
34         }
35
36         path.remove(path.size() - 1);
37         nextThreshold = Math.min(nextThreshold, result.nextThreshold);
38     }
39
40     return new Result(false, null, nextThreshold);
41 }
```

Method int getNodesExplored() dan Helper class Result

```
● ● ●  
1  @Override  
2  public int getNodesExplored() {  
3      return nodesExplored;  
4  }  
5  
6  private static class Result {  
7      final boolean found;  
8      final List<GameState> path;  
9      final int nextThreshold;  
10  
11     Result(boolean found, List<GameState> path, int nextThreshold) {  
12         this.found = found;  
13         this.path = path;  
14         this.nextThreshold = nextThreshold;  
15     }  
16 }
```

3.2.8. Comparator

Kelas-kelas yang merupakan metode untuk menentukan game state mana yang akan dikunjungi selanjutnya berdasarkan $f(n)$ dan $g(n)$.

UCSComparator

```
● ● ●  
1 public class UCSComparator implements Comparator<GameState>{  
2     @Override  
3     public int compare(GameState a, GameState b) {  
4         return Integer.compare(a.getCost(), b.getCost());  
5     }  
6 }
```

GreedyComparator

```
● ● ●  
1 public class GreedyComparator implements Comparator<GameState>{  
2     private final HeuristicFunction heuristic;  
3  
4     public GreedyComparator(HeuristicFunction heuristic) {  
5         this.heuristic = heuristic;  
6     }  
7  
8     @Override  
9     public int compare(GameState a, GameState b) {  
10         return Integer.compare(  
11             heuristic.calculate(a),  
12             heuristic.calculate(b)  
13         );  
14     }  
15 }
```

AStarComparator

```
● ● ●  
1 public class AStarComparator implements Comparator<GameState> {  
2     private final HeuristicFunction heuristic;  
3  
4     public AStarComparator(HeuristicFunction heuristic) {  
5         this.heuristic = heuristic;  
6     }  
7  
8     @Override  
9     public int compare(GameState a, GameState b) {  
10         return Integer.compare(  
11             a.getCost() + heuristic.calculate(a),  
12             b.getCost() + heuristic.calculate(b)  
13         );  
14     }  
15 }
```

3.2.9. Heuristic

Kelas ini berisi perhitungan heuristic untuk algoritma heuristic search.

Interface HeuristicFunction



```
1 public interface HeuristicFunction {  
2     int calculate(GameState state);  
3 }
```

Class DistanceHeuristic

Kelas ini menghitung fungsi heuristic dengan menggunakan Manhattan distance antara primary piece dengan goal.

```
1 public class DistanceHeuristic implements HeuristicFunction {
2     @Override
3     public int calculate(GameState state) {
4         GamePiece primaryPiece = getPrimaryPiece(state);
5         char[][] grid = state.getBoard().getGridCopy();
6         GameEnums.GoalPlacement goalSide = state.getBoard().getGoalPlacement();
7
8         int distanceToGoal = 0;
9
10        for (int[] pos : primaryPiece.getPositions()) {
11            int x = pos[0];
12            int y = pos[1];
13
14            switch (goalSide) {
15                case RIGHT → distanceToGoal = grid[0].length - 1 - x;
16                case LEFT → distanceToGoal = x;
17                case BOTTOM → distanceToGoal = grid.length - 1 - y;
18                case TOP → distanceToGoal = y;
19            }
20            break;
21        }
22
23        return distanceToGoal;
24    }
```

Class BlockerHeuristic

Kelas ini menghitung fungsi heuristic dengan menggunakan banyaknya piece yang menghalangi primary piece ke goal.

```
● ● ●
1 public class BlockerHeuristic implements HeuristicFunction {
2     @Override
3     public int calculate(GameState state) {
4         GamePiece primaryPiece = getPrimaryPiece(state);
5         char[][] grid = state.getBoard().getGridCopy();
6         GameEnums.GoalPlacement goalSide = state.getBoard().getGoalPlacement();
7
8         Set<Character> blockers = new HashSet<>();
9
10        for (int[] pos : primaryPiece.getPositions()) {
11            int x = pos[0];
12            int y = pos[1];
13
14            switch (goalSide) {
15                case RIGHT → {
16                    for (int col = x + 1; col < grid[0].length; col++) {
17                        char cell = grid[y][col];
18                        if (cell ≠ '.' && cell ≠ 'P') {
19                            blockers.add(cell);
20                        }
21                    }
22                }
23                case LEFT → {
24                    for (int col = x - 1; col ≥ 0; col--) {
25                        char cell = grid[y][col];
26                        if (cell ≠ '.' && cell ≠ 'P') {
27                            blockers.add(cell);
28                        }
29                    }
30                }
31                case BOTTOM → {
32                    for (int row = y + 1; row < grid.length; row++) {
33                        char cell = grid[row][x];
34                        if (cell ≠ '.' && cell ≠ 'P') {
35                            blockers.add(cell);
36                        }
37                    }
38                }
39                case TOP → {
40                    for (int row = y - 1; row ≥ 0; row--) {
41                        char cell = grid[row][x];
42                        if (cell ≠ '.' && cell ≠ 'P') {
43                            blockers.add(cell);
44                        }
45                    }
46                }
47            }
48            break;
49        }
50
51        return blockers.size();
52    }
}
```

Class CombinedHeuristic

Kombinasi antara heuristic distance dan blocker. Heuristic ini menjumlahkan nilai kedua heuristic.

```
● ● ●
1 public class CombinedHeuristic implements HeuristicFunction {
2     private final DistanceHeuristic distanceHeuristic = new DistanceHeuristic();
3     private final BlockerHeuristic blockerCountHeuristic = new BlockerHeuristic();
4
5     @Override
6     public int calculate(GameState state) {
7         return distanceHeuristic.calculate(state) + blockerCountHeuristic.calculate(state);
8     }
}
```

3.2.10. GameSolver.java

Kelas yang berkomunikasi dengan algorithm, comparator, dan heuristic untuk melakukan solving berdasarkan pilihan user.

Method HeuristicFunction createHeuristic(String heuristicName)

```
● ● ●
```

```
1 public static HeuristicFunction createHeuristic(String heuristicName) {  
2     return switch (heuristicName) {  
3         case "Distance" → new DistanceHeuristic();  
4         case "Blocker Count" → new BlockerHeuristic();  
5         case "Combined" → new CombinedHeuristic();  
6         default → new CombinedHeuristic(); // Default to combined  
7     };  
8 }
```

Method SearchAlgorithm createSolver(String algorithmName, String heuristicName)

```
● ● ●
```

```
1 public static SearchAlgorithm createSolver(String algorithmName, String heuristicName) {  
2     HeuristicFunction heuristic = createHeuristic(heuristicName);  
3  
4     return switch (algorithmName) {  
5         case "UCS" → new StandardSearch(new UCSComparator());  
6         case "Greedy Best First" → new StandardSearch(new GreedyComparator(heuristic));  
7         case "A*" → new StandardSearch(new AStarComparator(heuristic));  
8         default → throw new IllegalArgumentException("Unknown algorithm: " + algorithmName);  
9     };  
10 }
```

Method SolverResult solve(GameState initialState, String algorithmName, String heuristicName)

```
● ● ●  
1 public static SolverResult solve(GameState initialState, String algorithmName, String heuristicName) {  
2     SearchAlgorithm algorithm = createSolver(algorithmName, heuristicName);  
3     List<GameState> path = algorithm.solve(initialState);  
4     return new SolverResult(path, algorithm.getNodesExplored());  
5 }
```

Class SolverResult

Class untuk menyimpan hasil pencarian solusi.



```
1 public static class SolverResult {
2     private final List<GameState> path;
3     private final int nodesExplored;
4
5     public SolverResult(List<GameState> path, int nodesExplored) {
6         this.path = path;
7         this.nodesExplored = nodesExplored;
8     }
9
10    public List<GameState> getPath() {
11        return path;
12    }
13
14    public int getNodesExplored() {
15        return nodesExplored;
16    }
17 }
```

BAB IV

EKSPERIMEN

4.1 Test Case 1-1: UCS

Masukan	<pre>test > test-case1.txt 1 6 6 2 11 3 AAB.. F 4 ..BCDF 5 GPPCDFK 6 GH.III 7 GHJ... 8 LLJMM.</pre>
GUI:	

Rush Hour Solver

Rush Hour Solver with Pathfinding Algorithm

Algorithm:

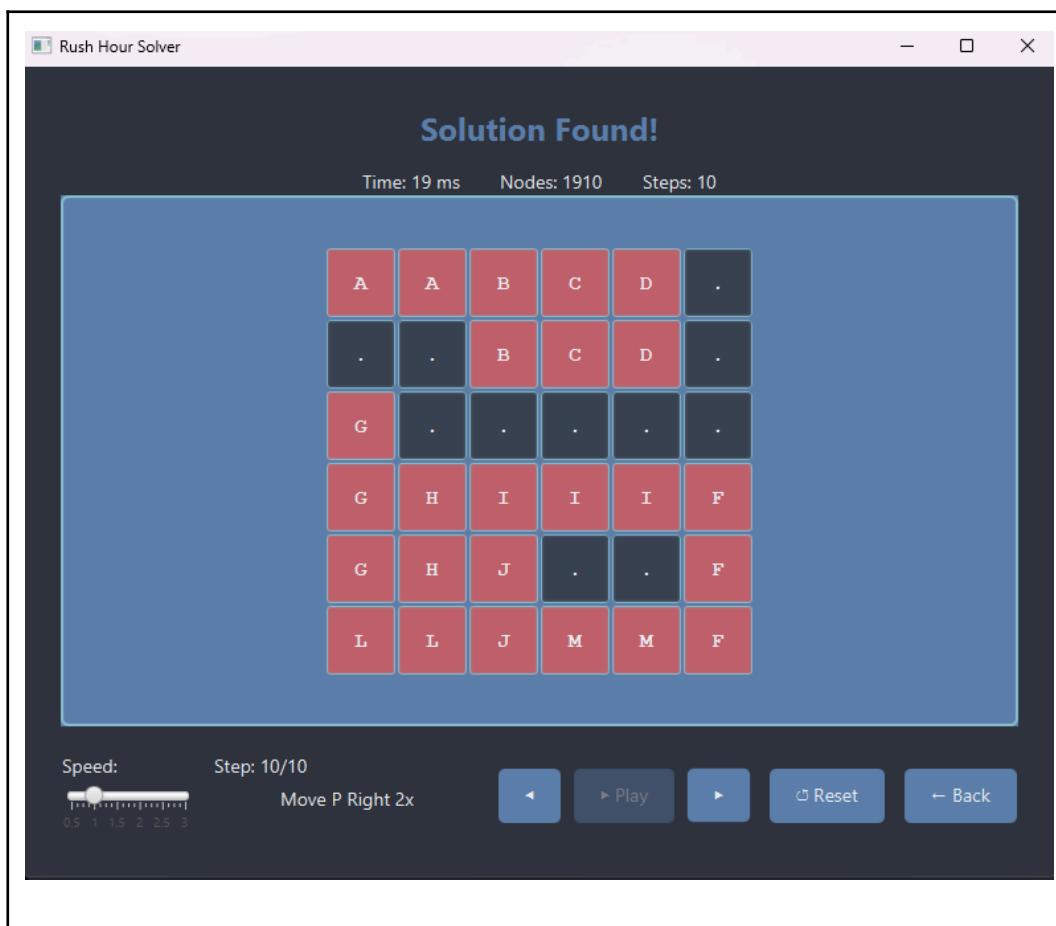
Select Puzzle File

Keluaran

Video Step by Step:

Video Test Case Stima Tucil 3

GUI:

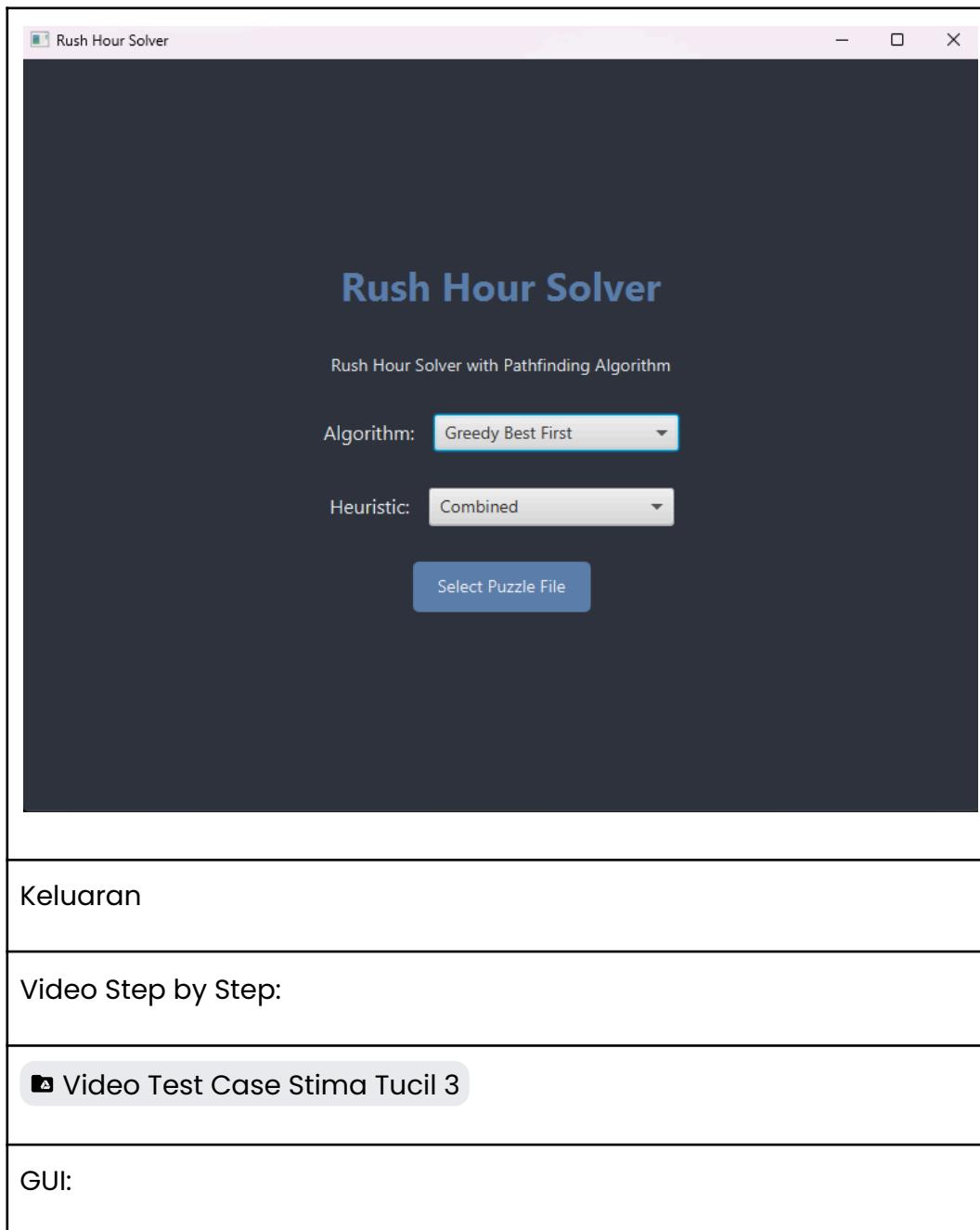


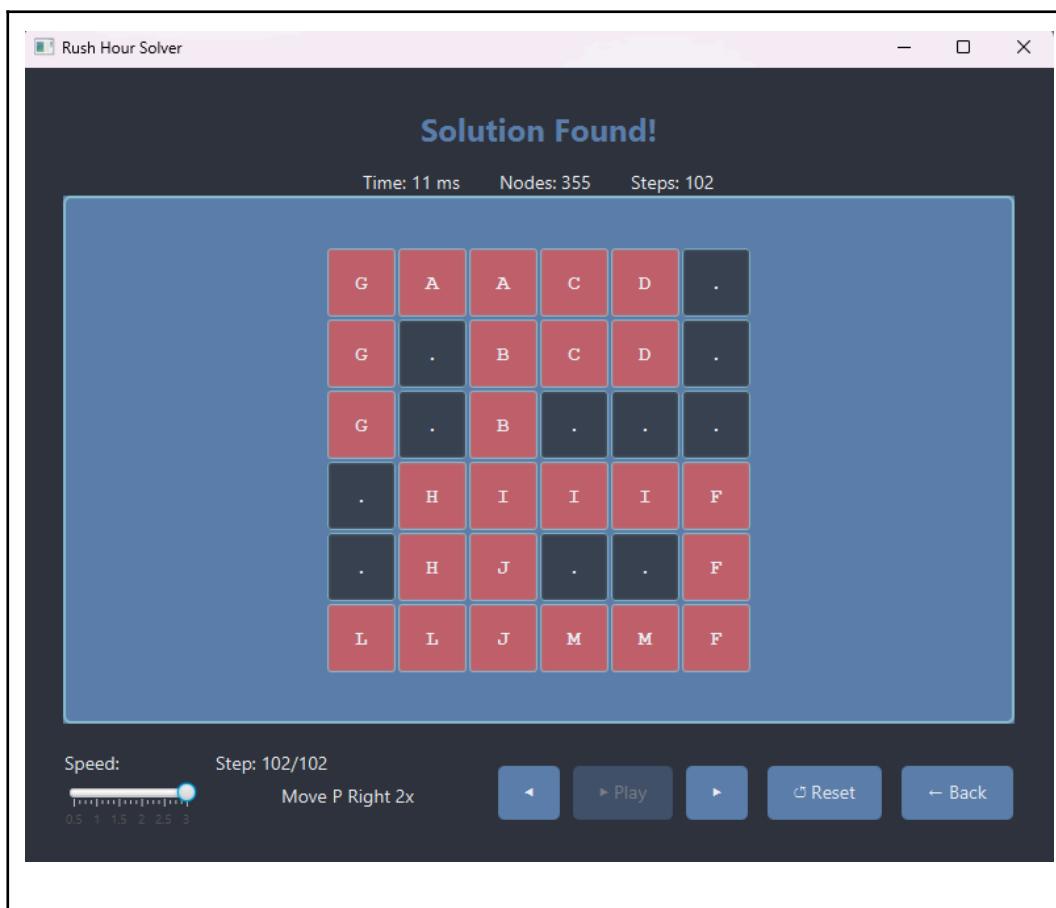
4.2 Test Case 1-2A: Greedy Best + Combine Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



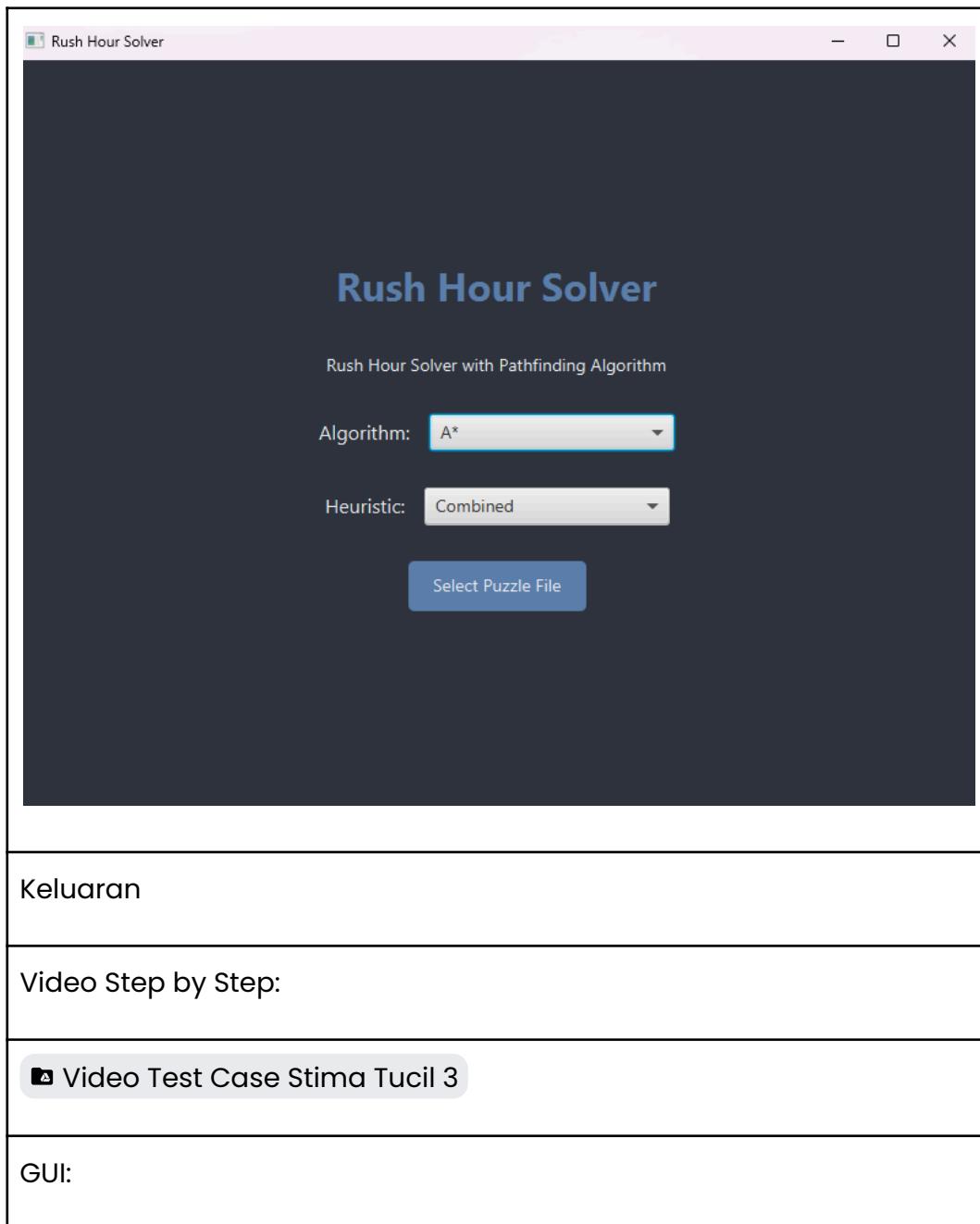


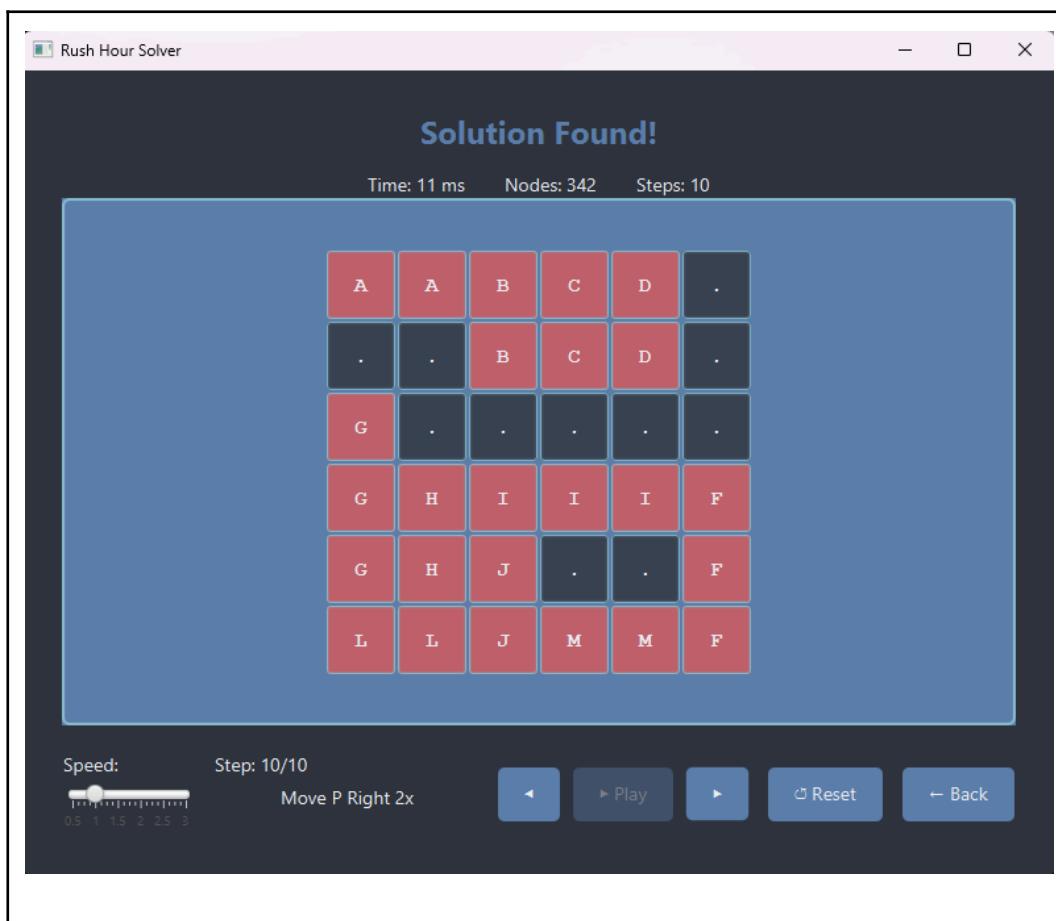
4.3 Test Case 1-3A: A* + Combine Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



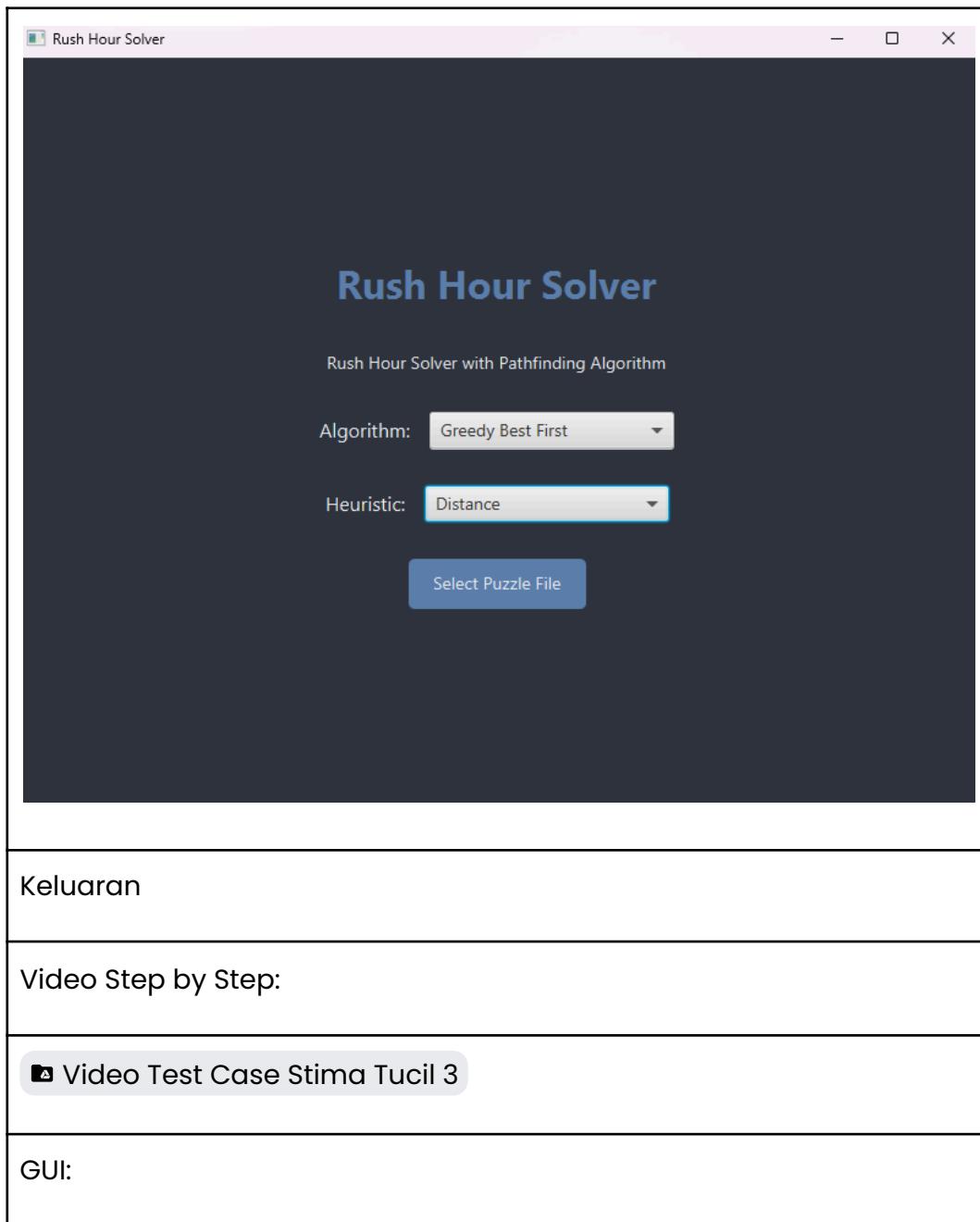


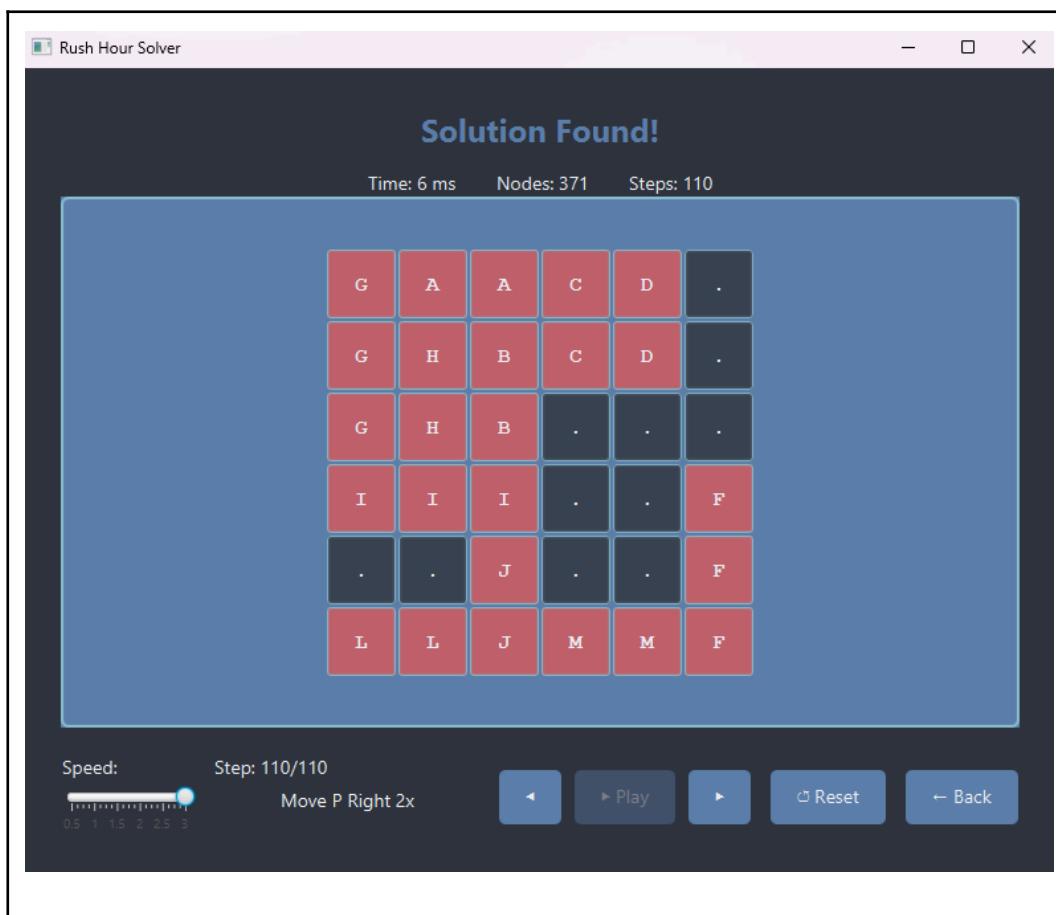
4.4 Test Case 1-2B: Greedy Best + Manhattan Distance Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



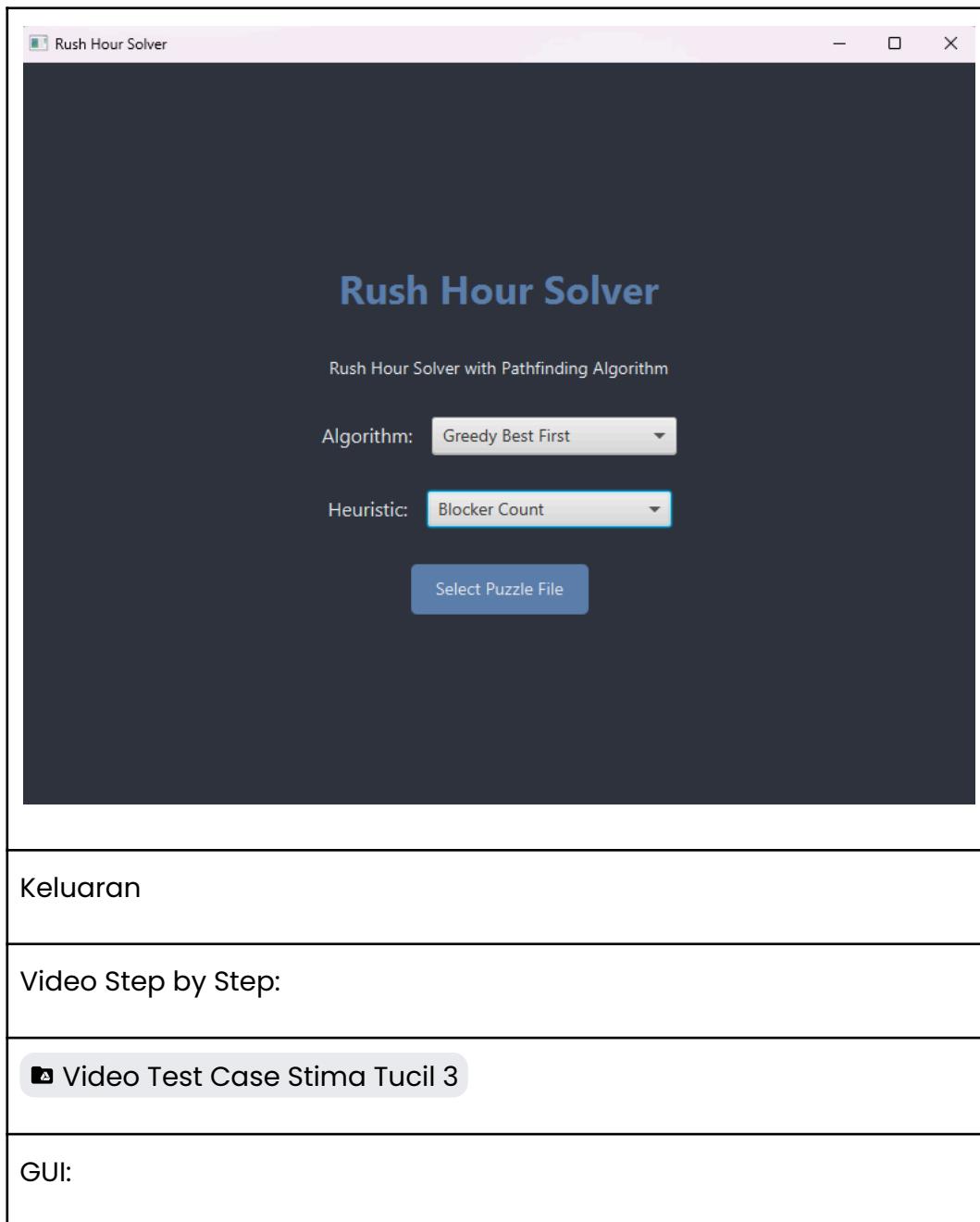


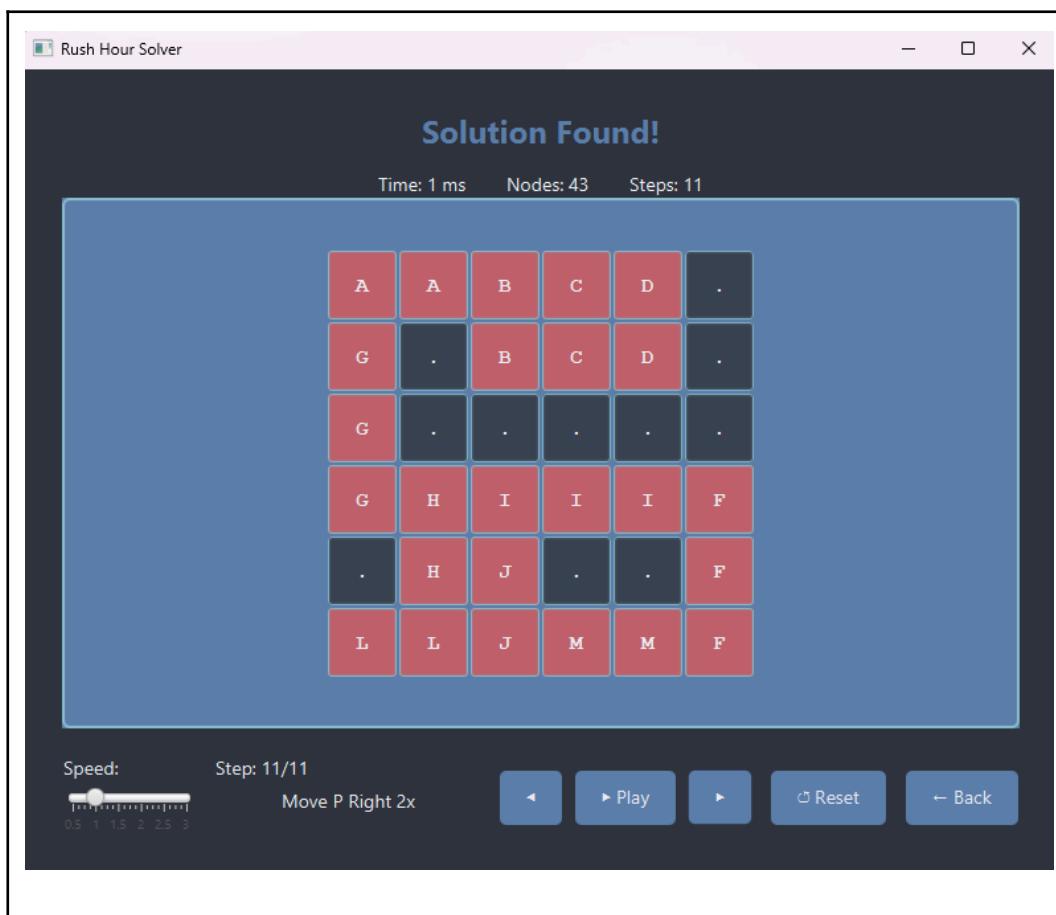
4.5 Test Case 1-2C: Greedy Best + Blocker Count Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



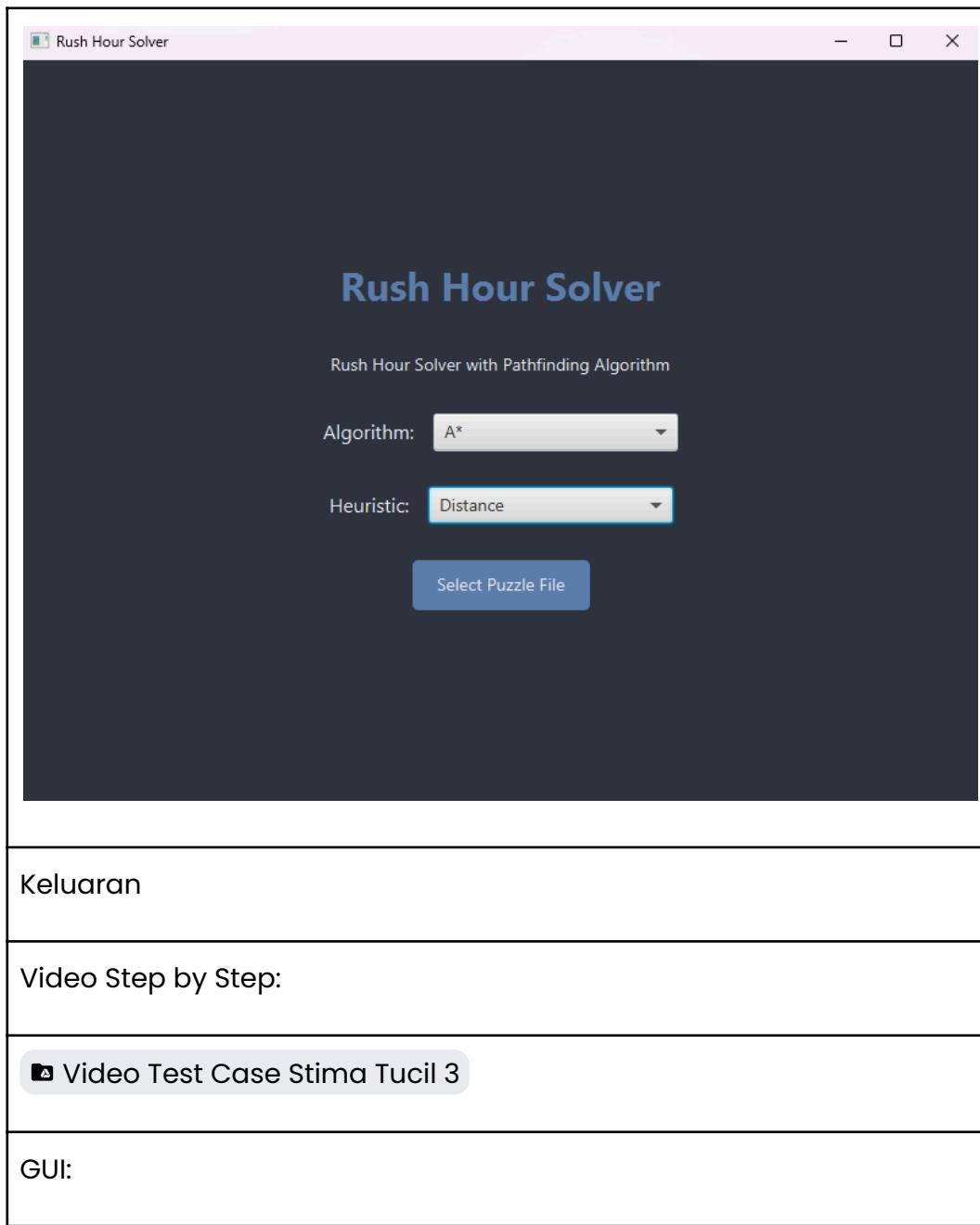


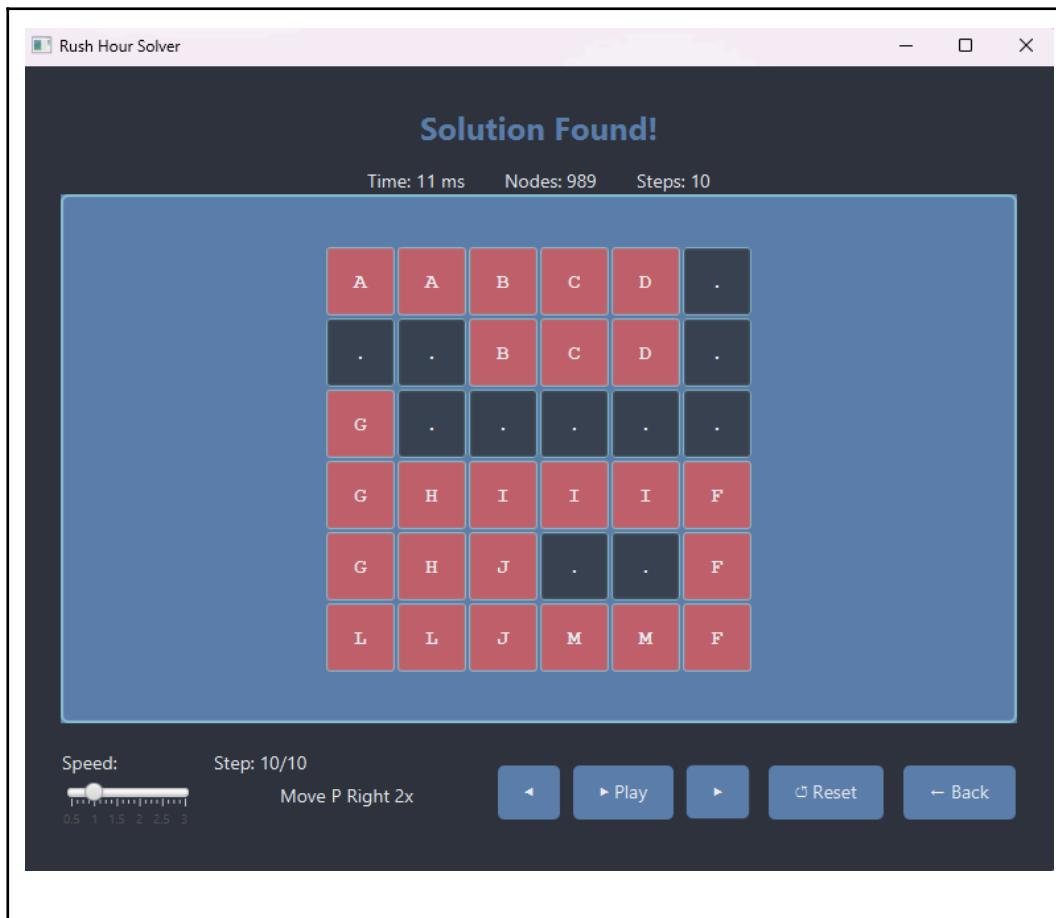
4.6 Test Case 1-3B: A* + Manhattan Distance Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



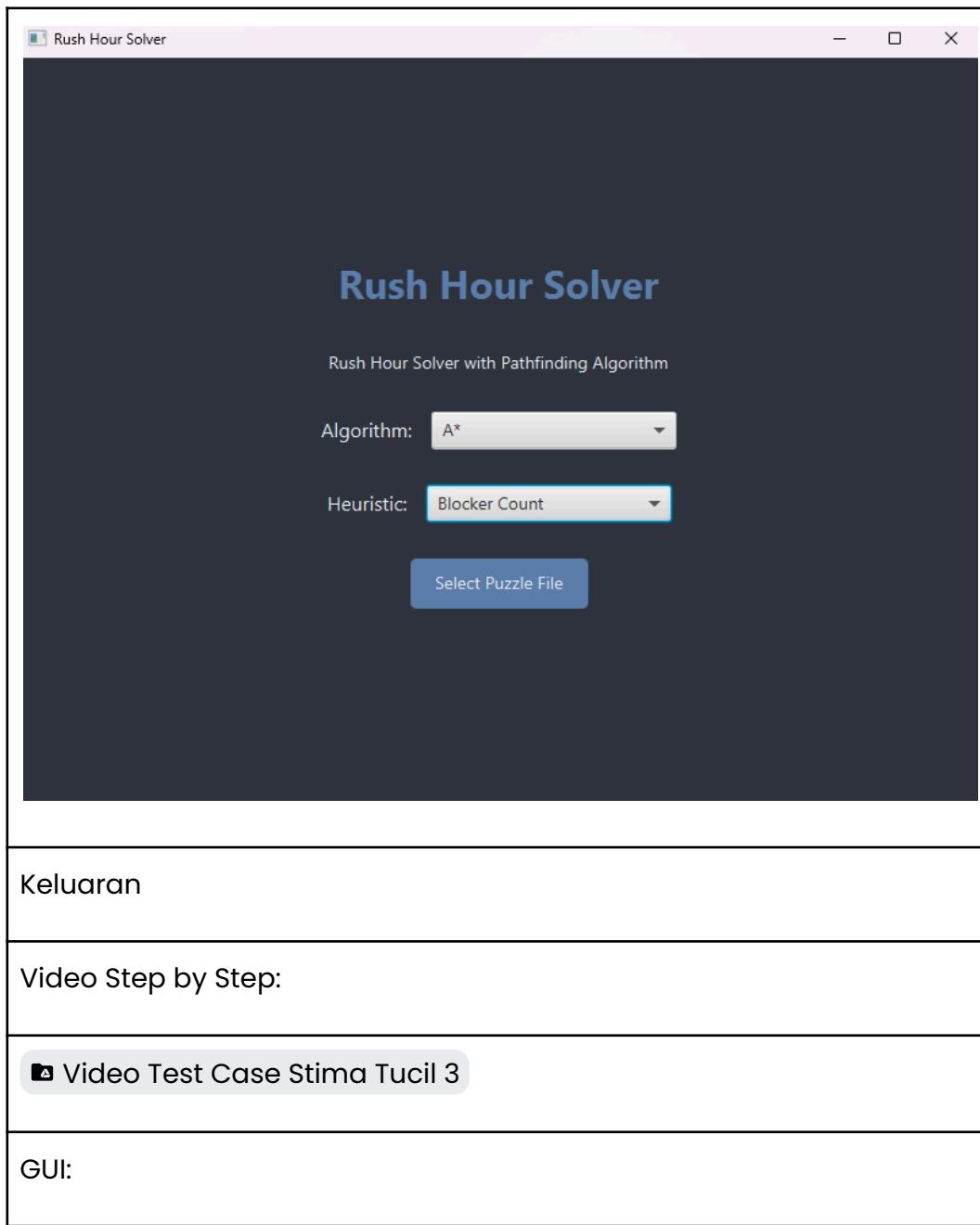


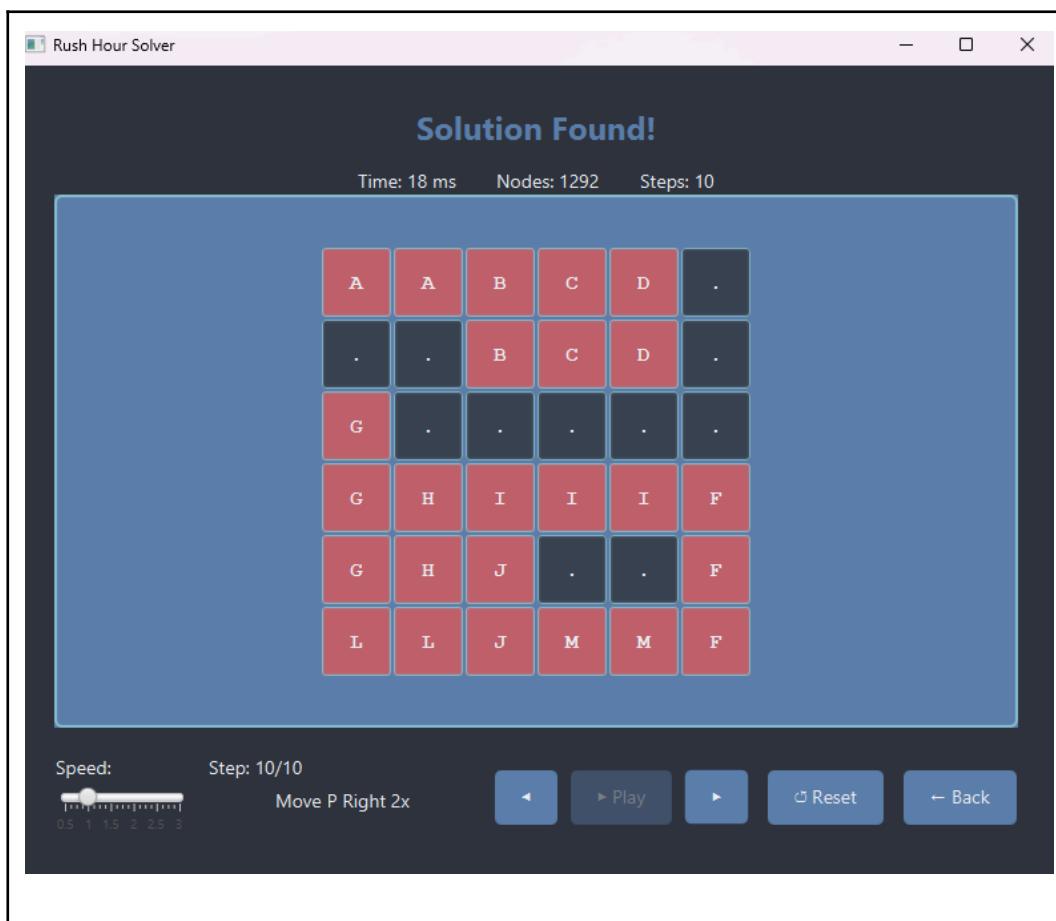
4.7 Test Case 1-3C: A*+ Blocker Count Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



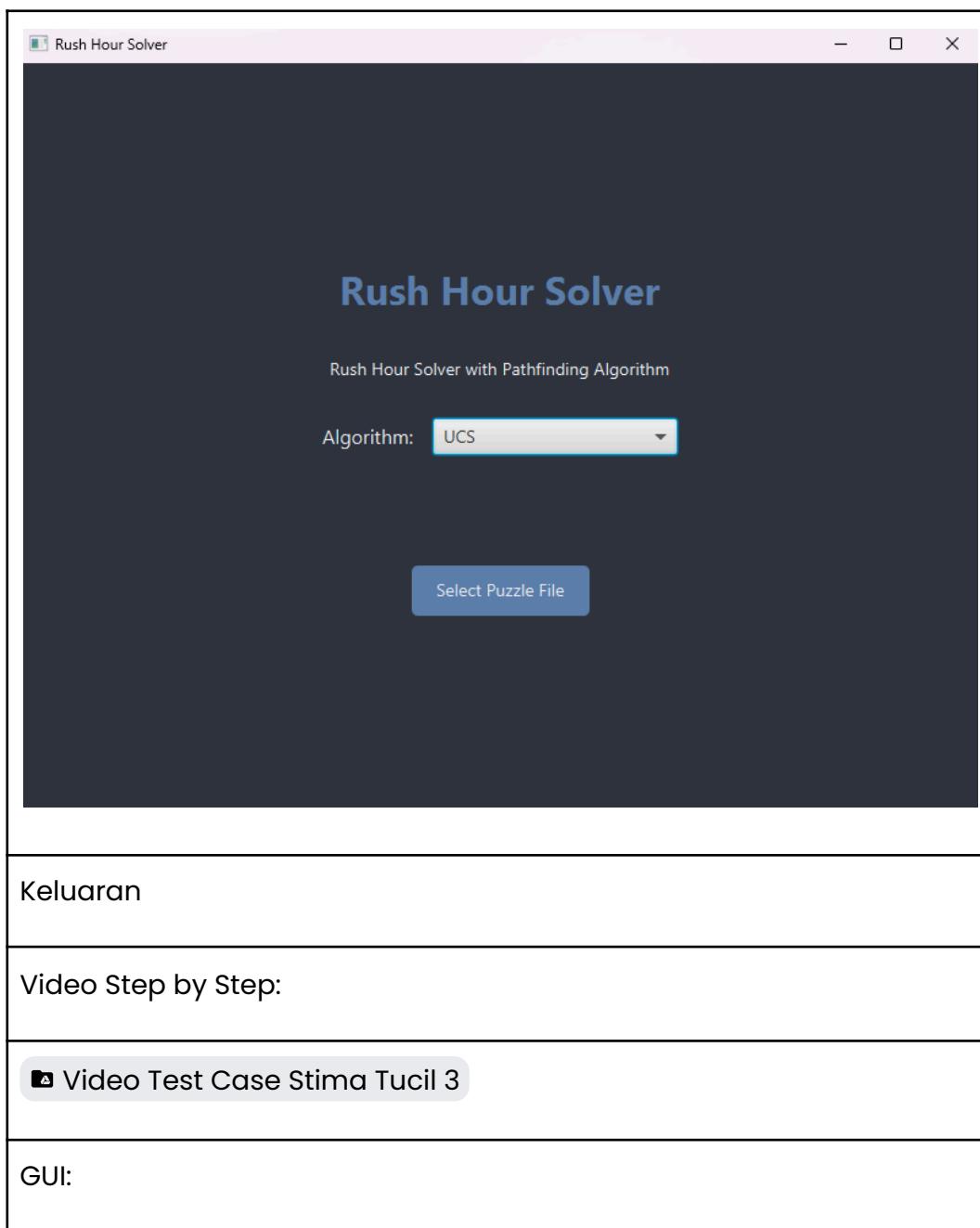


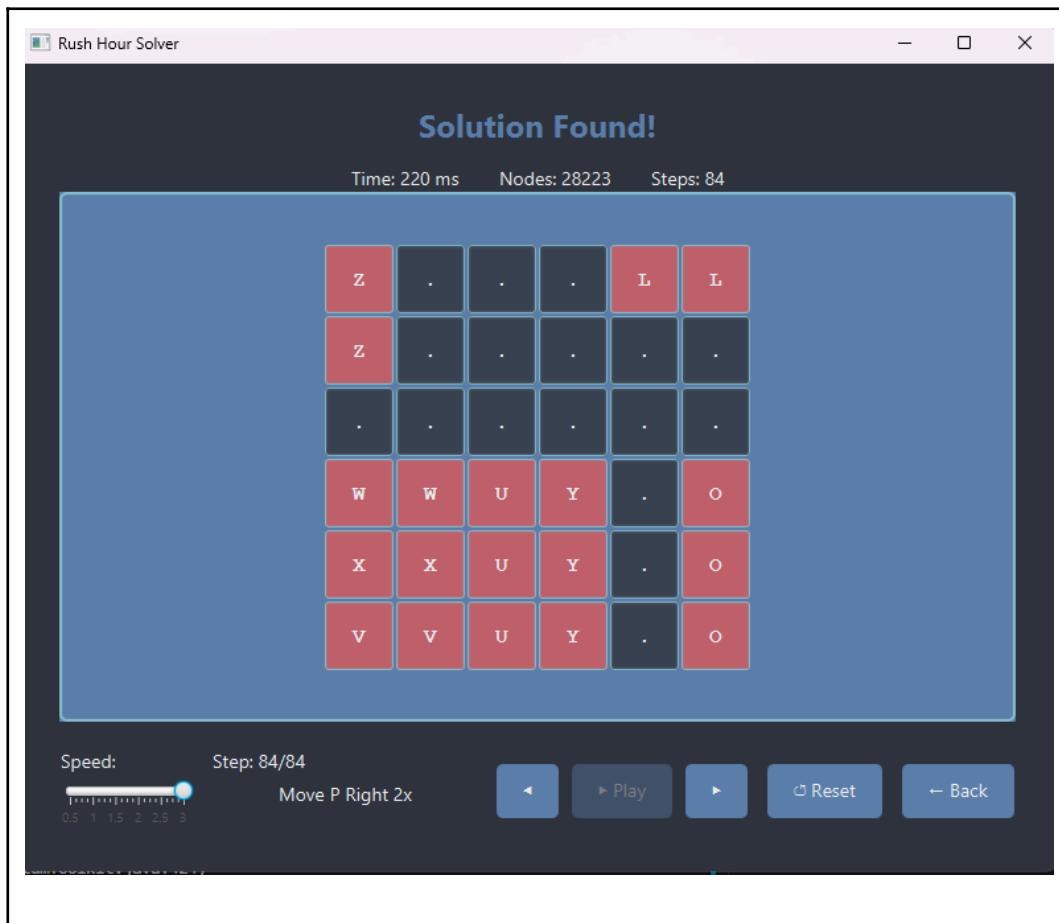
4.8 Test Case 2-1: UCS

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



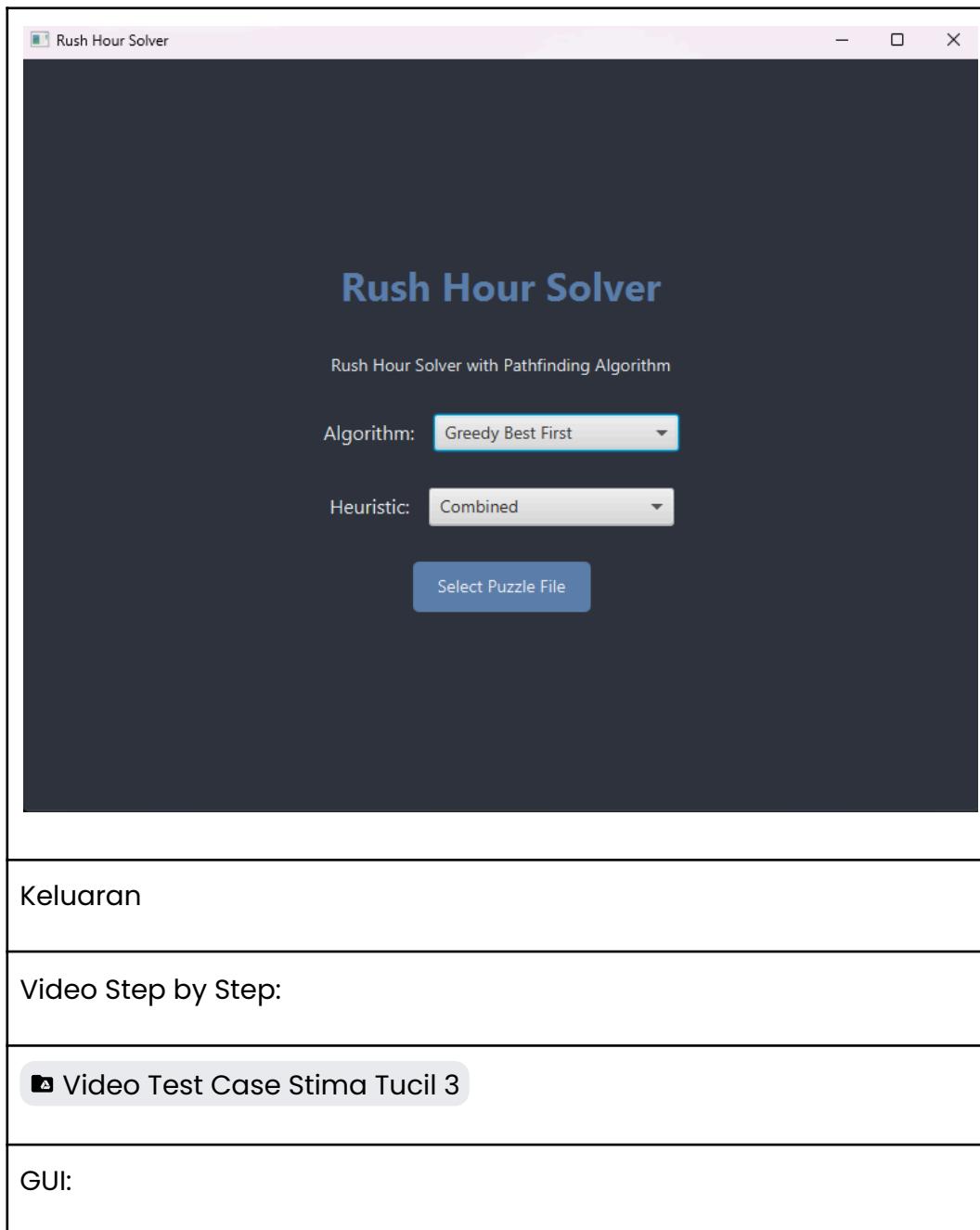


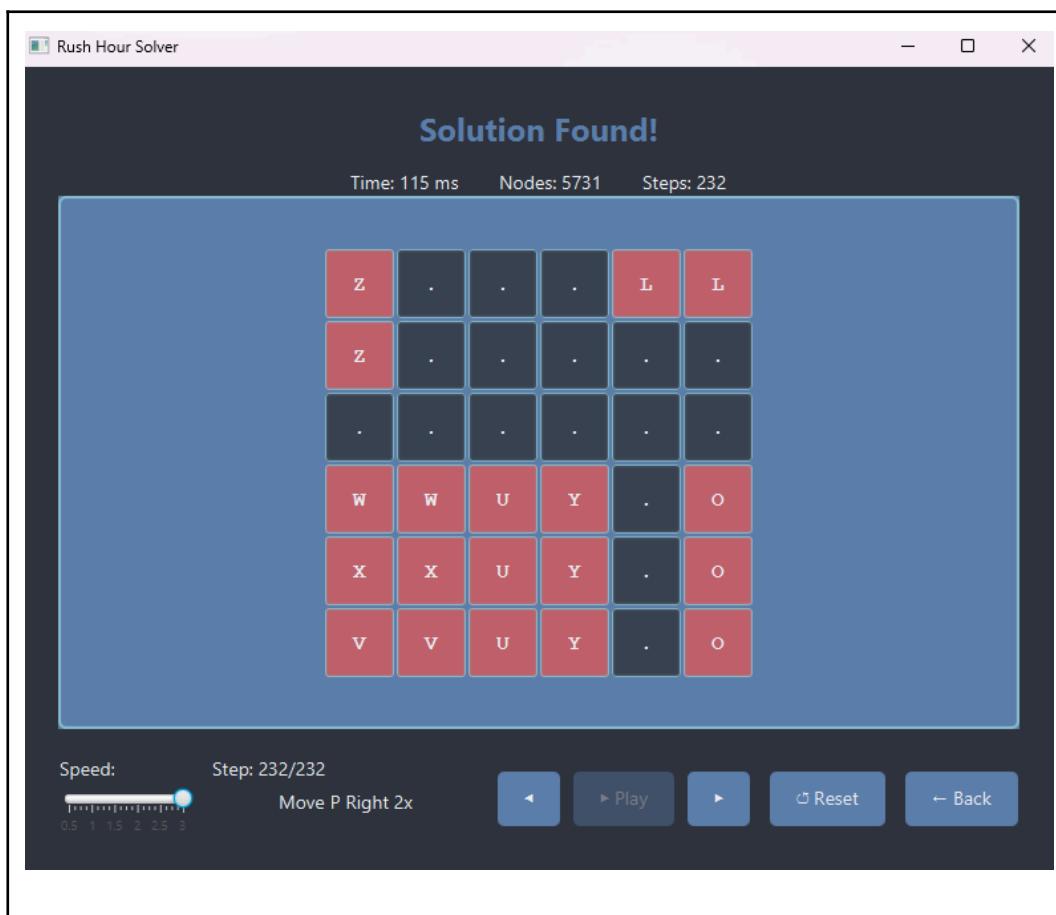
4.9 Test Case 2-2A: Greedy Best + Combine Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



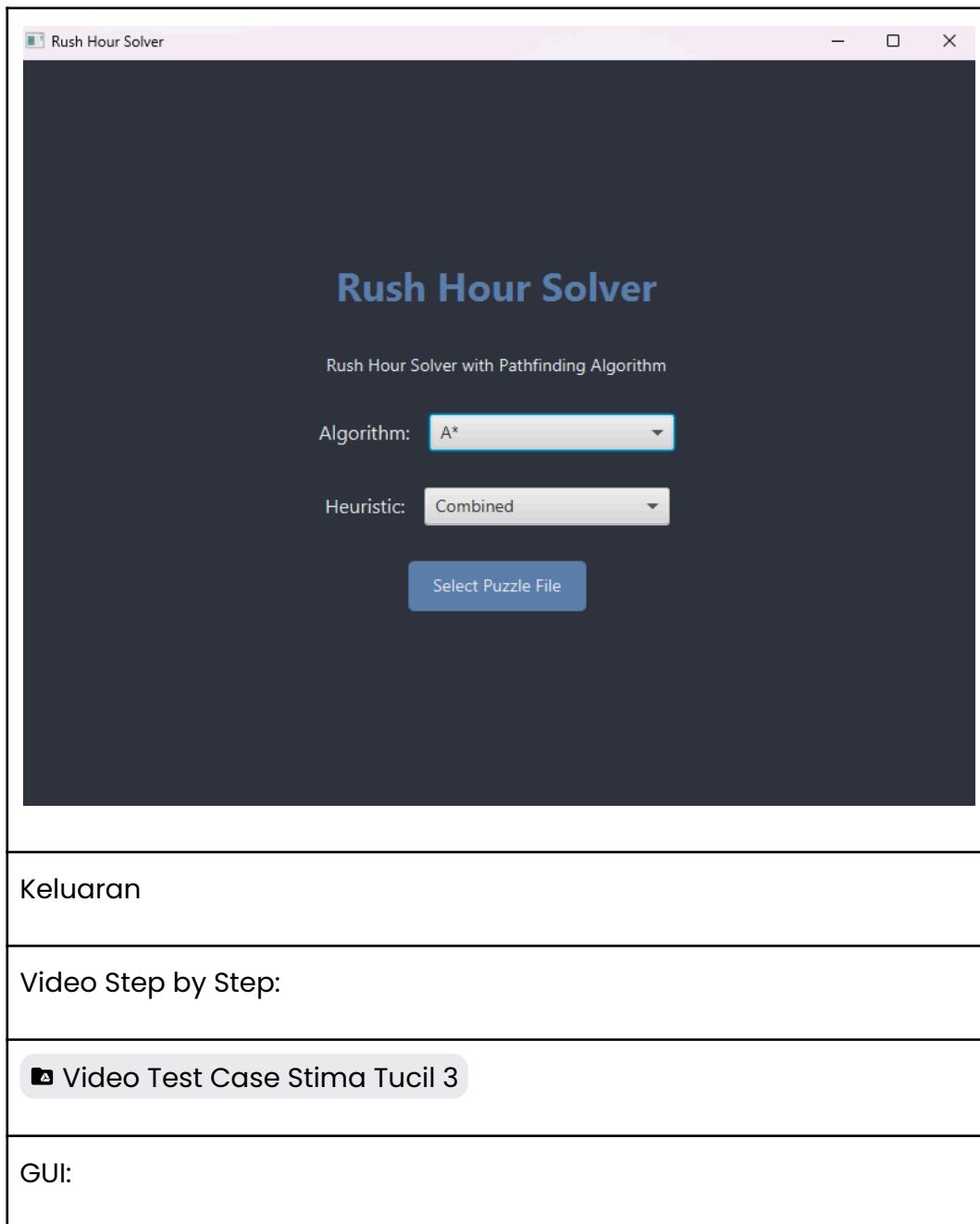


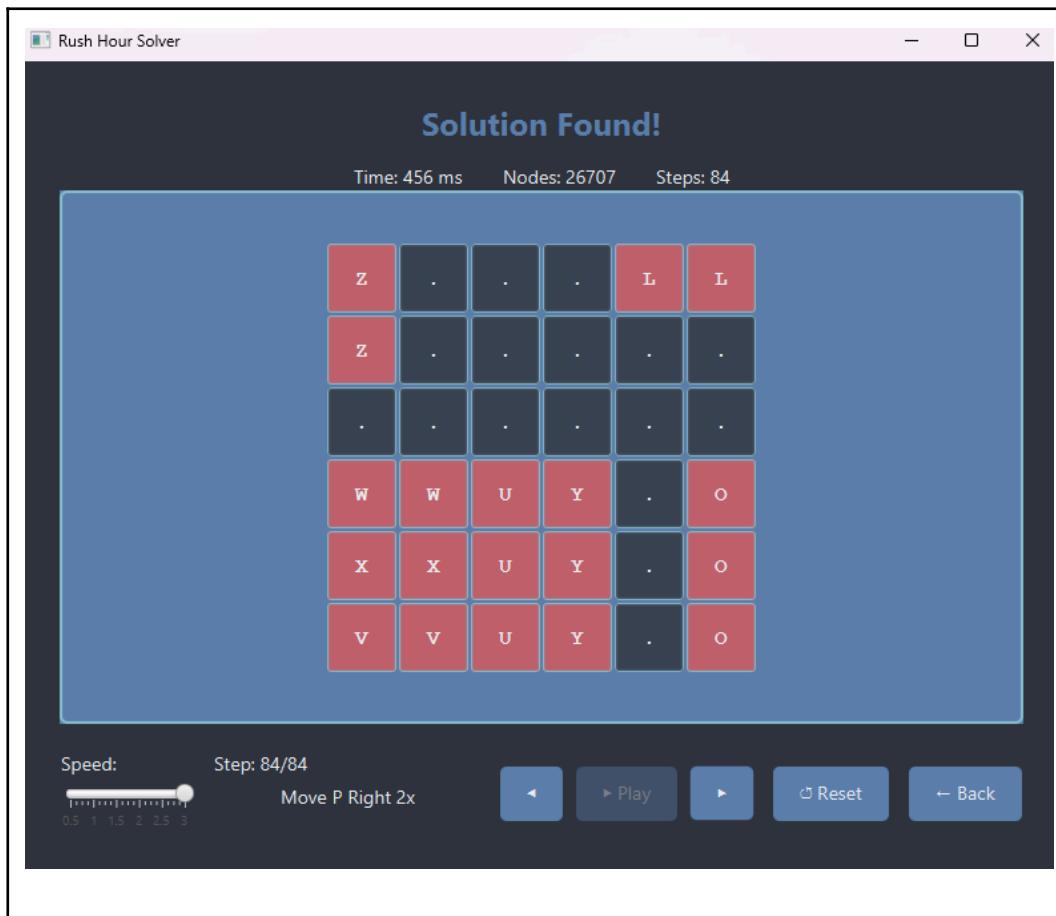
4.10 Test Case 2-3A: A* + Combine Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



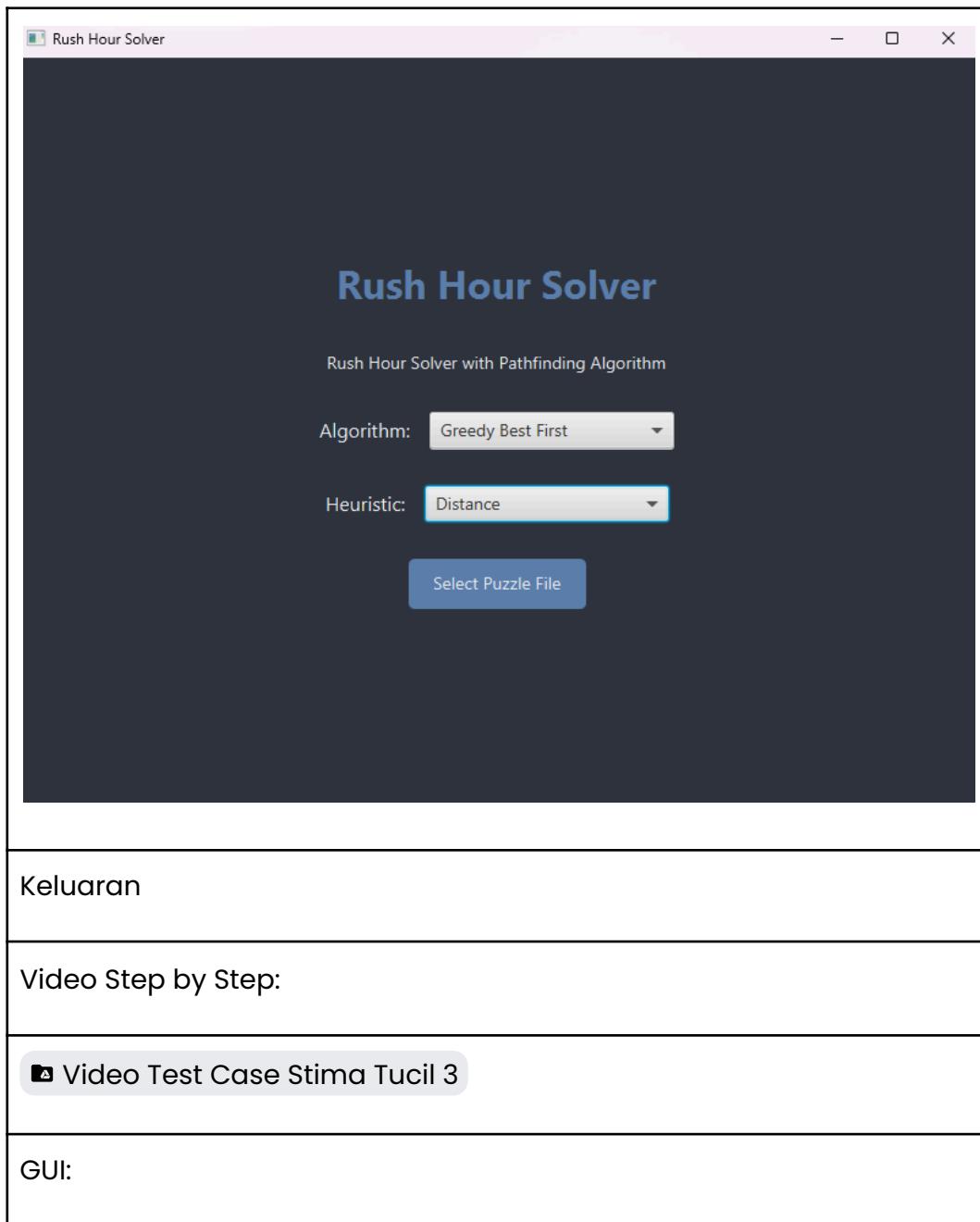


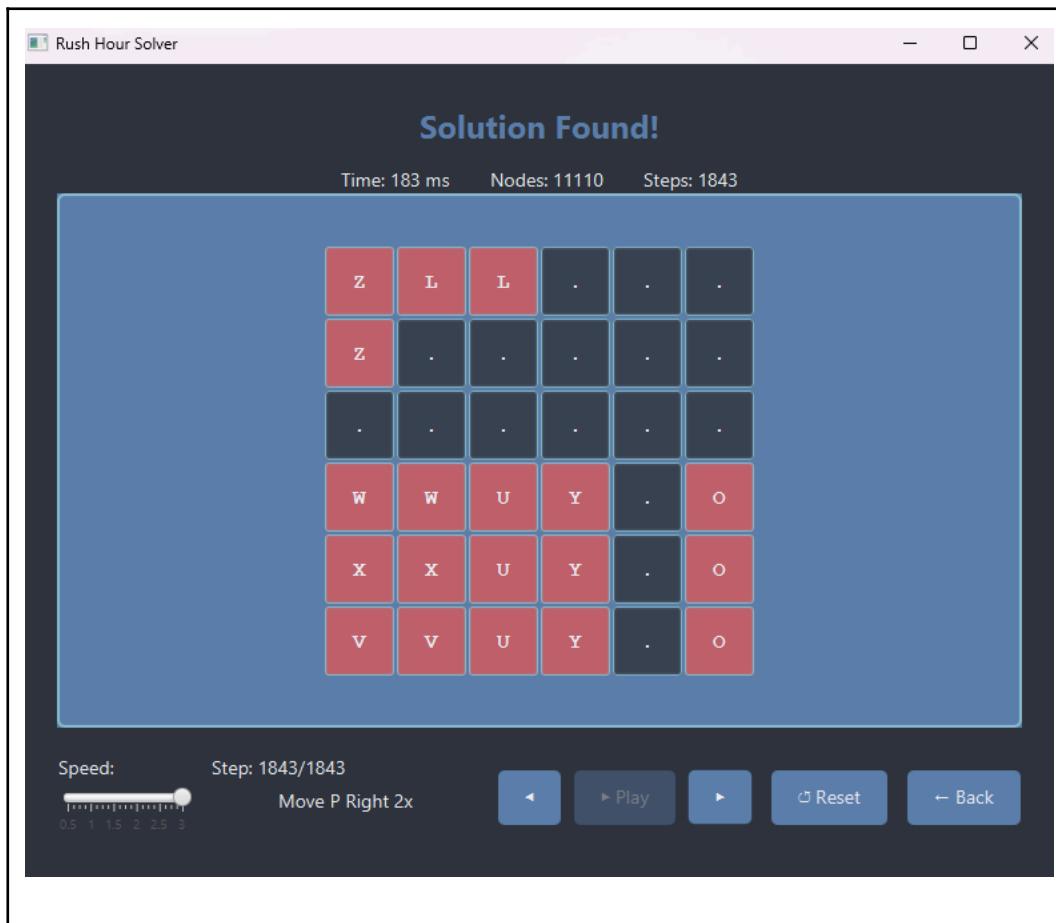
4.11 Test Case 2-2B: Greedy Best + Manhattan Distance Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



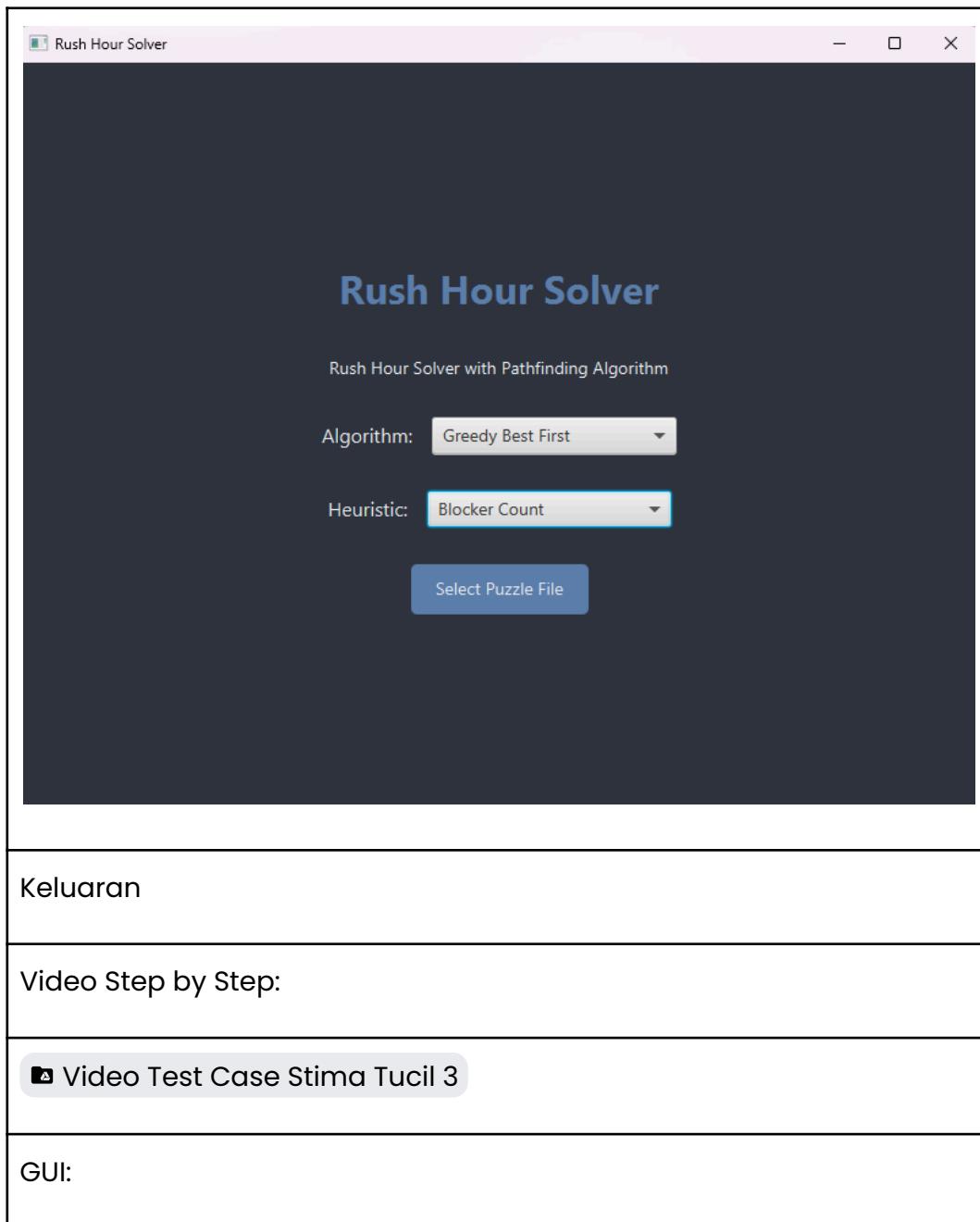


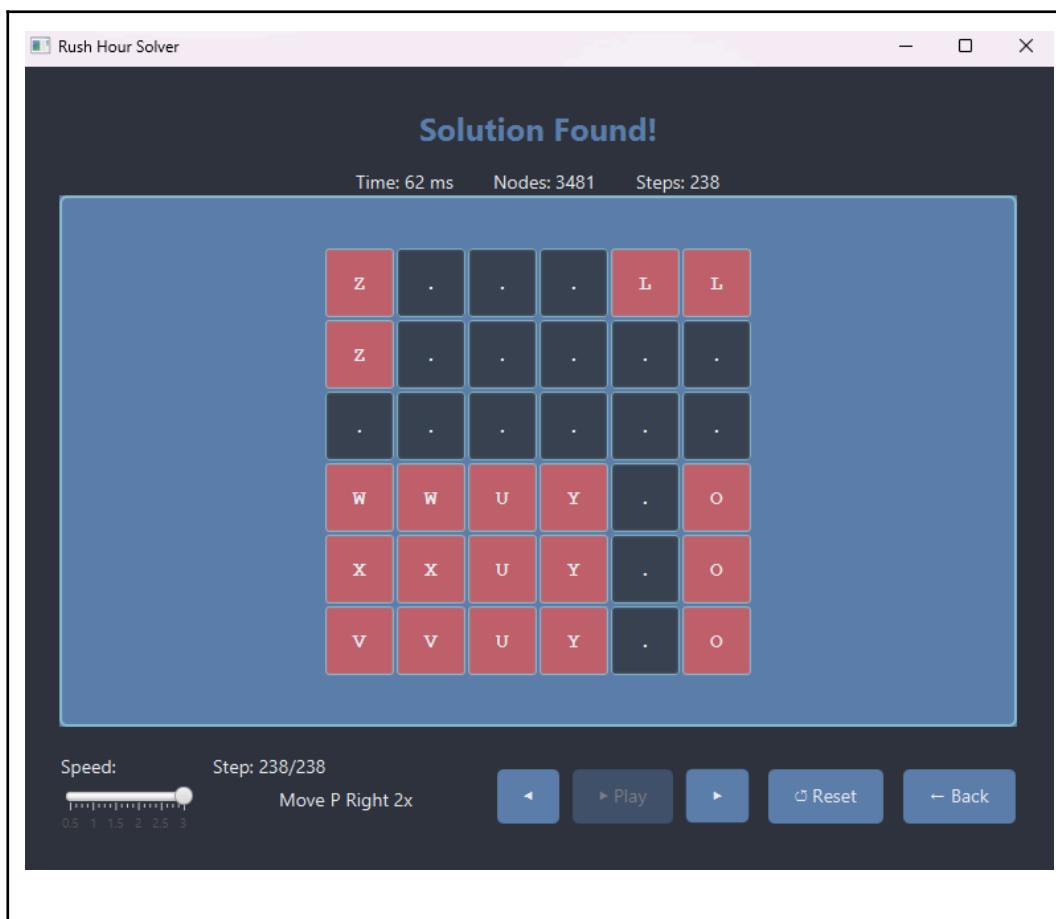
4.12 Test Case 2-2C: Greedy Best + Blocker Count Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



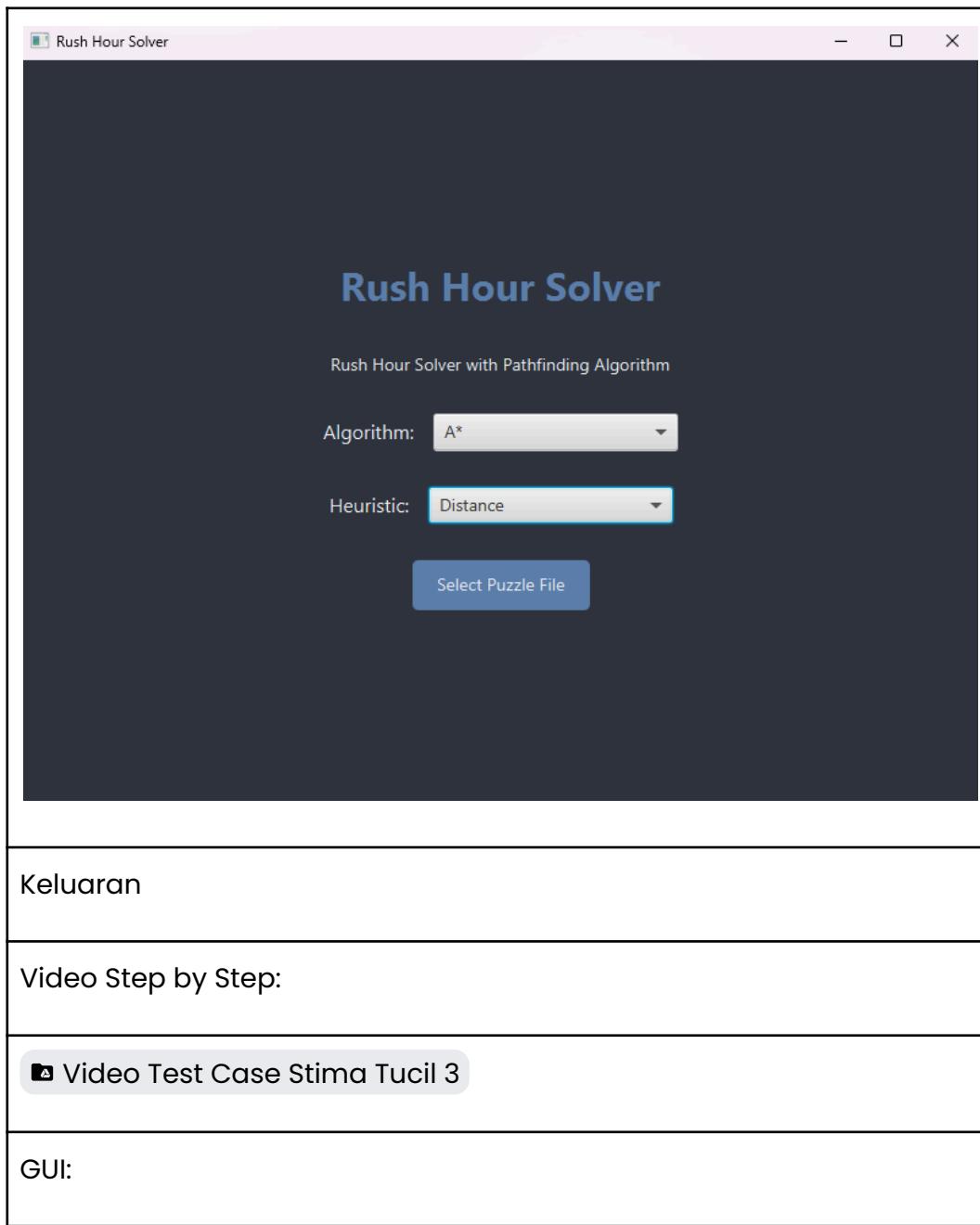


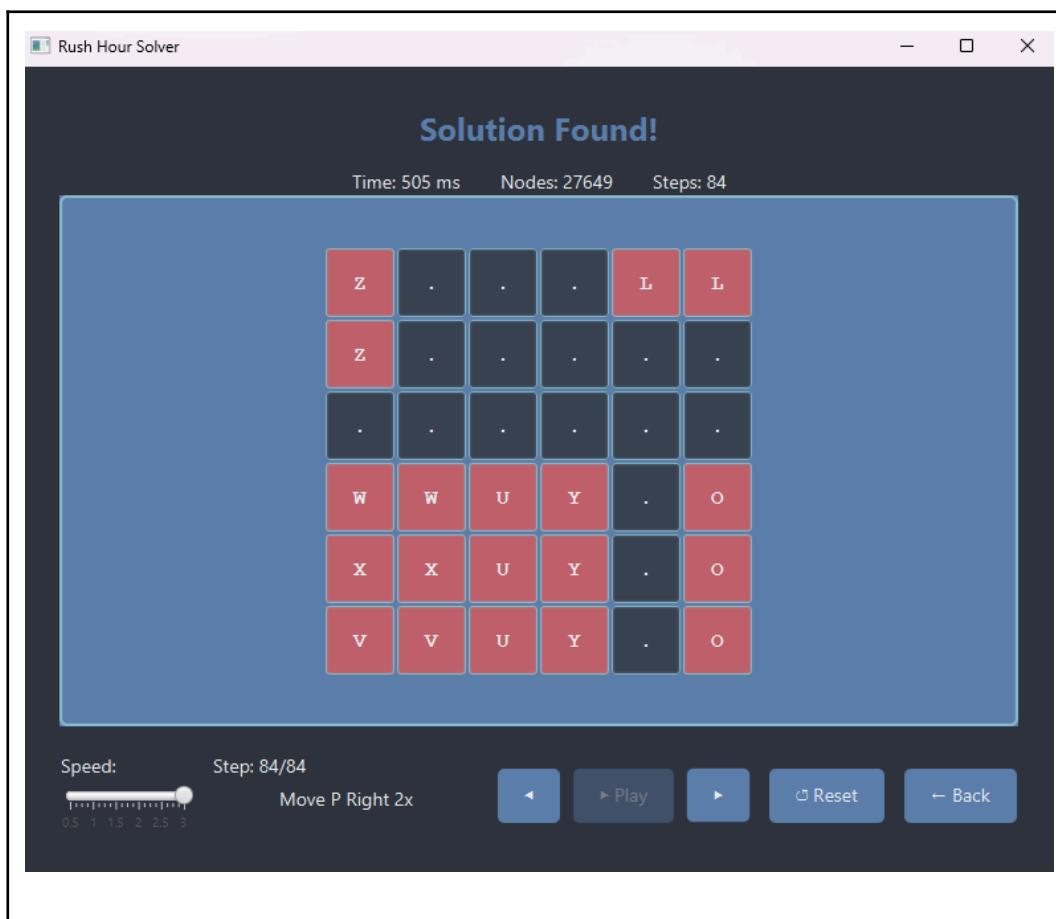
4.13 Test Case 2-3B: A* + Manhattan Distance Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



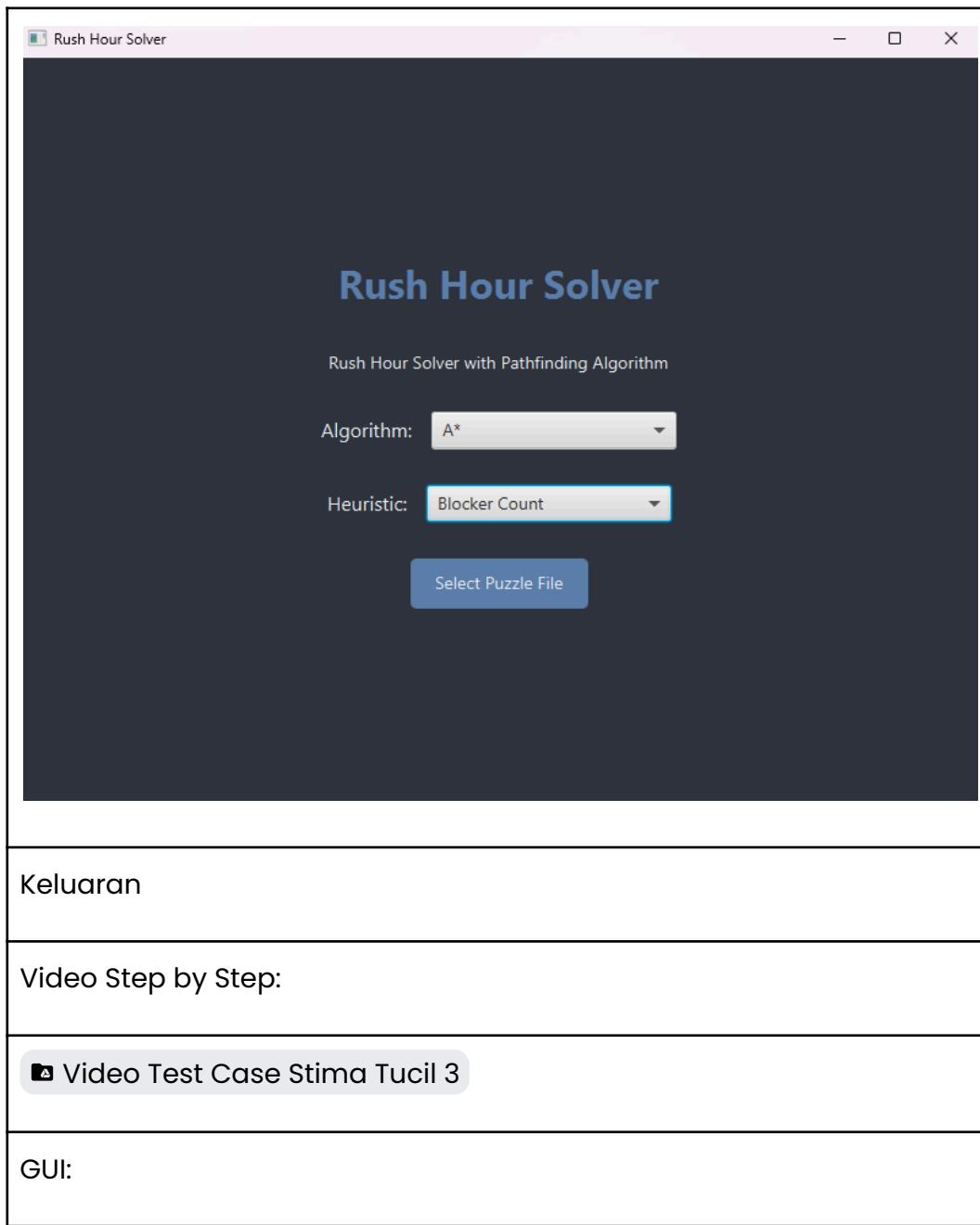


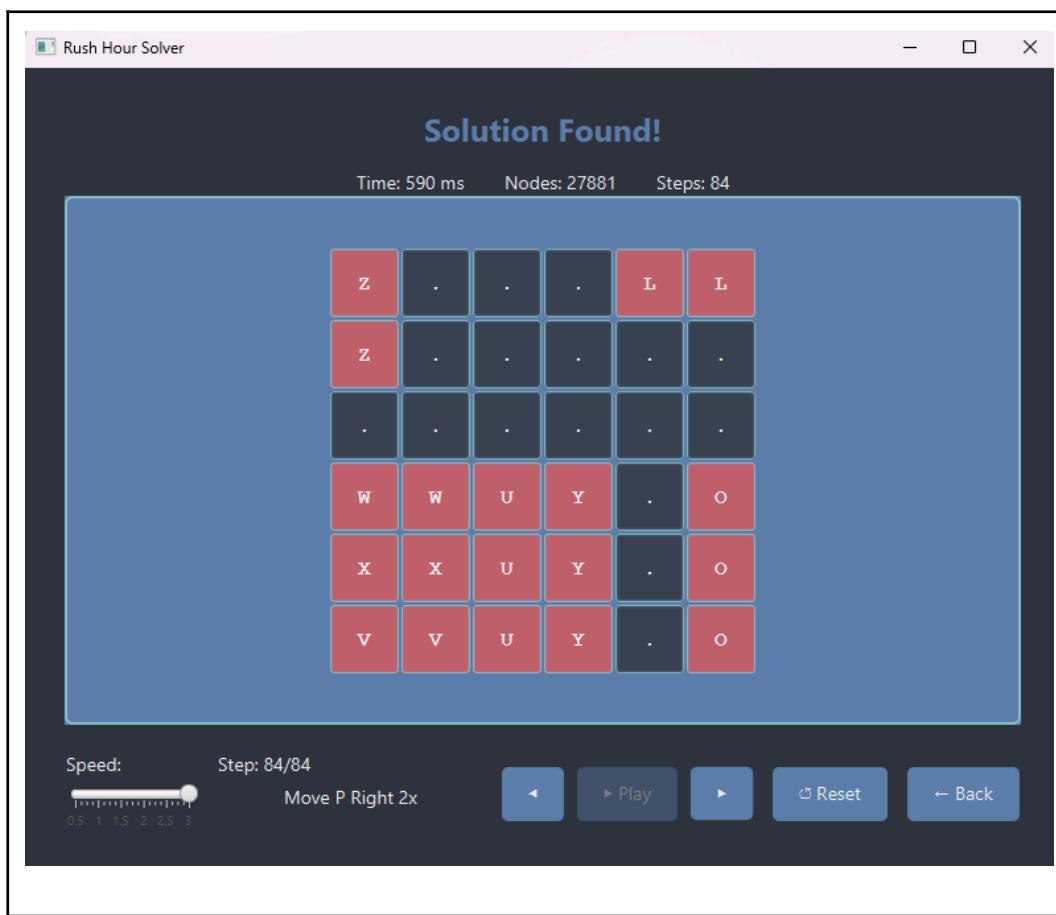
4.14 Test Case 2-3C: A*+ Blocker Count Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



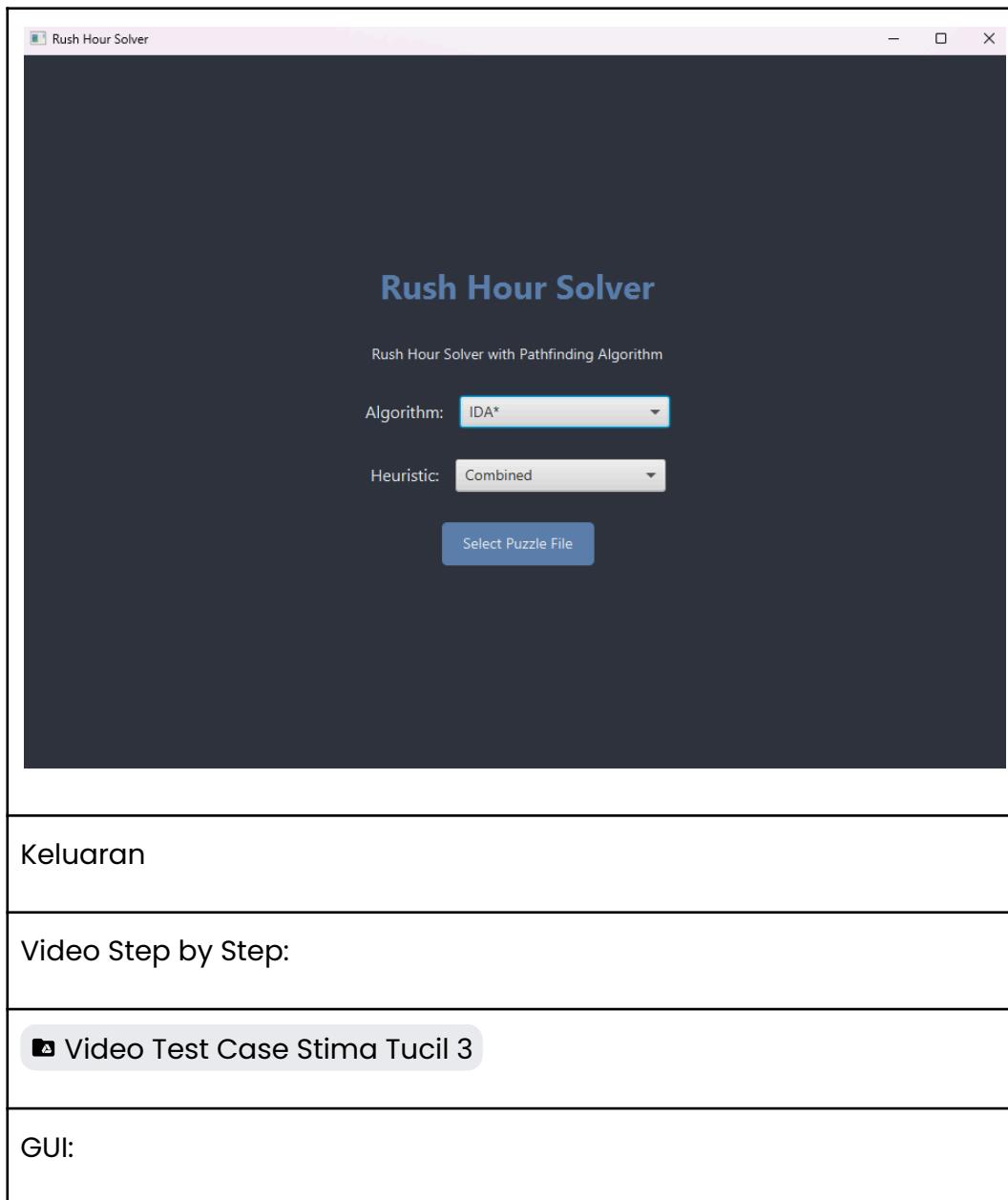


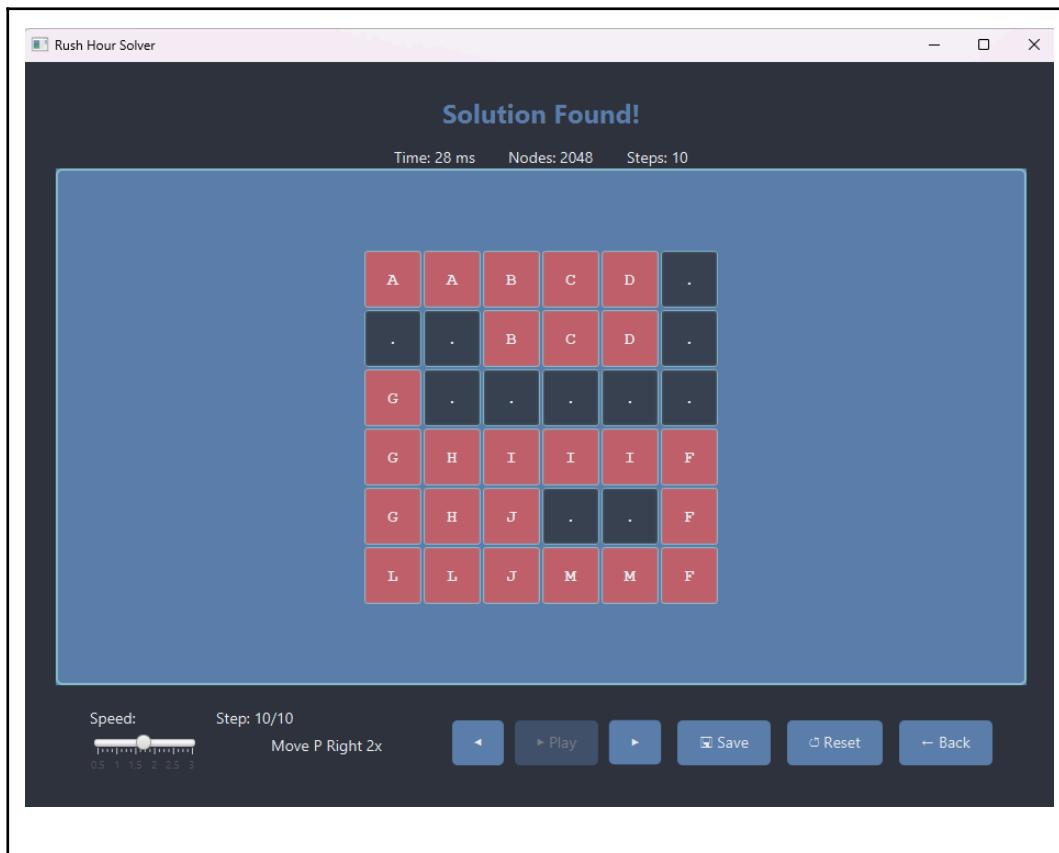
4.15 Test Case 1-4A: IDA*+ Combine Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



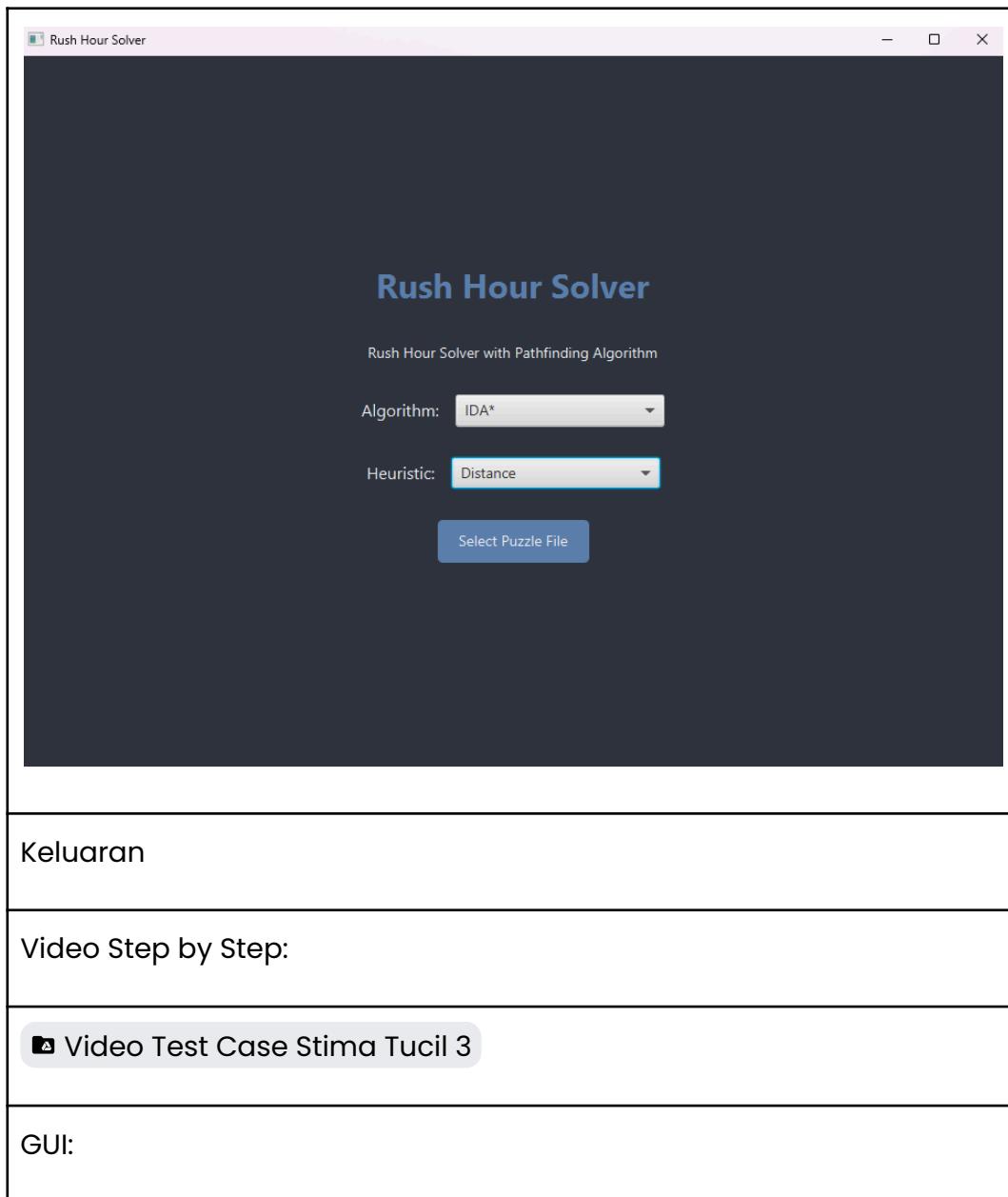


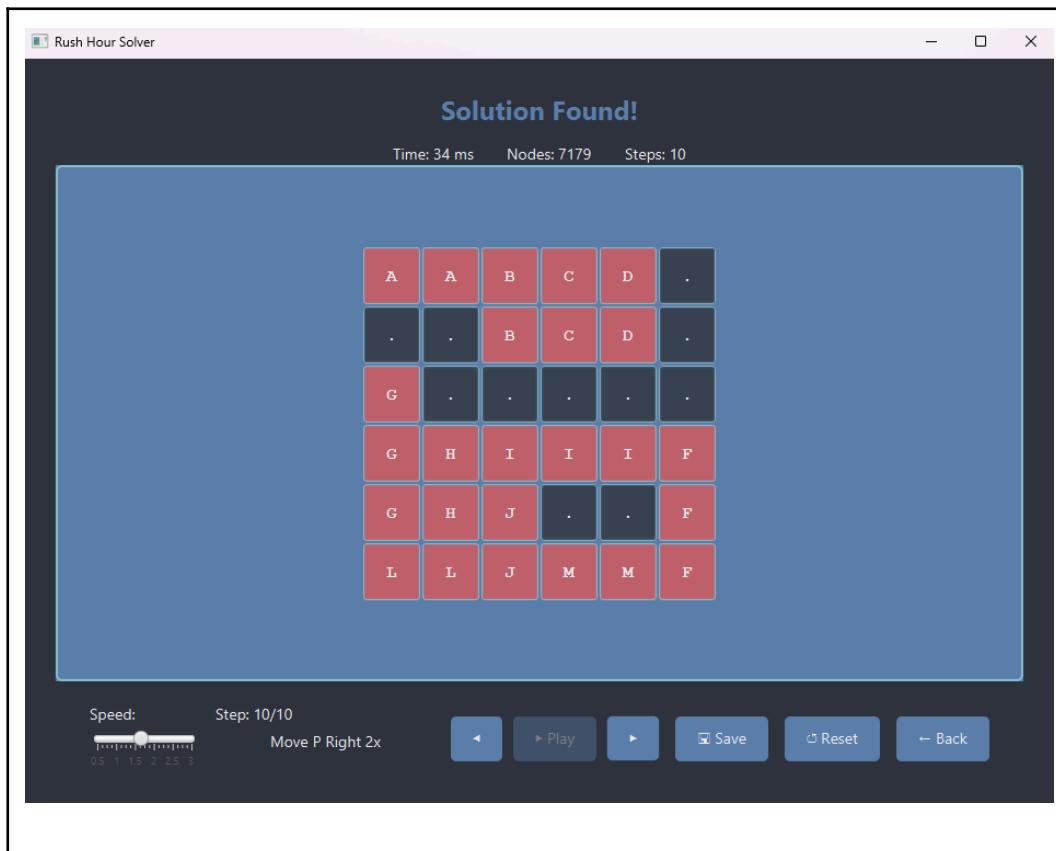
4.16 Test Case 1-4B: IDA*+ Manhattan Distance Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



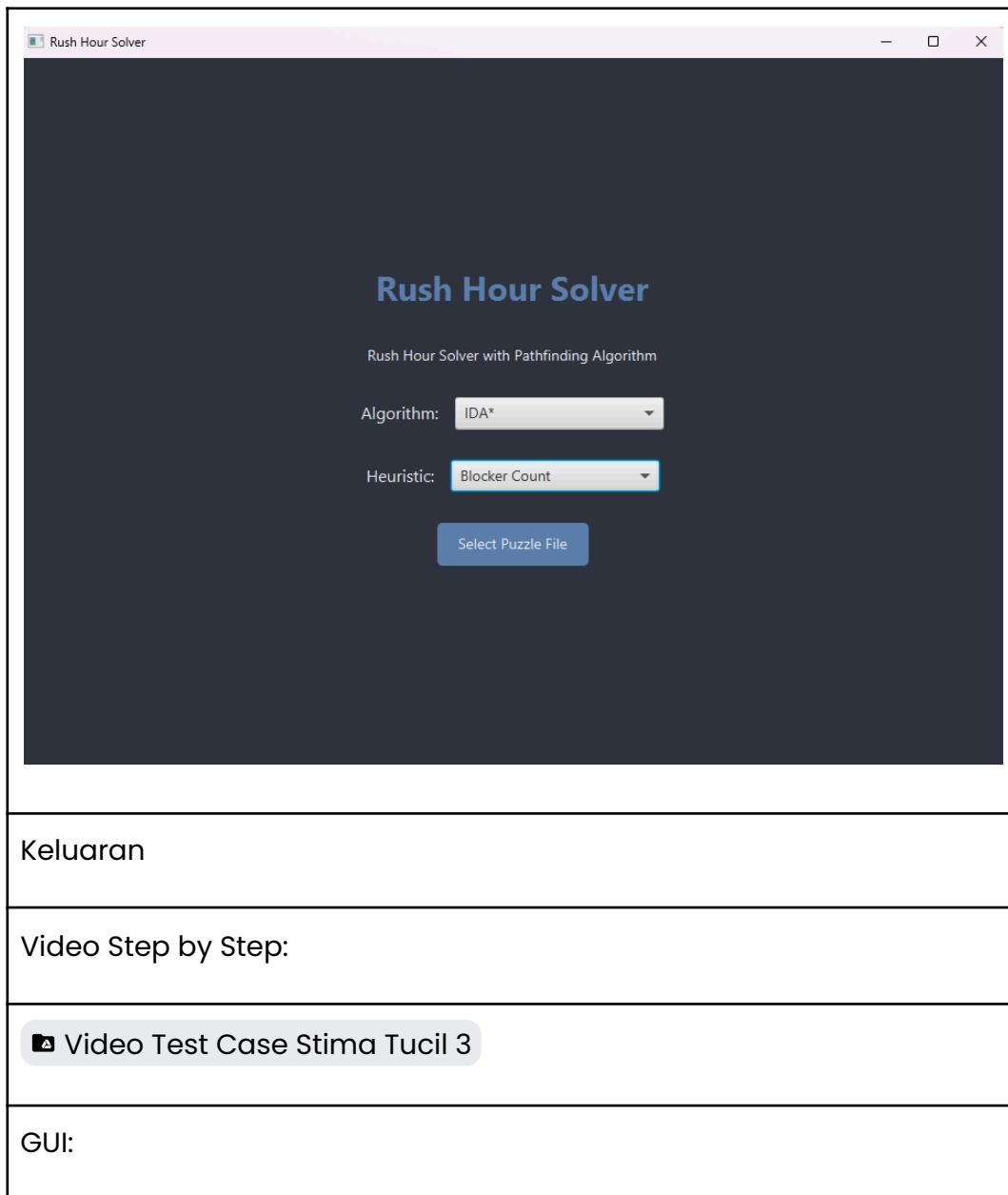


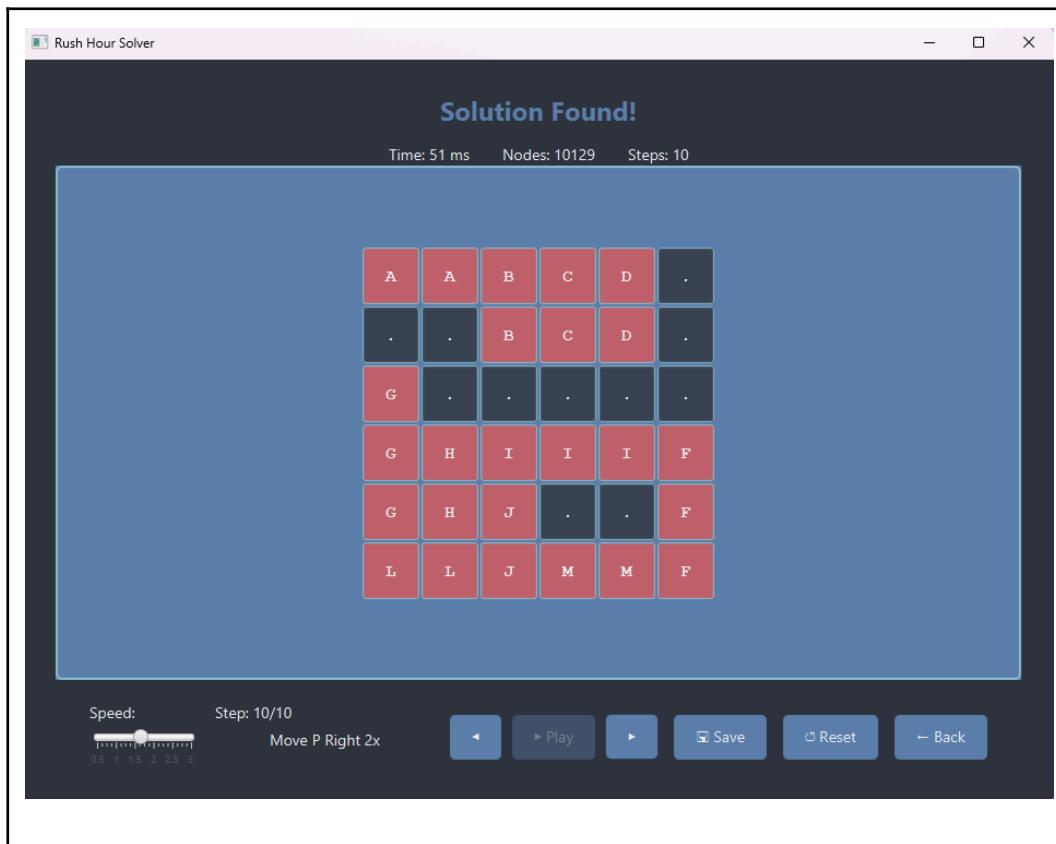
4.17 Test Case 1-4C: IDA*+ Blocker Count Heuristic

Masukan

```
test > test-case1.txt
1 6 6
2 11
3 AAB..F
4 ..BCDF
5 GPPCDFK
6 GH.III
7 GHJ...
8 LLJMM.
```

GUI:



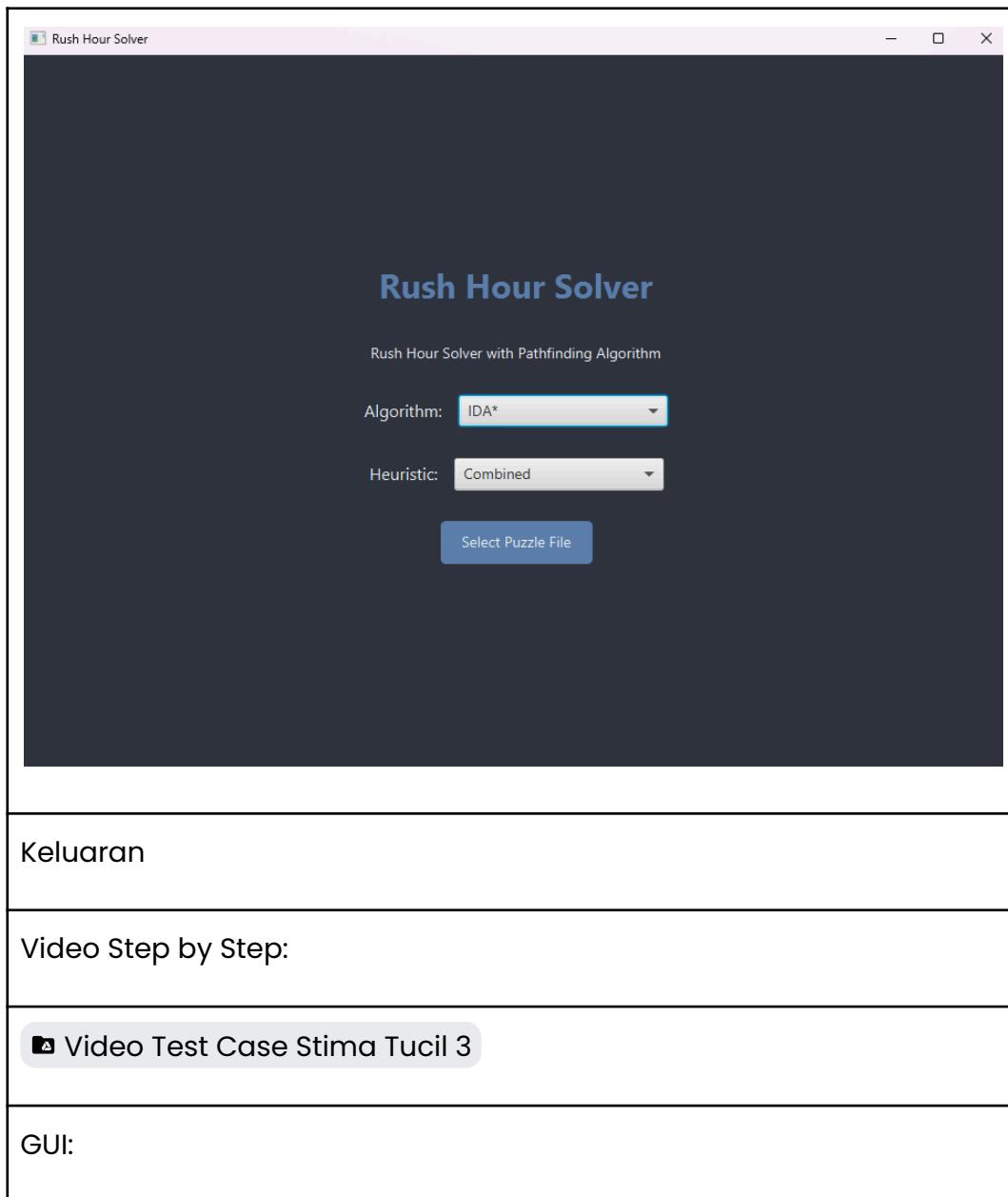


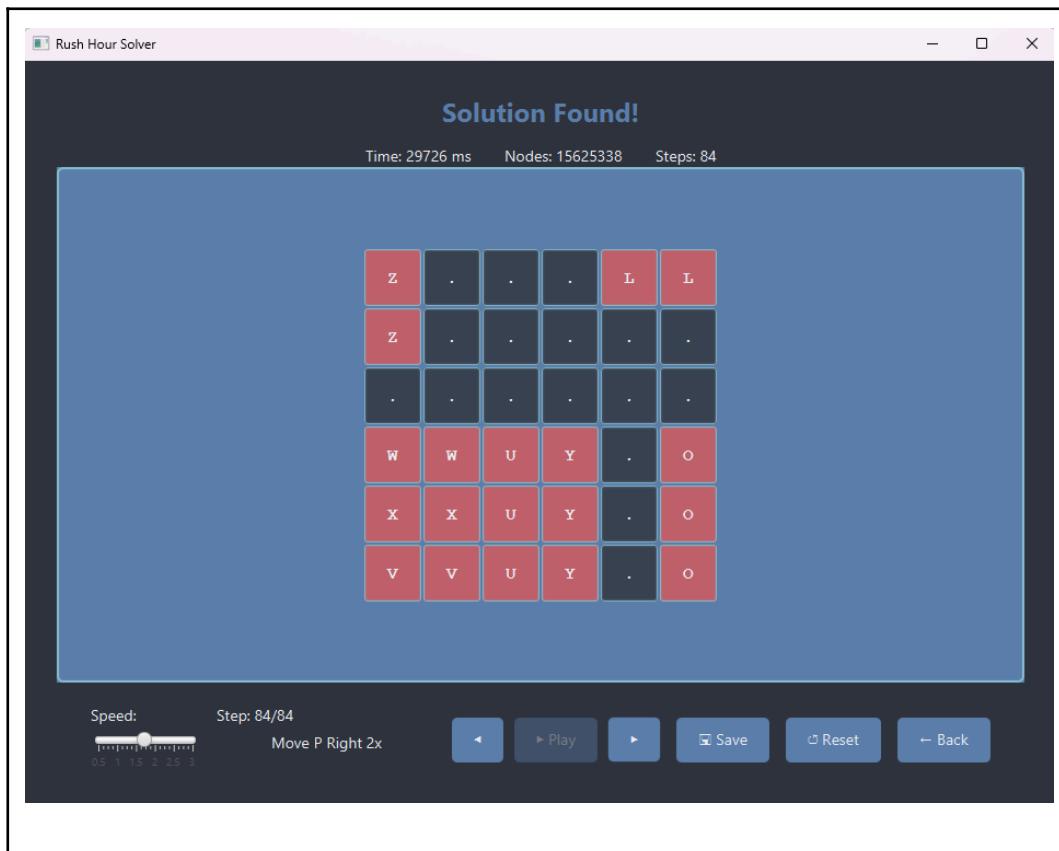
4.18 Test Case 2-4A: IDA*+ Combine Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



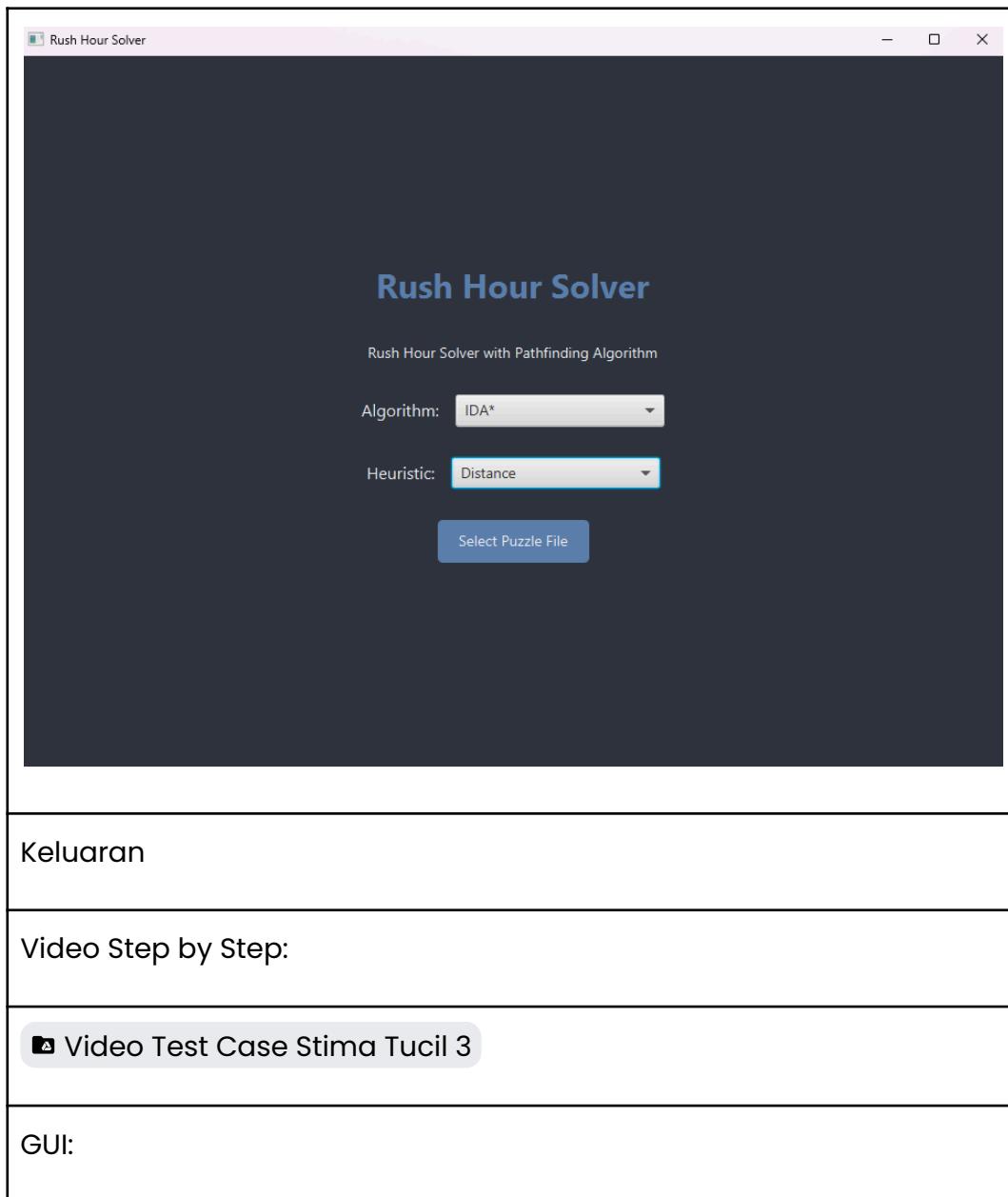


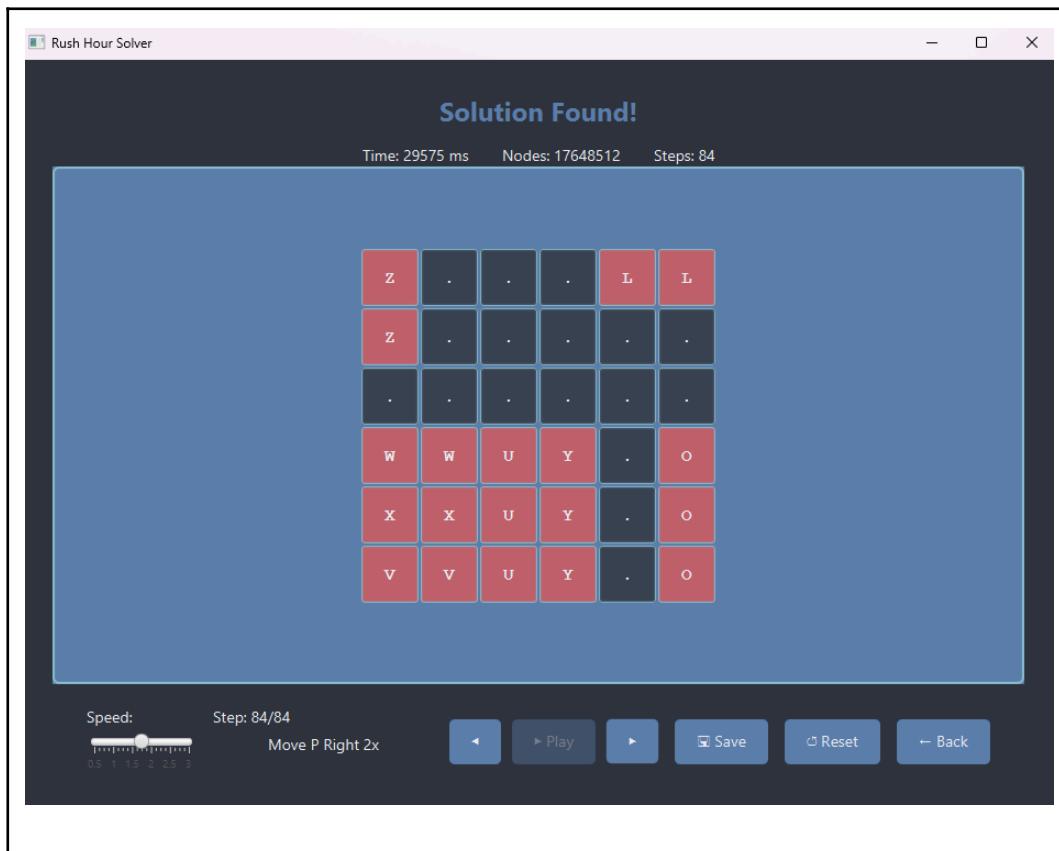
4.19 Test Case 2-4B: IDA*+ Manhattan Distance Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:



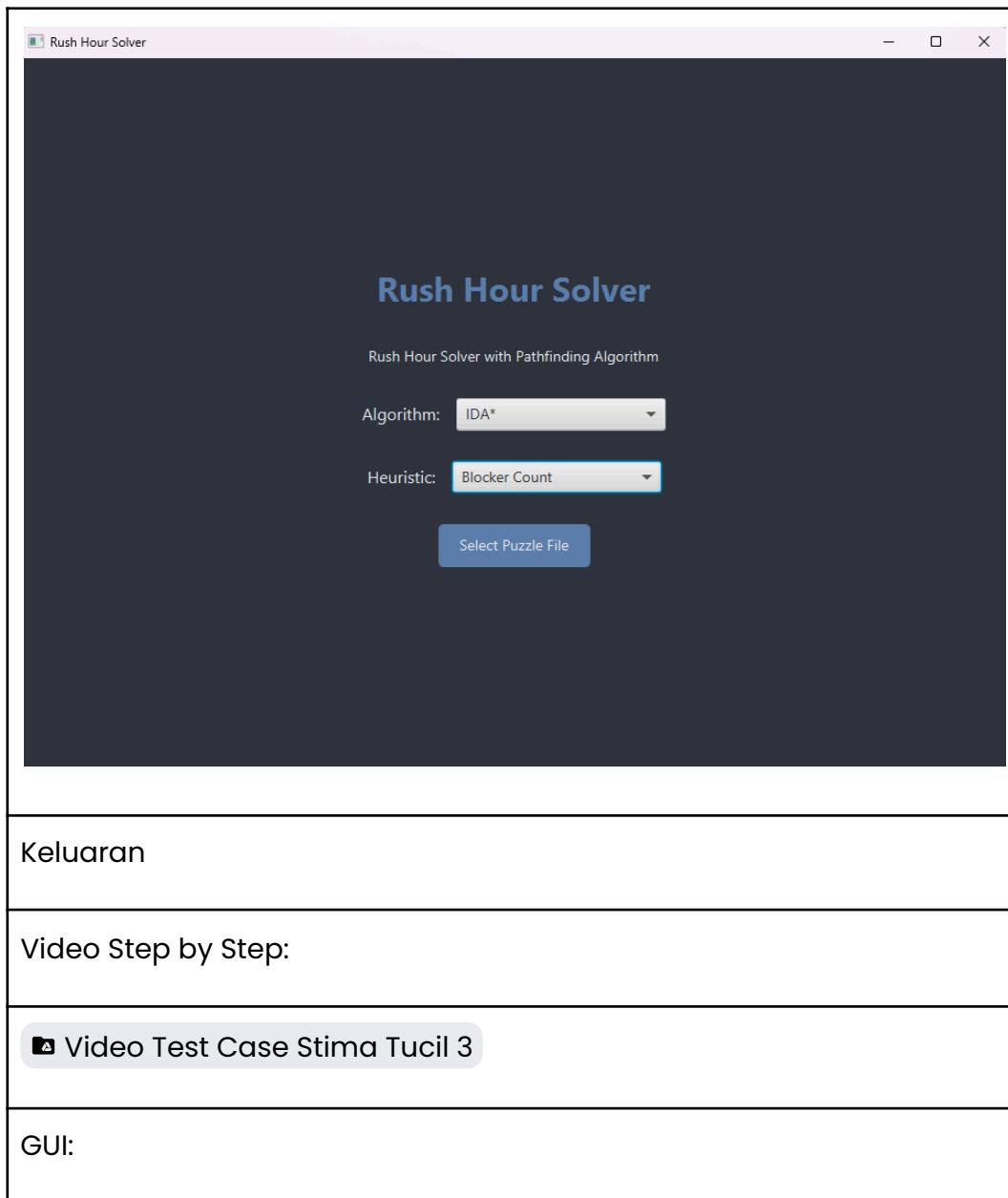


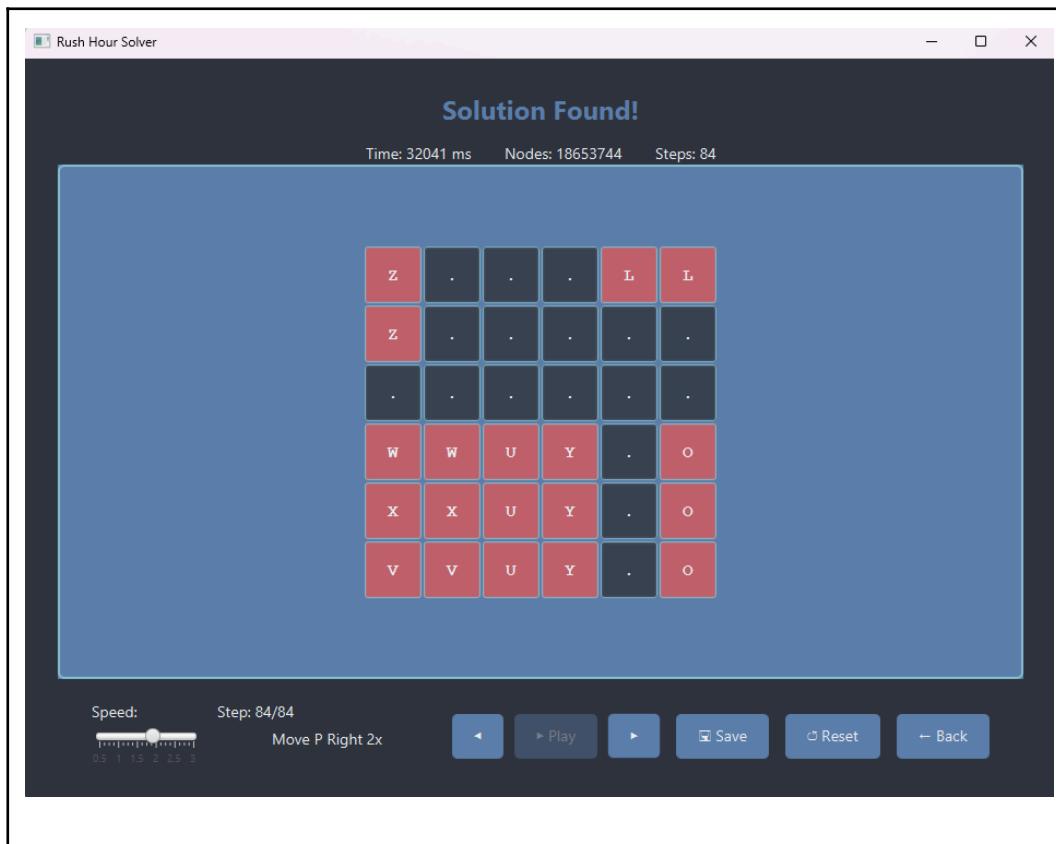
4.20 Test Case 2-4C: IDA*+ Blocker Count Heuristic

Masukan

```
test > test-case2.txt
1   6  6
2   8
3   ..ULLO
4   ..U..O
5   ..UPPOK
6   ...YW
7   ZXXY..
8   Z..YV
```

GUI:





4.21 Test Case 3-1: UCS

Masukan

```
test > test-case3.txt
1 7 9
2 √ 15
3 |   |   K
4 ..AAABBBB
5 .....CC
6 D...E..FF
7 D...E.GGG
8 D..HIIJJJ
9 LM.H..NOP
10 LMQQQQNOP
```

GUI:

Rush Hour Solver

Rush Hour Solver with Pathfinding Algorithm

Algorithm:

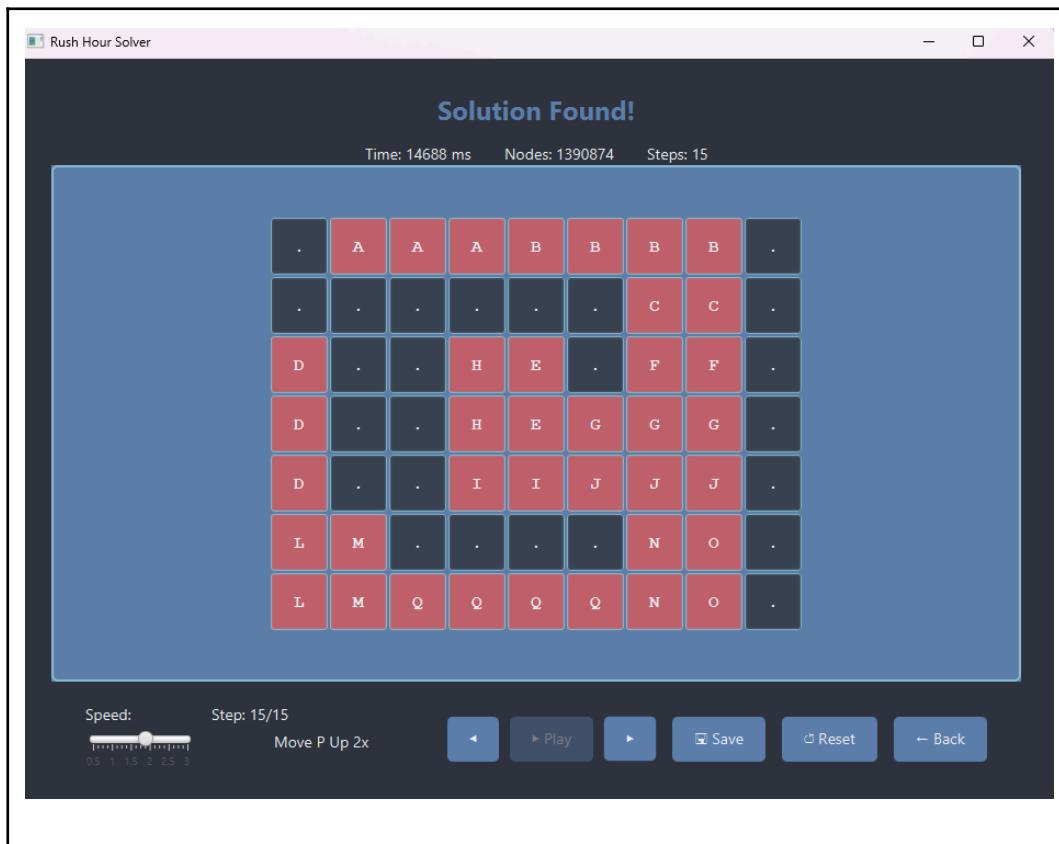
Select Puzzle File

Keluaran

Video Step by Step:

Video Test Case Stima Tucil 3

GUI:

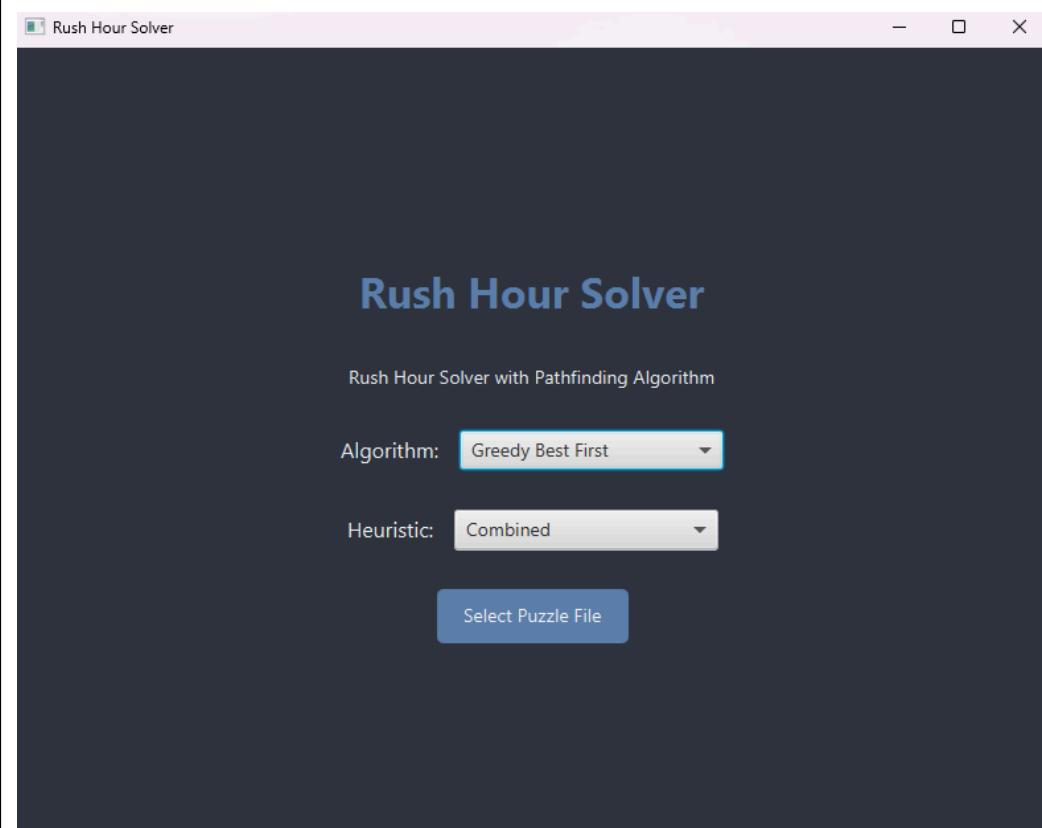


4.22 Test Case 3-2A: Greedy + Combined Heuristic

Masukan

```
test > test-case3.txt
 1  7 9
 2  v 15
 3  |   |   K
 4  ..AAABBBB
 5  .....CC
 6  D...E..FF
 7  D...E.GGG
 8  D..HIIJJJ
 9  LM.H..NOP
10 LMQQQQNQP
```

GUI:

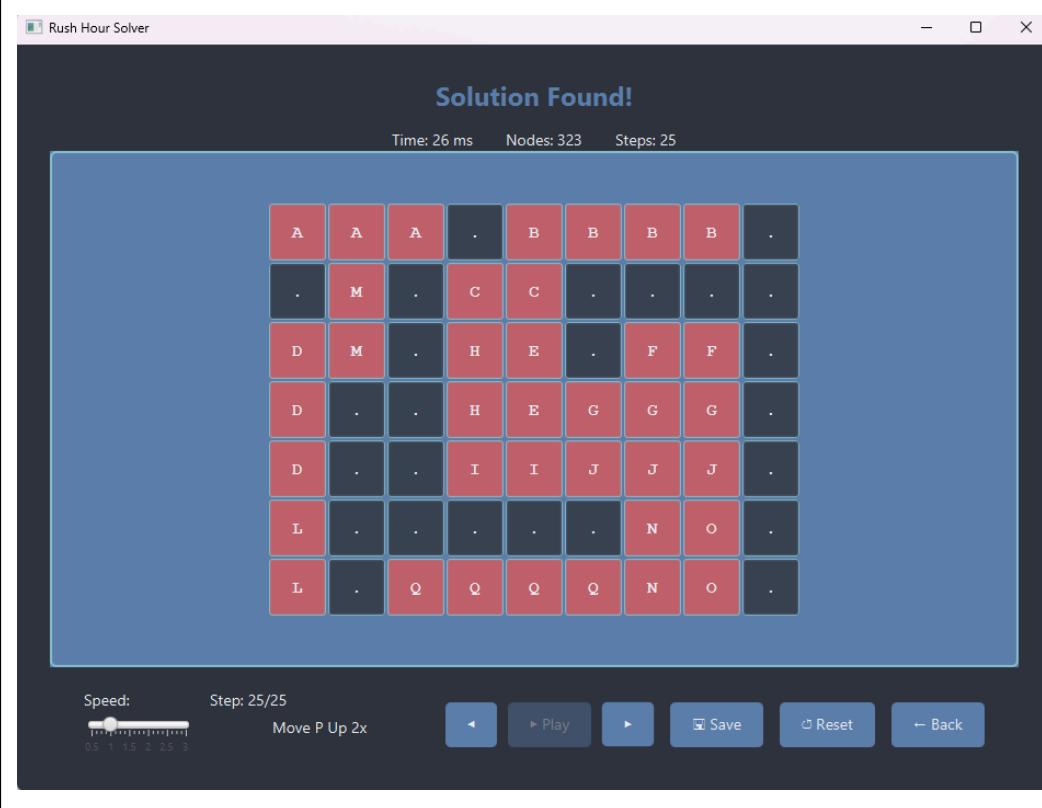


Keluaran

Video Step by Step:

[▶ Video Test Case Stima Tucil 3](#)

GUI:

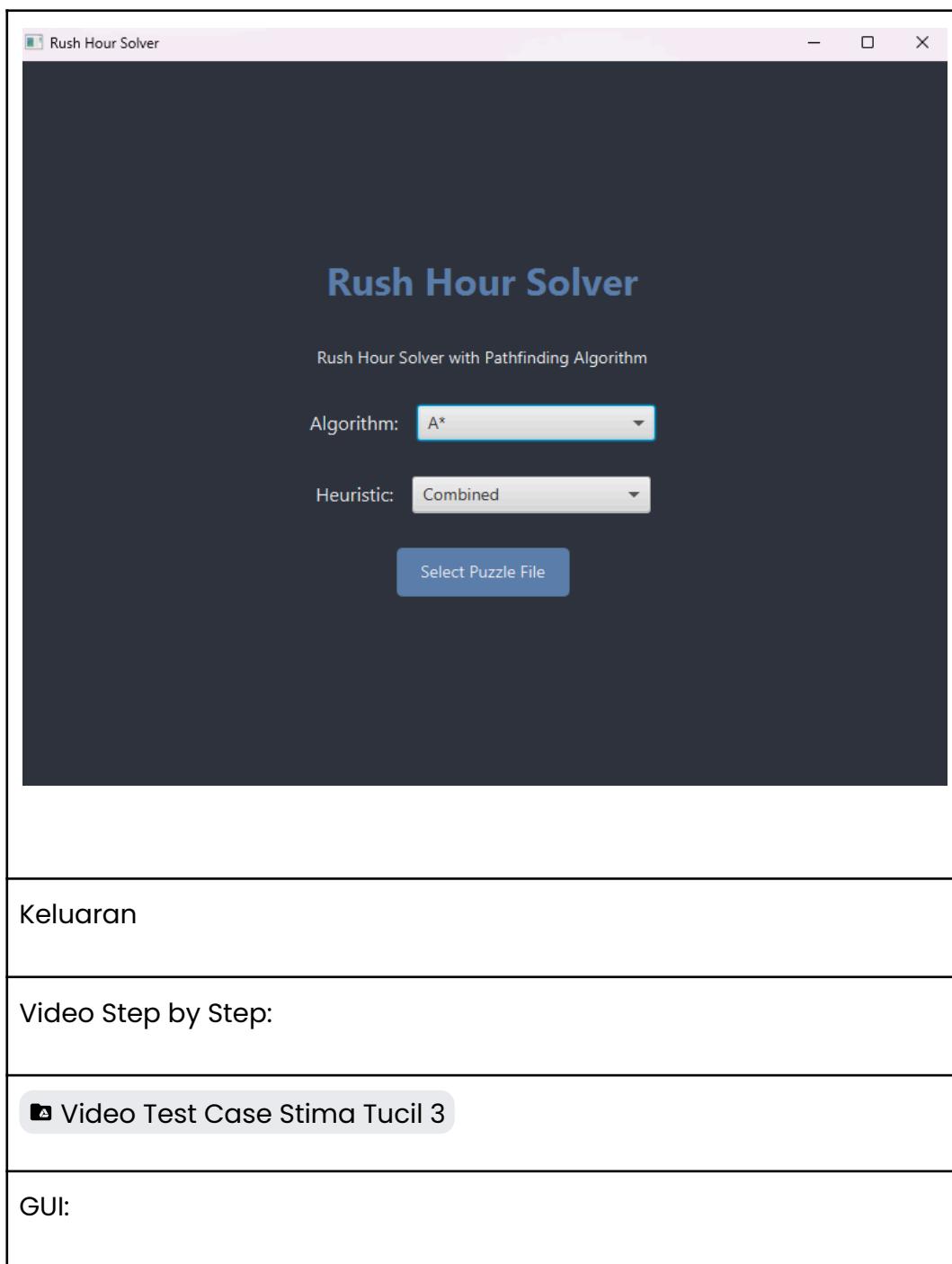


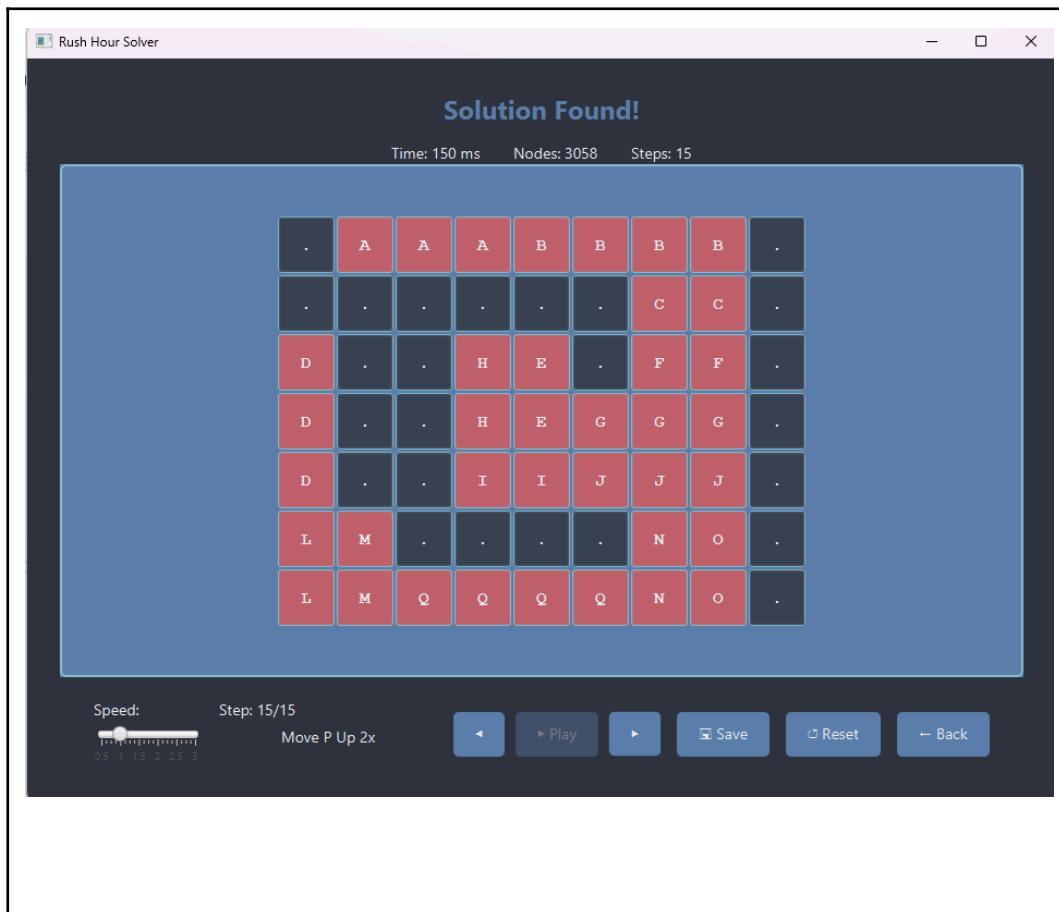
4.23 Test Case 3-3A: A* + Combined Heuristic

Masukan

```
test > test-case3.txt
1 7 9
2 √ 15
3 |   |   K
4 ..AAABBBB
5 .....CC
6 D...E..FF
7 D...E.GGG
8 D..HIIJJJ
9 LM.H..NOP
10 LMQQQQNOP
```

GUI:



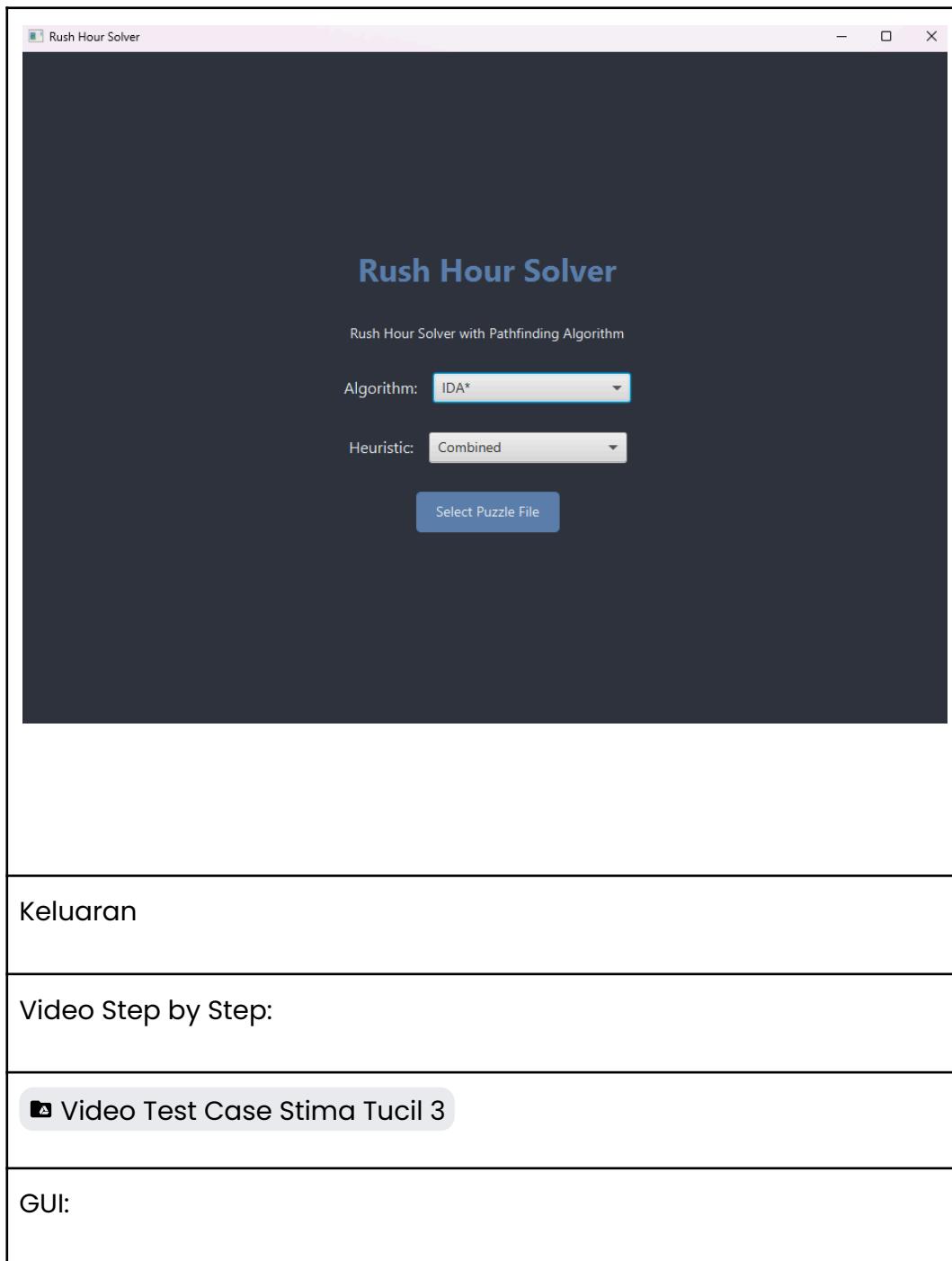


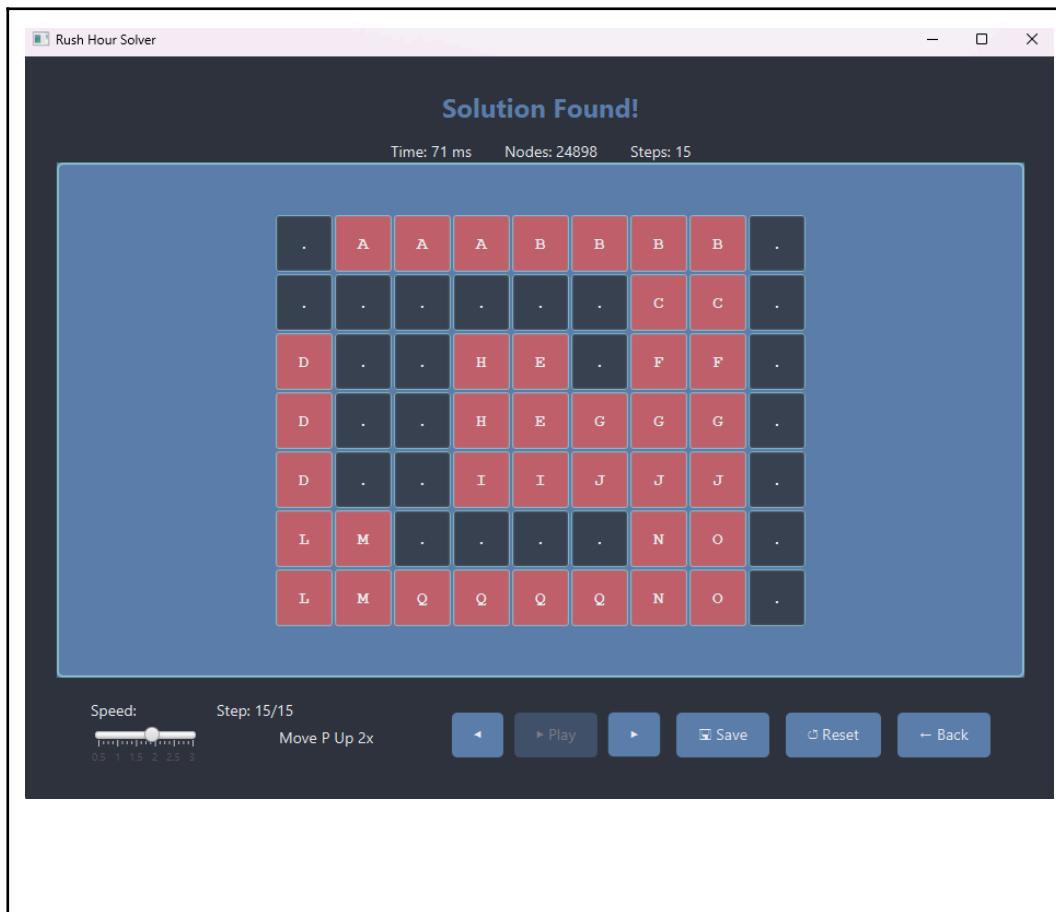
4.24 Test Case 3-4A: IDA* + Combined Heuristic

Masukan

```
test > test-case3.txt
1 7 9
2 √ 15
3 |   |   K
4 ..AAABBBB
5 .....CC
6 D...E..FF
7 D...E.GGG
8 D..HIIJJJ
9 LM.H..NOP
10 LMQQQQNOP
```

GUI:





BAB V

ANALISIS DAN PEMBAHASAN

5.1 Analisis Hasil Percobaan

Pengujian dilakukan menggunakan beberapa test case yang telah tercantum di Bab 4. Hasil pengujian test case 1 menunjukkan hasil berikut:

Hasil Test Case 1				
Ukuran Papan: Normal (6x6)			Kepadatan: Cukup Padat (8/36 Cell Kosong)	
Algoritma	Heuristik	Nodes Explored	Steps	Times (ms)
UCS	-	1910	10	19
Greedy Best	Combine	355	102	11
	Distance	371	110	6
	Blocker	43	11	1
A*	Combine	342	10	11
	Distance	989	10	11
	Blocker	1292	10	18

Pengujian dilakukan pada konfigurasi papan berukuran normal (6x6) dengan tingkat kepadatan padat, dengan 11 buah kendaraan dan tersisa 8 dari 36 cell yang kosong. Evaluasi dilakukan terhadap tiga algoritma pencarian, yaitu Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), dan A*, masing-masing dengan varian heuristik: Combine, Distance, dan Blocker. Hasil menunjukkan bahwa UCS

mengeksplorasi jumlah node terbanyak (1910 node), dengan waktu eksekusi 19 ms dan menghasilkan solusi sepanjang 10 langkah. Hal ini wajar karena UCS tidak menggunakan informasi heuristik dan mengeksplorasi seluruh kemungkinan berdasarkan biaya kumulatif, sehingga tidak efisien dalam konteks masalah ini.

Sebaliknya, algoritma Greedy Best-First Search menunjukkan kinerja bervariasi tergantung pada heuristik yang digunakan. Heuristik Blocker terbukti paling efisien secara waktu (1 ms) dan eksplorasi node (43 node), namun menghasilkan solusi paling panjang (18 langkah), menunjukkan bahwa fokus murni pada hambatan di depan kendaraan target dapat mengarah pada solusi yang cepat ditemukan tetapi tidak optimal. Heuristik Distance dan Combine pada Greedy Best menghasilkan solusi yang lebih pendek (6 dan 11 langkah), namun dengan eksplorasi node dan waktu yang sedikit lebih tinggi memperlihatkan adanya trade-off antara kualitas solusi dan efisiensi pencarian.

A* menunjukkan performa yang lebih seimbang. Dengan heuristik Combine, A* mengeksplorasi hanya 342 node dan mencapai solusi paling optimal diantara semuanya sepanjang 10 langkah dalam 11 ms, lebih baik dibandingkan UCS dalam efisiensi dan setara dalam kualitas solusi. A* dengan heuristik Distance dan Blocker menunjukkan variasi jumlah node yang lebih besar (989 dan 1292 node), tetapi tetap menjaga panjang solusi pada 10 hingga 11 langkah. Ini membuktikan bahwa A* dapat menjaga keseimbangan antara efisiensi pencarian dan kualitas solusi dengan menggabungkan biaya aktual dan estimasi heuristik.

Hasil-hasil test case 1 ini menegaskan bahwa pemilihan algoritma dan heuristik sangat memengaruhi efisiensi dan efektivitas penyelesaian puzzle Rush Hour. A* dengan heuristik gabungan (Combine) menawarkan kinerja paling optimal dalam skenario ini, sedangkan Greedy dengan Blocker unggul dalam kecepatan namun mengorbankan kualitas solusi.

Hasil pengujian test case 2 menunjukkan hasil berikut:

Hasil Test Case 2				
Ukuran Papan: Normal (6x6)			Kepadatan: Sedang (15/36 Cell Kosong)	
Algoritma	Heuristik	Nodes Explored	Steps	Times (ms)
UCS	-	28223	84	220
Greedy Best	Combine	5731	232	115
	Distance	11110	1843	183
	Blocker	3481	238	62
A*	Combine	26707	84	456
	Distance	27649	84	505
	Blocker	27881	84	590

Pengujian kedua dilakukan pada papan berukuran normal (6x6) dengan kepadatan sedang, di mana hanya 15 dari 36 sel yang kosong. Konfigurasi ini memiliki tingkat kompleksitas yang relatif tinggi karena ruang gerak terbatas, meskipun hanya melibatkan 8 kendaraan. Hasil pengujian menunjukkan bahwa algoritma Uniform Cost Search (UCS) kembali menghasilkan solusi optimal dengan 84 langkah, tetapi dengan harga yang sangat mahal dalam hal eksplorasi node dan waktu komputasi, mencapai 28.223 node dan waktu eksekusi sebesar 220 ms. Ini menegaskan kembali bahwa UCS, meskipun optimal, sangat tidak efisien pada konfigurasi yang kompleks karena ketidaktahuan terhadap arah solusi.

Greedy Best-First Search menunjukkan efisiensi tinggi dalam waktu komputasi, namun dengan biaya signifikan terhadap kualitas solusi. Heuristik Blocker mengeksplorasi jumlah node paling sedikit

(3.481 node) dan mencatat waktu tercepat (62 ms), tetapi menghasilkan solusi cukup panjang, yaitu 238 langkah. Heuristik Combine dan Distance juga menunjukkan pola serupa, dengan solusi masing-masing sepanjang 232 dan 1843 langkah, meskipun tetap lebih cepat dibanding UCS dalam hal waktu eksekusi. Khusus pada heuristik Distance, tampak kelemahan mendasar dari Greedy Best-First Search, yaitu kecenderungan terjebak pada plateau atau jalur yang tampak menjanjikan namun stagnan. Hal ini menegaskan bahwa pendekatan greedy yang hanya mempertimbangkan satu aspek dari solusi cenderung mengarah pada eksplorasi jalur suboptimal dan solusi yang jauh dari optimal.

A* menunjukkan keunggulan dalam menjaga optimalitas solusi (84 langkah, sama seperti UCS) sekaligus memberikan pendekatan yang lebih bagus dalam pencarian dibanding UCS. Namun, efisiensinya menurun drastis dalam konteks ini. Ketiga varian heuristik yang digunakan, Combine, Distance, dan Blocker, semuanya mengeksplorasi lebih dari 26.000 node, dengan waktu eksekusi masing-masing mencapai 456, 505, dan 590 ms. Ini menunjukkan bahwa meskipun A* mampu menjaga kualitas solusi, kompleksitas heuristik yang digunakan dalam papan dengan ruang gerak terbatas memperlambat proses pencarian secara signifikan.

Secara keseluruhan, analisis pada test case 2 ini memperlihatkan bahwa dalam kondisi papan dengan kepadatan menengah dan konfigurasi kendaraan yang menghambat pergerakan, algoritma pencarian dengan pendekatan heuristik sempit (seperti GBFS) memberikan solusi yang cepat namun tidak efektif. Sebaliknya, pendekatan berbasis pencarian menyeluruh seperti UCS dan A* mampu mencapai solusi optimal, namun memerlukan eksplorasi node dan waktu komputasi yang jauh lebih besar. A* masih merupakan pilihan terbaik dalam hal keseimbangan antara optimalitas dan strategi eksplorasi, tetapi efisiensinya sangat dipengaruhi oleh kompleksitas spasial dari konfigurasi papan.

Selain UCS, GBFS, dan A*, algoritma IDA* juga diujikan pada test case 1 dan 2 dengan hasil berikut:

Hasil Algoritma IDA* pada TC1 dan TC2				
Test Case	Heuristik	Nodes Explored	Steps	Times (ms)
Test Case 1	Combine	2048	10	28
	Distance	7179	10	34
	Blocker	10129	10	51
Test Case 2	Combine	15625338	84	29726
	Distance	17648512	84	29575
	Blocker	18653744	84	32041

Pada Test Case 1, IDA* berhasil menemukan solusi optimal sepanjang 10 langkah dengan jumlah eksplorasi node yang relatif rendah. Heuristik Combine menunjukkan performa terbaik dengan hanya 2.048 node dieksplorasi dalam waktu 28 ms. Heuristik Distance dan Blocker juga mencapai solusi yang sama, tetapi memerlukan eksplorasi lebih banyak node, masing-masing 7.179 dan 10.129 node, dengan waktu eksekusi sedikit lebih tinggi (34 dan 51 ms). Hasil ini menunjukkan bahwa IDA* cukup kompetitif dibandingkan A*, bahkan mampu mengimbangi efisiensinya, terutama saat digunakan dengan heuristik gabungan yang mempertimbangkan lebih dari satu aspek solusi.

Namun, pada Test Case 2, performa IDA* menurun drastis dalam hal efisiensi eksplorasi. Meskipun tetap menghasilkan solusi optimal sepanjang 84 langkah, jumlah node yang dieksplorasi melonjak sangat tinggi. Heuristik Combine mengeksplorasi lebih dari 15 juta node (15.625.338) dalam waktu 29.726 ms (hampir 30 detik), sementara heuristik Distance dan Blocker masing-masing mengeksplorasi lebih dari 17 dan 18 juta node, dengan waktu pencarian melebihi 29 detik. Peningkatan eksplorasi ini menunjukkan bahwa meskipun IDA* efisien dalam penggunaan memori karena tidak menyimpan seluruh priority queue seperti A*, ia sangat bergantung pada kualitas heuristik dan jumlah iterasi batas biaya (threshold) yang harus dilakukan ulang dari awal. Pada konfigurasi papan yang kompleks dan padat seperti pada

Test Case 2, pendekatan depth-first iterative deepening menyebabkan overhead eksplorasi yang besar.

Secara keseluruhan, IDA* menunjukkan performa yang solid dan efisien pada konfigurasi ringan seperti Test Case 1, terutama saat dikombinasikan dengan heuristik gabungan. Namun, untuk konfigurasi yang lebih kompleks seperti Test Case 2, IDA* menjadi sangat tidak efisien dari sisi waktu dan jumlah node yang harus dieksplorasi, meskipun solusi yang dihasilkan tetap optimal. Hal ini menegaskan bahwa IDA* cocok digunakan pada problem ruang keadaan terbatas, tetapi kurang ideal untuk kasus dengan kompleksitas tinggi tanpa heuristik yang sangat tajam dan terarah.

Pengujian jika dilakukan dengan konfigurasi papan permainan yang tidak normal dengan dimensi 7 x 9 dengan berikut hasilnya:

Hasil Test Case 3			
Ukuran Papan: Lebih Besar (7 x 9)		Kepadatan: Cukup Padat (23 / 63 Cell Kosong)	
Algoritma	Nodes Explored	Steps	Times (ms)
UCS	1390874	15	14688
GBFS Combine	323	25	26
A* Combine	3058	15	150
IDA* Combine	24898	15	71

Pengujian ketiga dilakukan pada papan permainan berdimensi lebih besar, yaitu 7x9, dengan tingkat kepadatan cukup padat (23 dari 63 sel kosong dan 15 pieces). Konfigurasi ini menghadirkan tantangan baru berupa ruang pencarian yang lebih luas. Pada pengujian ini, keempat algoritma utama, Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), A*, dan Iterative Deepening A* (IDA*), diuji menggunakan heuristik Combine, yang menggabungkan berbagai aspek estimasi untuk memberikan pandangan lebih menyeluruh terhadap solusi.

UCS kembali menunjukkan kelemahannya dalam hal efisiensi eksplorasi, meskipun berhasil menemukan solusi optimal sepanjang 15 langkah. Jumlah node yang dieksplorasi sangat besar, yakni 1.390.874 node, dengan waktu eksekusi mencapai 14.688 ms. Hal ini mengonfirmasi bahwa UCS tidak skala dengan baik terhadap ukuran papan dan kompleksitas yang meningkat karena ia mengeksplorasi seluruh jalur berdasarkan biaya tanpa bantuan informasi arah dari heuristik.

Sebaliknya, GBFS dengan heuristik Combine menunjukkan efisiensi luar biasa, hanya mengeksplorasi 323 node dengan waktu eksekusi 26 ms. Namun, solusi yang dihasilkan tidak optimal, yakni 25 langkah. Ini mengindikasikan bahwa meskipun cepat, GBFS tidak mampu menjamin kualitas solusi karena hanya mempertimbangkan nilai heuristik saat memilih jalur dan mengabaikan biaya aktual.

A* menawarkan keseimbangan antara optimalitas dan efisiensi. Dengan solusi optimal 15 langkah, A* hanya mengeksplorasi 3.058 node, jauh lebih sedikit dibanding UCS, dan menyelesaikan pencarian dalam waktu 150 ms. Ini menunjukkan bahwa A* tetap menjadi pilihan unggul untuk papan berukuran besar dengan heuristik yang baik karena dapat secara efektif menyempurnakan pencarian dengan kombinasi biaya aktual($g(n)$) dan estimasi($h(n)$).

IDA* menunjukkan performa yang cukup kompetitif di konfigurasi ini, dengan 24.898 node dieksplorasi dan waktu eksekusi 71 ms. Meski lebih lambat dari GBFS dan mengeksplorasi lebih banyak node, IDA* tetap menemukan solusi optimal 15 langkah, menunjukkan efektivitas pendekatan *iterative deepening* yang tetap hemat memori. Namun, peningkatan ukuran papan secara signifikan memperbesar jumlah iterasi yang dibutuhkan, yang berdampak langsung pada jumlah node yang harus ditelusuri.

5.2 Analisis Kompleksitas Waktu Program

Berdasarkan hasil pengujian yang telah dilakukan pada tiga konfigurasi Rush Hour, kita dapat mengkaji kompleksitas algoritma dari sisi teoritis dan membandingkannya terhadap hasil empiris. Fokus

utama analisis ini meliputi empat algoritma pencarian: Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), A*, dan Iterative Deepening A* (IDA*), dengan mempertimbangkan performa aktual dalam jumlah simpul dieksplorasi, langkah solusi, dan waktu eksekusi.

Secara teoritis, UCS memiliki kompleksitas waktu terburuk sebesar $O(b^{1+C})$, di mana b adalah branching factor dan C adalah cost solusi optimal. Dalam konteks Rush Hour, di mana semua langkah memiliki cost seragam, UCS berperilaku identik dengan Breadth-First Search. Hal ini tampak dari hasil eksperimen, terutama pada Test Case 3, di mana UCS mengeksplorasi lebih dari 1,3 juta simpul dan membutuhkan lebih dari 14 detik untuk solusi sepanjang 15 langkah. Jumlah node yang besar mengonfirmasi sifat eksplorasi menyeluruh UCS, dan hal ini selaras dengan teori bahwa kompleksitas ruang dan waktu UCS tumbuh eksponensial terhadap kedalaman solusi.

Dari sisi teoritis, kompleksitas waktu GBFS berada pada $O(b^m)$, dengan m adalah kedalaman maksimal pencarian. GBFS mengandalkan heuristik tanpa memperhatikan biaya aktual, yang menyebabkan eksplorasi sering kali menyimpang dari solusi optimal, terutama jika heuristik tidak cukup informatif. Hal ini tercermin dalam eksperimen: pada Test Case 3, meskipun hanya mengeksplorasi 323 simpul dalam 26 ms, sangat efisien, solusi yang dihasilkan jauh dari optimal (25 langkah). Kompleksitas yang lebih ringan dalam waktu eksekusi tetapi lebih berat dalam kualitas solusi menunjukkan bahwa GBFS cocok untuk pendekatan cepat namun bukan optimal.

Sama seperti GBFS, A* memiliki kompleksitas waktu teoretis sebesar $O(b^m)$, namun berbeda karena ia mempertimbangkan baik cost aktual maupun heuristik estimasi. Dalam uji empiris, A* mampu mencapai solusi optimal dengan jumlah langkah minimal di seluruh test case, seperti terlihat pada Test Case 3 yang hanya memerlukan 3.058 simpul dan 150 ms untuk solusi sepanjang 15 langkah. Hal ini menunjukkan bahwa meskipun kompleksitas A* secara teoretis serupa dengan GBFS, kontribusi fungsi biaya membuat eksplorasi A* lebih terarah dan efisien dalam mencari solusi optimal.

IDA* memiliki kompleksitas waktu teoritis yang sulit dinyatakan secara eksak karena sangat tergantung pada efektivitas heuristik. Pada kasus terburuk, ia mereduksi menjadi pencarian mendalam iteratif dengan kompleksitas $O(b^d)$, di mana d adalah kedalaman solusi optimal. Dalam praktiknya, hasil pengujian memperlihatkan kontras yang kuat antara papan sederhana dan kompleks. Pada Test Case 1, IDA* cukup efisien, mengeksplorasi hanya 2.048 simpul dengan waktu 28 ms untuk solusi 10 langkah. Namun, pada Test Case 2, eksplorasi melonjak hingga lebih dari 15 juta simpul dan waktu mencapai lebih dari 29 detik, meskipun solusi tetap optimal. Pada Test Case 3, IDA* mengeksplorasi 24.898 simpul dalam 71 ms, kinerja yang masih efisien. Perbedaan ini mengonfirmasi analisis teoritis bahwa performa IDA* sangat tergantung pada kualitas heuristik: semakin akurat estimasi heuristik dalam memangkas cabang tidak perlu, semakin baik performa IDA*. Tanpa heuristik yang kuat, IDA* harus mengulang pencarian dengan threshold yang lebih tinggi, menyebabkan banyak simpul dikunjungi ulang.

Secara keseluruhan, hasil empiris mendukung model kompleksitas waktu yang sudah tercantum di Bab 1. UCS dan A* cenderung menjelajah ruang secara ekstensif, namun A* lebih efisien karena bantuan heuristik. GBFS sangat cepat tetapi rentan menghasilkan solusi suboptimal. Sementara itu, IDA* menyeimbangkan antara efisiensi memori dan jaminan optimalitas, meski kinerjanya sangat fluktuatif tergantung kompleksitas masalah dan kekuatan heuristik. Oleh karena itu, pemilihan algoritma harus disesuaikan dengan konteks ukuran papan dan kebutuhan optimalitas..

BAB VI
LAMPIRAN

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	✓	

Repository Program:

https://github.com/kin-ark/Tucil3_13523146_13523152

Drive Video Test Case:

 [Video Test Case Stima Tucil 3](#)

Daftar Pustaka

- Maulidevi, N. U. (2025). Route Planning (Bagian 1). Institut Teknologi Bandung. Diakses dari
[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf)
- Munir, R. (2025). Route Planning (Bagian 2). Institut Teknologi Bandung. Diakses dari
[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf)
- GeeksforGeeks. (n.d.). Iterative Deepening A Algorithm (IDA) in Artificial Intelligence*. Diakses dari
<https://www.geeksforgeeks.org/iterative-deepening-a-algorithm-ida-artificial-intelligence/>