

PENYELESAIAN PUZZLE RUSH HOUR MENGGUNAKAN ALGORITMA PATHFINDING

Laporan Tugas Kecil 3

Disusun untuk memenuhi mata kuliah IF2211 Strategi Algoritma



oleh:

Mayla Yaffa Ludmilla 13523050

Nayla Zahira 13523079

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2025**

DAFTAR ISI

DAFTAR ISI	1
BAB I	3
DESKRIPSI TUGAS DAN ALGORITMA	3
1.1. Permainan Rush Hour	3
1.2. Algoritma Uniform Cost Search	4
1.3. Algoritma A*	5
1.4. Algoritma Greedy Best First Search	6
1.5. Algoritma IDS	7
1.6. Analisis algoritma UCS, Greedy Best First Search, dan A*	8
1.6.1. Fungsi pengevaluasi dalam penentuan rute	8
1.6.2. Admissible heuristics	9
1.6.3. Perbedaan Algoritma UCS dan BFS pada penyelesaian Rush Hour	9
1.6.4. Perbandingan efisiensi algoritma A* dengan algoritma UCS pada penyelesaian Rush Hour	9
1.6.5. Penyelesaian Rush Hour dengan Algoritma Greedy Best First Search	10
BAB II	11
IMPLEMENTASI PROGRAM	11
2.1. Struktur Kode Program	11
2.2. Penjelasan Kelas-Kelas	11
2.2.1. Kelas Main	11
2.2.2. Kelas AStar	12
2.2.3. Kelas GBFS	12
2.2.4. Kelas UCS	13
2.2.5. Kelas IDS	14
2.2.6. Kelas Helper	15
2.2.7. Kelas SolveResult	20
2.2.8. Kelas Node	21
2.2.9. Kelas Board	22
2.2.10. Kelas Piece	26
2.2.11. Kelas Input	28
2.2.12. Kelas RushHourApp	30
2.2.13. Kelas Output	37
BAB III	39
EKSPERIMEN DAN ANALISIS	39
3.1. Test Case 1	39
3.2. Test Case 2	41

3.3. Test Case 3	43
3.4. Test Case 4	45
3.5. Test Case 5 (Bonus)	47
3.6. Test Case 6 (Bonus)	48
3.7. Hasil Analisis Percobaan Algoritma Pathfinding	50
BAB IV	52
IMPLEMENTASI BONUS	52
5.1. GUI (Graphical User Interface)	52
5.2. Algoritma Alternatif: Iterative Deepening Search	52
5.3. Heuristik Alternatif	53
LAMPIRAN	55
DAFTAR PUSTAKA	56

BAB I

DESKRIPSI TUGAS DAN ALGORITMA

1.1. Permainan Rush Hour

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

Uniform Cost Search (UCS) adalah algoritma pencarian (*uninformed*) atau tidak terinformasi yang dikembangkan untuk menemukan jalur optimal pada graf berbobot. Berbeda dengan BFS dan DFS yang hanya mempertimbangkan jumlah langkah, UCS dirancang untuk menghasilkan solusi dengan total biaya minimum menggunakan fungsi evaluasi $g(n)$ yang merepresentasikan biaya kumulatif dari simpul awal hingga simpul n . Dengan memanfaatkan priority queue yang selalu memprioritaskan simpul dengan biaya terendah, algoritma ini menjamin ditemukannya jalur terpendek berdasarkan bobot edge.

Implementasi algoritma Uniform Cost Search (UCS) pada program dimulai dengan penggunaan struktur data PriorityQueue untuk menyimpan simpul hidup dengan prioritas berdasarkan nilai $g(n)$ yaitu biaya dari titik awal ke simpul n . Simpul awal dimasukkan ke dalam antrian dengan biaya awal 0, kemudian algoritma melakukan iterasi dengan mengambil simpul dengan biaya terendah menggunakan metode poll(). Untuk mencegah eksplorasi berulang pada state yang sama, implementasi ini menggunakan HashSet visited yang menyimpan representasi string dari setiap board yang telah dikunjungi. Pada setiap iterasi, algoritma memeriksa apakah simpul saat ini merupakan goal state, dan jika belum, maka mengekskansi simpul tersebut dengan menambahkan semua child node yang valid ke dalam priority queue. Ketika goal state ditemukan, algoritma mengembalikan objek SolveResult yang berisi simpul solusi dan jumlah simpul yang telah dikunjungi.

Algoritma UCS ini menjamin menemukan solusi optimal karena selalu memprioritaskan ekspansi simpul dengan biaya terendah terlebih dahulu.

```
function solve(Board start) -> Node
{ Mencari solusi jalur minimum dari papan awal menggunakan algoritma UCS }

Deklarasi
    simpulHidup : PriorityQueue<Node> // diurutkan berdasarkan  $f(n) = g(n) + h(n)$ 
    visited      : Set<String>       // menyimpan representasi dari state yang telah dikunjungi
    current, child : Node              // simpul saat ini dan simpul hasil perluasan
    hash : string                     // representasi unik dari papan

Algoritma:
    visited ← new HashSet<>();
    int h0 ← Helper.blockingDistance(start);
    simpulHidup.add(new Node(start, null, null, 0, h0));
    while (simpulHidup.isEmpty() != true) do
        current ← simpulHidup.poll();
        hash ← current.board.toString();
        if (visited.add(hash) != true) then
            if (Helper.isGoal(current.board)) then
                ->new SolveResult(current, visited.size())
            for (child : Helper.expand(current, false, true))
                simpulHidup.add(child);
    ->null;
```

1.3. Algoritma A*

A* adalah algoritma pencarian jalur yang menghindari perluasan jalur yang mahal sejak awal. Berbeda dengan algoritma pencarian biasa seperti Breadth-First Search atau Uniform Cost Search, A* memanfaatkan informasi estimasi (heuristik) untuk menyempurnakan arah pencarian, sehingga tidak hanya menjelajah berdasarkan biaya aktual, tetapi juga mempertimbangkan seberapa dekat sebuah simpul ke tujuan. A* dilakukan dengan menggabungkan dua nilai:

- $g(n)$ = biaya dari titik awal ke simpul n
- $h(n)$ = estimasi biaya dari n ke tujuan (heuristik)

A* termasuk algoritma yang lengkap, artinya akan selalu menemukan solusi jika memang ada, asalkan tidak terdapat jumlah simpul tak hingga dengan nilai $f(n)$ lebih kecil dari $f(goal)$. Namun, dalam kasus terburuk, kompleksitas waktu dan ruang A* bisa menjadi eksponensial, yaitu $O(b^m)$ dengan b adalah faktor percabangan dan m adalah kedalaman atau panjang solusi.

Implementasi algoritma A* pada program ini menggunakan PriorityQueue yang diatur berdasarkan nilai $f(n) = g(n) + h(n)$. Pertama-tama, simpul awal dimasukkan ke dalam simpulHidup dengan $g = 0$ dan h dihitung oleh fungsi helper. Pada setiap iterasi, algoritma mengambil simpul dengan nilai f terendah dari simpul hidup. Untuk menghindari eksplorasi ulang, setiap board yang sudah diproses dicatat dalam visited (HashSet<String>), sehingga simpul yang mewakili state yang sudah pernah diproses sebelumnya diabaikan. Setelah mengambil simpul, jika simpul memenuhi fungsi Helper.isGoal(), algoritma langsung mengembalikan hasil berupa simpul solusi dan jumlah simpul unik yang sudah dikunjungi. Jika bukan goal, ia memanggil Helper.expand(current, true, true) di mana flag pertama menandakan bahwa child-child diurutkan menurut evaluasi A*, dan flag kedua menyalakan penggunaan heuristik. Kemudian, semua anak hasil expand ditambahkan ke dalam queue. Dengan selalu mengekspansi simpul dengan estimasi total biaya $f(n)$ paling kecil, A* menjamin menemukan jalur solusi yang optimal sekaligus seringkali jauh lebih efisien daripada UCS murni, karena heuristik $h(n)$ mengarahkan pencarian langsung ke area yang dekat dengan goal.

```
function solve(Board start) → Node
{ Mencari solusi jalur minimum dari papan awal menggunakan algoritma A*. }

Deklarasi:
    simpulHidup : PriorityQueue<Node> // diurutkan berdasarkan  $f(n) = g(n) + h(n)$ 
    visited     : Set<String>         // menyimpan representasi string dari state yang
    telah dikunjungi
    current, child : Node              // simpul saat ini dan simpul hasil perluasan
    hash          : String            // representasi unik dari papan

Algoritma:
```

```

visited ← new HashSet<>();
int h0 ← Helper.blockingDistance(start);
simpulHidup.add(Node(start, null, null, 0, h0)); // g = 0, h = h0

while (simpulHidup is not empty) do
    current ← simpulHidup.poll();
    hash ← current.board.toString();
    if (hash ∈ visited) then
        continue;
    visited.add(hash);
    if (Helper.isGoal(current.board)) then
        → current;
    for each child ∈ Helper.expand(current, true, true) do // expand dengan
    memperhitungkan h(n) dan g(n)
        simpulHidup.add(child);
    → null; // tidak ditemukan solusi

```

1.4. Algoritma Greedy Best First Search

Greedy Best-First Search (GBFS) adalah algoritma pencarian heuristik yang bekerja dengan menggunakan fungsi evaluasi $f(n) = h(n)$, di mana $h(n)$ adalah perkiraan nilai heuristik dari node n ke tujuan. Berbeda dengan algoritma pencarian tak terarah seperti BFS atau DFS, GBFS menggunakan informasi heuristik untuk selalu memprioritaskan ekspansi node yang "terlihat" paling dekat dengan tujuan. Algoritma ini memilih node dengan nilai heuristik terkecil dari antrian prioritas untuk dieksplorasi terlebih dahulu. Meskipun GBFS dapat menemukan solusi dengan cepat, ia tidak menjamin solusi optimal karena hanya mempertimbangkan jarak ke tujuan tanpa memperhitungkan biaya yang telah ditempuh dari node awal.

Implementasi GBFS pada program ini menggunakan PriorityQueue dengan komparator berdasarkan nilai heuristik (h) untuk menyimpan simpul-simpul yang akan dieksplorasi. Program dimulai dengan menghitung nilai heuristik awal menggunakan fungsi `Helper.heuristic()` yang menghitung nilai heuristik sesuai dengan jenis heuristik yang dipilih. Pada setiap iterasi, node dengan nilai heuristik terkecil diambil dari antrian prioritas dan diperiksa apakah sudah mencapai tujuan (mobil target dapat keluar dari papan). Jika belum, semua kemungkinan langkah berikutnya (node anak) akan dibangkitkan melalui fungsi `Helper.expand()` dan ditambahkan ke antrian prioritas. Proses ini berlanjut hingga solusi ditemukan atau semua kemungkinan telah dieksplorasi. Untuk efisiensi, program juga menggunakan HashSet untuk melacak posisi papan yang telah dikunjungi, sehingga menghindari pengulangan eksplorasi pada konfigurasi yang sama.

```

function solve(Board start) -> Node
{ Mencari solusi jalur minimum dari papan awal menggunakan algoritma GBFS }

Deklarasi
simpulHidup : PriorityQueue<Node>

```

```

visited : Set<String>
current, child : Node
hash : string

```

Algoritma:

```

visited ← new HashSet<>();
int h0 ← Helper.blockingDistance(start);
simpulHidup.add(new Node(start, null, null, 0, h0));
while (simpulHidup.isEmpty() != true) do
    current ← simpulHidup.poll();
    hash ← current.board.toString();
    if (visited.add(hash) != true) then
        if (Helper.isGoal(current.board)) then
            →current
        for (child : Helper.expand(current, true, false)) {
            simpulHidup.add(child);
        }
    →null;

```

1.5. Algoritma IDS

Strategi algoritma Iterative Deepening Search (IDS) adalah algoritma pencarian yang menggabungkan keunggulan dari Depth-First Search (DFS) dan Breadth-First Search (BFS). IDS bekerja dengan cara menjalankan DFS berulang kali dengan batas kedalaman yang meningkat secara bertahap. Setiap iterasi hanya mengeksplorasi hingga kedalaman tertentu, sehingga simpul yang lebih dekat dengan akar akan ditemukan lebih awal, mirip seperti BFS, namun dengan penggunaan memori seperti DFS.

Kompleksitas waktunya adalah $O(b^d)$ sedangkan kompleksitas ruangnya adalah $O(d)$. Implementasi IDS dalam program ini sebagai berikut.

```

function solve(Board start, int mode) → SolveResult
{ Mencari solusi jalur minimum dari papan awal menggunakan algoritma IDS }

```

Deklarasi

```

depthLimit    : int
visitedCount   : int
visited        : Set<String>
result         : Node

```

Algoritma:

```

depthLimit ← 0
visitedCount ← 0

loop
    visited ← new HashSet<>()
    result ← dls( new Node(start, null, null, 0, 0),
                  depthLimit,
                  Visited, mode)

    if (result ≠ null) then
        → new SolveResult(result, visitedCount + visited.size())

    visitedCount ← visitedCount + visited.size()
    depthLimit ← depthLimit + 1

```



```

    end loop

function dls(Node current, int limit, Set<String> visited, int mode) → Node
{ Depth-Limited Search rekursif hingga kedalaman "limit" }

Deklarasi
    hash    : String
    child    : Node
    found    : Node

Algoritma:
    hash ← current.board.toString()
    if (visited.contains(hash)) then
        → null
    visited.add(hash)

    if (Helper.isGoal(current.board)) then
        → current

    if (limit == 0) then
        → null

    for each child in Helper.expand(current, false, false, mode) do
        found ← dls(child, limit - 1, visited)
        if (found ≠ null) then
            → found
    end for

    → null

```

1.6. Analisis algoritma UCS, Greedy Best First Search, dan A*

1.6.1. Fungsi pengevaluasi dalam penentuan rute

Dalam penentuan rute atau route planning, beberapa komponen penting digunakan untuk mengevaluasi node dalam algoritma pencarian. Fungsi $g(n)$ merepresentasikan biaya sebenarnya yang telah ditempuh dari node awal hingga node n saat ini, mencerminkan jarak atau usaha yang telah dilakukan dalam pencarian. Sementara itu, $h(n)$ adalah fungsi heuristik yang mengestimasi biaya dari node n ke node tujuan, memberikan "pandangan ke depan" tentang prospek suatu jalur. Kombinasi kedua fungsi ini membentuk fungsi evaluasi $f(n)$ yang digunakan oleh berbagai algoritma pencarian dengan keterangan seperti berikut:

- UCS (Uniform Cost Search): $f(n) = g(n)$
- Greedy Best First Search: $f(n) = h(n)$
- A*: $f(n) = g(n) + h(n)$.

Pada program penyelesaian permainan Rush Hour ini, $g(n)$ yang digunakan adalah jumlah langkah yang telah dilakukan dari posisi awal hingga posisi saat ini. Sedangkan $h(n)$ yang digunakan adalah hasil dari fungsi heuristic() yang memiliki 3 mode perhitungan yaitu:

- Distance + Blockers: berdasarkan jarak primary piece ke pintu keluar dijumlahkan dengan banyaknya kendaraan lain yang menghalangi jalur menuju pintu keluar tersebut
- Blockers + 2nd Blockers: berdasarkan jumlah kendaraan yang menghalangi jalur primary piece menuju pintu keluar dijumlahkan dengan banyaknya kendaraan yang menghalangi penghalang dari primary piece.
- Distance only: berdasarkan jarak primary piece ke pintu keluar saja

1.6.2. Admissible heuristics

Suatu heuristik dikatakan admissible (dapat diterima) jika tidak pernah melebihi-lebihkan biaya sebenarnya untuk mencapai tujuan. Secara matematis, $h(n)$ harus selalu kurang dari atau sama dengan $h^*(n)$, dimana $h^*(n)$ adalah biaya sesungguhnya dari node n ke node tujuan. Dengan kata lain, heuristik harus bersifat optimistik dalam mengestimasi biaya. Jika syarat admissible ini terpenuhi, algoritma A* dijamin akan menemukan solusi optimal. Dalam konteks pencarian rute, contoh heuristik yang admissible adalah jarak garis lurus (straight-line distance) ke tujuan, karena jarak tersebut tidak akan pernah lebih besar dari jarak sebenarnya yang harus ditempuh melalui jalan yang tersedia

1.6.3. Perbedaan Algoritma UCS dan BFS pada penyelesaian Rush Hour

Pada penyelesaian Rush Hour dengan implementasi yang diberikan, algoritma UCS tidak akan menghasilkan urutan pembangkitan node dan jalur yang sama dengan BFS. Perbedaan krusial terletak pada cara pergerakan kendaraan diperhitungkan. Dari metode generate(), terlihat bahwa sebuah kendaraan dapat bergerak beberapa langkah secara berurutan dalam satu gerakan dan setiap gerakan dihitung sebagai satu biaya. Hal ini berarti bahwa jika sebuah kendaraan bergerak 3 langkah ke depan dalam satu gerakan, biayanya tetap dihitung sebagai 1, bukan 3. UCS akan memprioritaskan node berdasarkan total biaya gerakan, bukan berdasarkan jumlah langkah fisik yang ditempuh. Sementara itu, BFS akan memprioritaskan node berdasarkan kedalaman level dari node awal tanpa mempertimbangkan biaya.

Karena itu, dalam skenario di mana beberapa gerakan dengan jumlah langkah berbeda memiliki biaya yang sama, UCS dan BFS akan menghasilkan urutan pembangkitan node yang berbeda dan kemungkinan juga menghasilkan jalur solusi yang berbeda. UCS akan memprioritaskan jalur dengan jumlah gerakan paling sedikit, sementara BFS bisa saja memprioritaskan jalur yang mencapai tujuan dengan jumlah langkah fisik yang lebih sedikit tetapi mungkin memerlukan lebih banyak gerakan terpisah.

1.6.4. Perbandingan efisiensi algoritma A* dengan algoritma UCS pada penyelesaian Rush Hour

Perbandingan efisiensi algoritma A* dengan algoritma UCS pada penyelesaian Rush Hour sangat bergantung pada kualitas fungsi heuristik yang digunakan. Pada

penyelesaian Rush Hour karena A* menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, dimana $g(n)$ merupakan biaya aktual ke simpul saat ini dan $h(n)$ merupakan fungsi heuristik yang digunakan. Pada kasus default fungsi heuristiknya menghitung jarak menuju simpul tujuan ditambah dengan jumlah penghalang dalam jalur menuju simpul tujuan.

Dengan heuristik yang admissible (tidak overestimasi), A* akan mengeksplorasi simpul lebih selektif dibanding UCS dan mengurangi ruang pencarian dengan memprioritaskan eksplorasi node yang lebih menjanjikan. Pada permainan Rush Hour yang memiliki branching factor cukup tinggi, penggunaan heuristik membantu algoritma lebih cepat menemukan jalur yang mengarah ke solusi. Hal ini jauh lebih efisien dibandingkan dengan UCS yang melakukan pencarian secara "buta" berdasarkan biaya aktual saja tanpa mempertimbangkan seberapa dekat suatu state dengan goal. Dengan demikian, A* umumnya memiliki kompleksitas waktu yang lebih efisien, walaupun kompleksitas ruangnya sama yaitu $O(b^d)$, dimana b adalah branching factor dan d adalah kedalaman.

1.6.5. Penyelesaian Rush Hour dengan Algoritma Greedy Best First Search

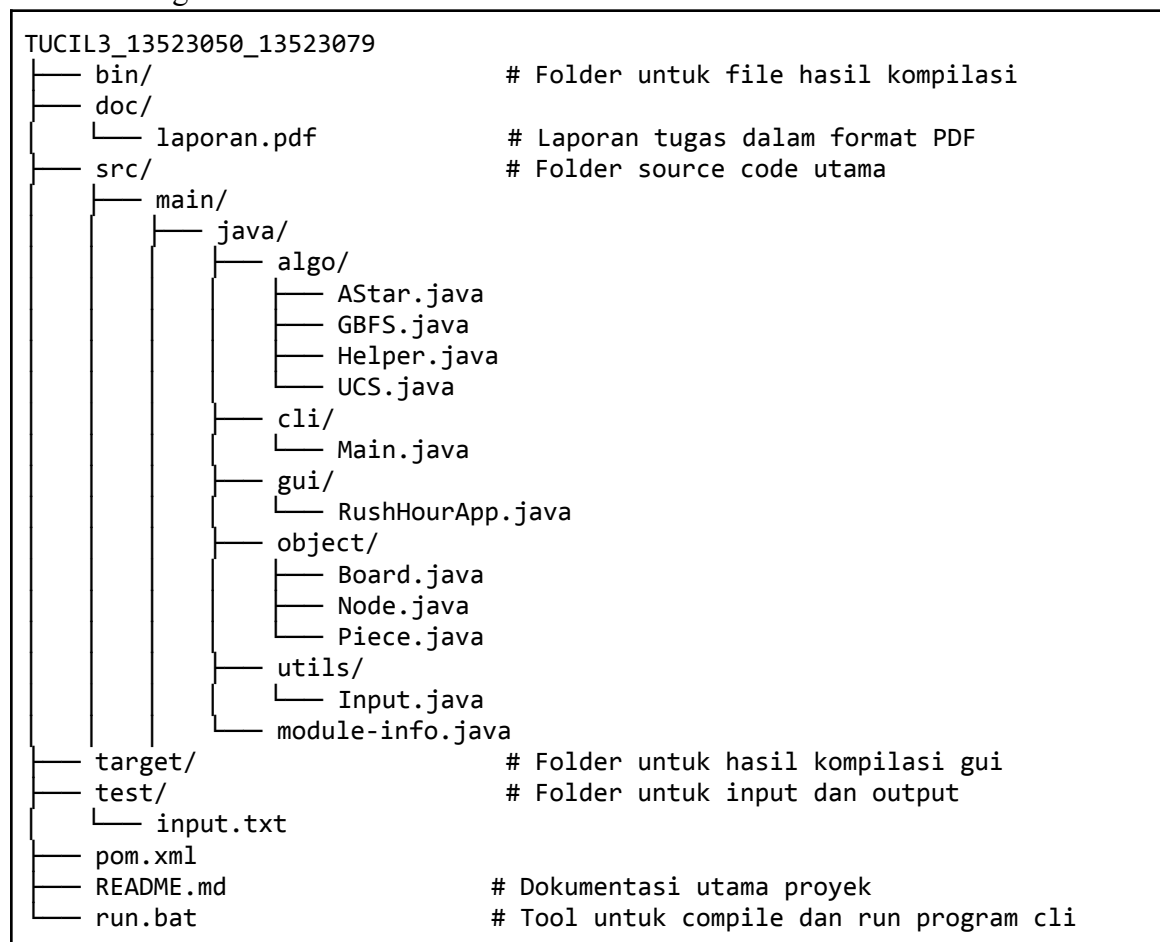
Algoritma Greedy Best First Search tidak menjamin solusi optimal untuk penyelesaian Rush Hour. Hal ini karena algoritma GBFS menggunakan priority queue yang hanya memprioritaskan node berdasarkan nilai heuristik $h(n)$ saja, tanpa mempertimbangkan biaya yang ditempuh ($g(n)$). Pemilihan berdasarkan nilai heuristik saja akan mengakibatkan pemilihan langkah berikutnya yang tampak menjanjikan tanpa mempertimbangkan jalur keseluruhan. Akibatnya, algoritma ini dapat memilih langkah-langkah yang tampaknya mendekat ke tujuan dalam jangka pendek, tetapi sebenarnya membutuhkan lebih banyak langkah secara keseluruhan dibandingkan jalur alternatif.

BAB II

IMPLEMENTASI PROGRAM

2.1. Struktur Kode Program

Pada program ini, sebagian kode yang krusial dibuat dengan paradigma berorientasi objek (OOP) untuk mempermudah pemrosesan metode atau fungsi yang merupakan tanggung jawab masing-masing kelas. Namun, terdapat beberapa kelas yang tidak menggunakan paradigma ini karena proses lebih bersifat prosedural. Folder program disusun sebagai berikut:



2.2. Penjelasan Kelas-Kelas

2.2.1. Kelas Main

Kelas ini berisi program utama cli yang akan dijalankan untuk menjalankan program di command line.

2.2.2. Kelas AStar

Kelas ini berisi program penyelesaian permainan rush hour menggunakan pendekatan algoritma A*:

Kelas A*	
Metode	Penjelasan
<pre>public static SolveResult solve(Board start, int mode)</pre>	Metode yang dipanggil di kelas Main untuk menghasilkan node solusi dari puzzle rush hour
<pre>package algo; import java.util.Comparator; import java.util.HashSet; import java.util.PriorityQueue; import java.util.Set; import object.Board; import object.Node; public class AStar { public static SolveResult solve(Board start, int mode) { PriorityQueue<Node> simpulHidup = new PriorityQueue<>(Comparator.comparingInt(node -> node.f())); Set<String> visited = new HashSet<>(); int h0 = Helper.heuristic(start, mode); simpulHidup.add(new Node(start, null, null, 0, h0)); while (!simpulHidup.isEmpty()) { Node current = simpulHidup.poll(); String hash = current.board.toString(); if (visited.contains(hash)) { continue; } visited.add(hash); if (Helper.isGoal(current.board)) { return new SolveResult(current, visited.size()); } for (Node child : Helper.expand(current, true, true, mode)) { simpulHidup.add(child); } } return new SolveResult(null, visited.size()); } }</pre>	

2.2.3. Kelas GBFS

Kelas ini berisi program penyelesaian permainan rush hour menggunakan pendekatan algoritma GBFS:

Kelas GBFS	
Metode	Penjelasan
<pre>public static SolveResult solve(Board start, int mode)</pre>	<p>Metode yang dipanggil di kelas Main untuk menghasikan node solusi dari puzzle rush hour</p>
<pre>package algo; import java.util.Comparator; import java.util.HashSet; import java.util.PriorityQueue; import java.util.Set; import object.Board; import object.Node; public class GBFS { public static SolveResult solve(Board start, int mode) { PriorityQueue<Node> simpulHidup = new PriorityQueue<>(Comparator.comparingInt(node -> node.h)); Set<String> visited = new HashSet<>(); int h0 = Helper.heuristic(start, mode); simpulHidup.add(new Node(start, null, null, 0, h0)); while (!simpulHidup.isEmpty()) { Node current = simpulHidup.poll(); String hash = current.board.toString(); if (visited.contains(hash)) { continue; } visited.add(hash); if (Helper.isGoal(current.board)) { return new SolveResult(current, visited.size()); } for (Node child : Helper.expand(current, true, false, mode)){ simpulHidup.add(child); } } return new SolveResult(null, visited.size()); } }</pre>	

2.2.4. Kelas UCS

Kelas ini berisi program penyelesaian permainan rush hour menggunakan pendekatan algoritma UCS.

Kelas UCS

Metode	Penjelasan
<pre>public static SolveResult solve(Board start, int mode)</pre>	Metode yang dipanggil di kelas Main untuk menghasilkan node solusi dari puzzle rush hour
<pre>package algo; import java.util.Comparator; import java.util.HashSet; import java.util.PriorityQueue; import java.util.Set; import object.Board; import object.Node; public class UCS { public static SolveResult solve(Board start, int mode) { PriorityQueue<Node> simpulHidup = new PriorityQueue<>(Comparator.comparingInt(node -> node.g)); Set<String> visited = new HashSet<>(); simpulHidup.add(new Node(start, null, null, 0, 0)); while (!simpulHidup.isEmpty()) { Node current = simpulHidup.poll(); String hash = current.board.toString(); if (!visited.add(hash)) continue; if (Helper.isGoal(current.board)) { return new SolveResult(current, visited.size()); } for (Node child : Helper.expand(current, false, true, mode)){ simpulHidup.add(child); } } return new SolveResult(null, visited.size()); } }</pre>	

2.2.5. Kelas IDS

Kelas ini berisi program penyelesaian permainan rush hour menggunakan pendekatan algoritma UCS.

Kelas IDS	
Metode	Penjelasan
<pre>public static SolveResult solve(Board start, int mode)</pre>	Metode yang dipanggil di kelas Main untuk menghasilkan node solusi dari puzzle rush hour
<pre>private static Node dls(Node current, int limit, Set<String> visited, int mode)</pre>	Metode untuk melakukan depth

	limited search
<pre> package algo; import java.util.HashSet; import java.util.Set; import object.Board; import object.Node; public class IDS { public static SolveResult solve(Board start, int mode) { int depthLimit = 0; int visitedCount = 0; while (true) { Set<String> visited = new HashSet<>(); Node result = dls(new Node(start, null, null, 0, 0), depthLimit, visited, mode); if (result != null) return new SolveResult(result, visitedCount); depthLimit++; visitedCount += visited.size(); } } private static Node dls(Node current, int limit, Set<String> visited, int mode) { String hash = current.board.toString(); if (visited.contains(hash)) return null; visited.add(hash); if (Helper.isGoal(current.board)) return current; if (limit == 0) return null; for (Node child : Helper.expand(current, false, false, mode)) { Node result = dls(child, limit - 1, visited, mode); if (result != null) return result; } return null; } } </pre>	

2.2.6. Kelas Helper

Kelas ini berisi fungsi helper dalam implementasi fungsi solver pada tiap algoritma

Kelas Helper	
Metode	Penjelasan
public static int heuristic(Board b, int mode)	Metode wrapper yang memanggil fungsi heuristik sesuai pilihan pengguna
public static int blockingDistance(Board	Metode yang menghitung h(n)

b)	suatu board dengan heuristiknya berupa jumlah petak p ke pintu keluar ditambah banyaknya mobil yang menghalangi
<code>public static boolean isGoal(Board b)</code>	Metode yang mengembalikan true jika primary piece sudah di depan pintu keluar
<code>public static List<Node> expand(Node n, boolean useHeuristic, boolean useG)</code>	Metode untuk menghasilkan semua node anak yang mungkin dari sebuah node dengan mencoba menggerakkan setiap piece ke semua arah yang mungkin (horizontal atau vertikal).
<code>private static void generate(Piece p, String dir, Node current, Board board, List<Node> out, boolean heuristic, boolean useG)</code>	Metode yang menghasilkan semua kemungkinan node baru dari menggerakkan satu piece ke satu arah tertentu, dengan meneruskan gerakan piece selama masih memungkinkan dan menghitung nilai heuristik dan biaya path jika diperlukan.
<code>private static int countDirectBlockers(Board b, Piece p)</code>	Metode yang menghitung jumlah penghalang dari suatu primary piece
<code>private static int distanceOnly(Board b)</code>	Metode yang menghitung h(n) suatu board dengan heuristiknya berupa jumlah petak p ke pintu keluar
<code>private static List<Piece> directBlockerPieces(Board b, Piece p)</code>	Metode yang mengembalikan list dari pieces yang menghalangi primary piece
<code>private static Piece findPieceById(Board b, char id)</code>	Metode yang mengembalikan piece berdasarkan id nya.
<code>private static int blockingPlusPlus(Board b)</code>	Metode yang menghitung h(n) suatu board dengan heuristiknya berupa banyaknya mobil penghalang dari primary piece menuju pintu keluar dijumlahkan

dengan banyaknya mobil lain
yang menghalangi penghalang

```
package algo;

import java.util.ArrayList;
import java.util.List;
import object.Board;
import object.Node;
import object.Piece;;

public final class Helper {

    public static int heuristic(Board b, int mode){
        switch(mode){
            case 0:
                return blockingDistance(b);
            case 1:
                return blockingPlusPlus(b);
            case 2:
                return distanceOnly(b);
            default:
                return blockingDistance(b);
        }
    }

    private static int blockingDistance(Board b) {
        Piece p = b.getPrimaryPiece();
        if (p == null) return 0;

        int dist = distanceOnly(b);
        int blockers = countDirectBlockers(b, p);
        return dist + blockers;
    }

    public static boolean isGoal(Board b) {
        Piece p = b.getPrimaryPiece();
        if (p == null) return false;
        switch (b.exitDir) {
            case 'R':
                return p.isHorizontal && p.row == b.goalRow &&
                    p.col + p.length == b.goalCol;
            case 'L':
                return p.isHorizontal && p.row == b.goalRow &&
                    p.col == b.goalCol + 1;
            case 'D':
                return !p.isHorizontal && p.col == b.goalCol &&
                    p.row + p.length == b.goalRow;
            case 'U':
                return !p.isHorizontal && p.col == b.goalCol &&
                    p.row == b.goalRow + 1;
            default:
                return false;
        }
    }
}
```

```

    }

    public static List<Node> expand(Node n, boolean useHeuristic, boolean
useG, int mode) {
        List<Node> childs = new ArrayList<>();
        Board b = n.board;

        for (Piece p : b.pieces) {
            if (p.isHorizontal) {
                generate(p, "L", n, b, childs, useHeuristic, useG, mode);
                generate(p, "R", n, b, childs, useHeuristic, useG, mode);
            } else {
                generate(p, "U", n, b, childs, useHeuristic, useG, mode);
                generate(p, "D", n, b, childs, useHeuristic, useG, mode);
            }
        }
        return childs;
    }

    private static void generate(Piece p, String dir, Node current, Board
board, List<Node> out, boolean heuristic, boolean useG, int mode) {
        Board next = board;
        int step = 0;
        Piece currentPiece = p;
        while (next.canMove(currentPiece, dir)) {
            step++;
            next = next.applyMove(currentPiece, dir);
            int h = heuristic ? Helper.heuristic(next, mode) : 0;
            String mv = p.id + " " + dir + " " + step;
            if (useG) out.add(new Node(next, current, mv, current.g + 1,
h));
            else out.add(new Node(next, current, mv, 0, h));
            currentPiece = next.pieces.stream().filter(pc -> pc.id ==
p.id).findFirst().get();
        }
    }

    private static int countDirectBlockers(Board b, Piece p) {
        return directBlockerPieces(b, p).size();
    }

    private static int distanceOnly(Board b) {
        Piece p = b.getPrimaryPiece();
        if (p == null) return 0;

        if (p.isHorizontal) {
            return Math.abs(b.getGoalCol() - (b.exitDir == 'L' ? p.col :
p.col + p.length - 1));
        } else {
            return Math.abs(b.getGoalRow() - (b.exitDir == 'U' ? p.row :
p.row + p.length - 1));
        }
    }
}

```

```

private static List<Piece> directBlockerPieces(Board b, Piece p) {
    List<Piece> list = new ArrayList<>();
    if (p.isHorizontal()) {
        int r = p.row;
        if (b.exitDir == 'L') {
            for (int c = p.col - 1; c >= 0; c--) {
                char id = b.grid[r][c];
                if (id != '.') {
                    list.add(findPieceById(b, id));
                }
            }
        } else {
            for (int c = p.col + p.length; c < b.width; c++) {
                char id = b.grid[r][c];
                if (id != '.') {
                    list.add(findPieceById(b, id));
                }
            }
        }
    } else {
        int c = p.col;
        if (b.exitDir == 'U') {
            for (int r = p.row - 1; r >= 0; r--) {
                char id = b.grid[r][c];
                if (id != '.') {
                    list.add(findPieceById(b, id));
                }
            }
        } else {
            for (int r = p.row + p.length; r < b.height; r++) {
                char id = b.grid[r][c];
                if (id != '.') {
                    list.add(findPieceById(b, id));
                }
            }
        }
    }
    return list;
}

private static Piece findPieceById(Board b, char id) {
    for (Piece pc : b.pieces) {
        if (pc.id == id) return pc;
    }
    return null;
}

private static int blockingPlusPlus(Board b) {
    Piece p = b.getPrimaryPiece();
    if (p == null) return 0;

    int dist = distanceOnly(b);
    int blockers = countDirectBlockers(b, p);
    int secBlocks = 0;

```

```

        for (Piece q : directBlockerPieces(b, p)) {
            if (q.isHorizontal) {
                boolean leftFree = q.col > 0 && b.grid[q.row][q.col-1] ==
                '.';
                boolean rightFree = q.col + q.length < b.width &&
                b.grid[q.row][q.col+q.length] == '.';
                if (!leftFree && !rightFree) secBlocks++;
            } else {
                boolean upFree = q.row > 0 && b.grid[q.row-1][q.col] ==
                '.';
                boolean downFree = q.row + q.length < b.height &&
                b.grid[q.row+q.length][q.col] == '.';
                if (!upFree && !downFree) secBlocks++;
            }
        }
        return dist + blockers + secBlocks;
    }
}

```

2.2.7. Kelas SolveResult

Kelas ini merupakan kelas wrapper yang digunakan untuk menyimpan hasil dari penyelesaian puzzle, jumlah simpul yang dikunjungi, dan waktu yang digunakan dalam pencarian. Berikut adalah daftar atribut dan metode pada kelas ini:

Kelas SolveResult	
Atribut: public final Node solution; public final int nodesVisited; public final long timeMs;	
Metode	Penjelasan
public SolveResult(Node solution, int nodesVisited) public SolveResult(Node solution, int nodesVisited, long timeMs)	Konstruktor kelas SolveResult.
<pre> package algo; import object.Node; public class SolveResult { public final Node solution; public final int nodesVisited; public final long timeMs; public SolveResult(Node solution, int nodesVisited) { this.solution = solution; } } </pre>	

```

        this.nodesVisited = nodesVisited;
        this.timeMs       = 0;
    }

    public SolveResult(Node solution, int nodesVisited, long timeMs) {
        this.solution      = solution;
        this.nodesVisited  = nodesVisited;
        this.timeMs        = timeMs;
    }
}

```

2.2.8. Kelas Node

Kelas ini merupakan implementasi simpul pada algoritma pathfinding. Berikut adalah daftar atribut dan metode pada kelas ini:

Kelas Node	
Atribut: <pre> public Board board; public Node parent; public String move; public int g; public int h; </pre>	
Metode	Penjelasan
<pre>public Node(Board board, Node parent, String move, int g, int h);</pre>	Konstruktor kelas Node.
<pre>public int f();</pre>	Mengembalikan nilai $f(n)$ yaitu $g(n) + h(n)$
<pre>public int compareTo(Node other)</pre>	Method ini membandingkan nilai fungsi $f()$ dari node saat ini dengan node lainnya untuk menentukan urutan prioritas dalam struktur data seperti PriorityQueue, dengan nilai yang lebih kecil mendapat prioritas lebih tinggi.
<pre>public int getPath();</pre>	Mengembalikan list node dari parent ke current node.
<pre> package object; import java.util.LinkedList; </pre>	

```

import java.util.List;

public class Node implements Comparable<Node> {
    public Board board;
    public Node parent;
    public String move;
    public int g;           // g(n)
    public int h;           // h(n)

    public Node(Board board, Node parent, String move, int g, int h) {
        this.board = board;
        this.parent = parent;
        this.move = move;
        this.g = g;
        this.h = h;
    }

    public int f() {
        return g + h;
    }

    @Override
    public int compareTo(Node other) {
        return Integer.compare(this.f(), other.f());
    }

    public List<Node> getPath(){
        LinkedList<Node> list = new LinkedList<>();
        for(Node n = this; n != null; n = n.parent) list.addFirst(n);
        return list;
    }
}

```

2.2.9. Kelas Board

Kelas ini mengimplementasikan ‘papan’ tempat permainan berlangsung. Berikut adalah daftar atribut dan metode pada kelas ini:

Kelas Board	
Atribut: <pre> public char[][] grid; public List<Piece> pieces; public int width, height; public int goalRow, goalCol; public char exitDir; </pre>	
Metode	Penjelasan

<code>public Board(char[][] grid, List<Piece> pieces, int goalRow, int goalCol, char exitDir)</code>	Konstruktor kelas Board.
<code>public int getGoalRow() public int getGoalCol()</code>	Getter untuk beberapa atribut Board
<code>public void print(boolean isFinished, char movedId, boolean[][] oldMask)</code>	Metode untuk menampilkan state board ke CLI
<code>public String toString()</code>	Metode override untuk mengubah object Board menjadi String
<code>public boolean equals</code>	Metode untuk mengecek apakah 2 board sama
<code>public Board applyMove(Piece p, String dir)</code>	Metode untuk memindahkan sebuah piece sesuai dengan arah yang ditentukan
<code>public boolean canMove(Piece p, String dir)</code>	Metode untuk menentukan apakah suatu piece dapat bergerak ke arah tertentu
<code>public Piece getPrimaryPiece()</code>	Metode untuk mendapat primary piece yang ada di board
<pre> package object; import java.util.ArrayList; import java.util.Arrays; import java.util.List; public class Board { public char[][] grid; public List<Piece> pieces; public int width, height; public int goalRow, goalCol; public char exitDir; public static final char EMPTY = '.'; public static final String RESET = "\u001B[0m"; public static final String RED = "\u001B[31m"; public static final String GREEN = "\u001B[32m"; public static final String BG_YELLOW = "\u001B[43m"; public static final String BLUE = "\u001B[34m"; public Board(char[][] grid, List<Piece> pieces, int goalRow, int </pre>	


```

goalCol, char exitDir) {
    this.height = grid.length;
    this.width = grid[0].length;
    this.grid = grid;
    this.pieces = pieces;
    this.goalRow = goalRow;
    this.goalCol = goalCol;
    this.exitDir = exitDir;
    this.exitDir = exitDir;
}

    public void print(boolean isFinished, char movedId, boolean[][]
oldMask) {
        System.out.println("Board (" + height + " x " + width + "), Goal
at (" + goalRow + ", " + goalCol + ")");
        System.out.println("Grid:");
        if (exitDir == 'U') {
            for (int i = 0; i < width; i++) {
                if (i == goalCol) System.err.print(GREEN + 'K' + RESET);
                else System.out.print(' ');
            }
            System.out.println();
        }
        for (int i = 0; i < height; i++) {
            if (exitDir == 'L'){
                if (i == goalRow) System.err.print(GREEN + 'K' + RESET);
            } else {
                for (int j = 0; j < width; j++) {
                    char ch = grid[i][j];
                    boolean highlightedNow = (ch == movedId);
                    boolean highlightedOld = (oldMask != null &&
oldMask[i][j]);

                    if (highlightedNow) {
                        System.out.print(BG_YELLOW + BLUE + ch + RESET);
                    } else if (highlightedOld) {
                        System.out.print(BG_YELLOW + ch + RESET);
                    } else {
                        boolean primary = false;
                        for (Piece p : pieces)
                            if (p.id == ch && p.isPrimary) { primary =
true; break; }
                        System.out.print(primary ? RED + ch + RESET :
ch);
                    }
                }
            }
            if (exitDir == 'R'){
                if (i == goalRow) System.err.print(GREEN + 'K' + RESET);
            }
            System.out.println();
        }
        if (exitDir == 'D') {
            for (int i = 0; i < width; i++) {

```

```

        if (i == goalCol) System.err.print(GREEN + 'K' + RESET);
        else System.out.print(' ');
    }
    System.out.println();
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder(height * (width + 1));
    for (int r = 0; r < height; r++) {
        for (int c = 0; c < width; c++) {
            char ch = grid[r][c];
            sb.append(ch == EMPTY ? EMPTY : ch);
        }
        sb.append('/'); // pemisah baris
    }
    sb.append(exitDir); // agar beda sisi exit beda
    hash
    return sb.toString();
}

@Override public boolean equals(Object o){
    return (o instanceof Board b) &&
    this.toString().equals(b.toString());
}

public Board applyMove(Piece p, String dir) {
    char[][] newGrid = new char[height][width];
    for (int i = 0; i < height; i++) {
        newGrid[i] = Arrays.copyOf(grid[i], width);
    }

    List<Piece> newPieces = new ArrayList<>();
    for (Piece piece : pieces) {
        if (piece.id == p.id) {
            Piece moved = piece.move(dir);
            newPieces.add(moved);

            for (int i = 0; i < piece.length; i++) {
                if (piece.isHorizontal) {
                    newGrid[piece.row][piece.col + i] = EMPTY;
                } else {
                    newGrid[piece.row + i][piece.col] = EMPTY;
                }
            }

            for (int i = 0; i < moved.length; i++) {
                if (moved.isHorizontal) {
                    newGrid[moved.row][moved.col + i] = moved.id;
                } else {
                    newGrid[moved.row + i][moved.col] = moved.id;
                }
            }
        }
    }
}

```

```

        } else {
            newPieces.add(new Piece(piece.id, piece.row, piece.col,
piece.length, piece.isHorizontal, piece.isPrimary));
        }
    }

    return new Board(newGrid, newPieces, goalRow, goalCol, exitDir);
}

public boolean canMove(Piece p, String dir) {
    if (p.isHorizontal) {
        if (dir.equals("L")) {
            if (p.col - 1 >= 0 && grid[p.row][p.col - 1] == EMPTY)
return true;
        } else if (dir.equals("R")) {
            if (p.col + p.length < width && grid[p.row][ p.col +
p.length] == EMPTY) return true;
        }
    } else {
        if (dir.equals("U")) {
            if (p.row - 1 >= 0 && grid[p.row - 1][p.col] == EMPTY)
return true;
        } else if (dir.equals("D")) {
            if (p.row + p.length < height && grid[p.row +
p.length][p.col] == EMPTY) return true;
        }
    }
    return false;
}

public Piece getPrimaryPiece() {
    for (Piece p : pieces) {
        if (p.isPrimary) {
            return p;
        }
    }
    return null;
}

public int getGoalRow() {
    return goalRow;
}

public int getGoalCol() {
    return goalCol;
}
}

```

2.2.10. Kelas Piece

Kelas Piece berisi implementasi untuk piece atau dalam kasus Rush Hour, mobil, yang berada di atas papan.

Kelas Piece	
Atribut: <pre> public char id; public int row; public int col; public int length; public boolean isHorizontal; public boolean isPrimary; </pre>	
Metode	Penjelasan
<pre> public Piece(char id, int row, int col, boolean isHorizontal, boolean isPrimary) public Piece(char id, int row, int col, int length, boolean isHorizontal, boolean isPrimary) </pre>	Konstruktor kelas Piece.
<pre>public char getId()</pre>	Getter untuk atribut Piece.
<pre>public void addLength()</pre>	Method untuk menambah atribut panjang Piece sebanyak 1, digunakan saat input
<pre>public Piece move(String direction)</pre>	Method untuk memindahkan Piece
<pre>public void print()</pre>	Method untuk menampilkan Piece ke CLI
<pre> package object; public class Piece { public char id; public int row; public int col; public int length; public boolean isHorizontal; public boolean isPrimary; public static final String RESET = "\u001B[0m"; public static final String RED = "\u001B[31m"; public Piece(char id, int row, int col, boolean isHorizontal, boolean isPrimary) { this.id = id; this.row = row; this.col = col; this.length = 1; this.isHorizontal = isHorizontal; this.isPrimary = isPrimary; } </pre>	

```

    public Piece(char id, int row, int col, int length, boolean
isHorizontal, boolean isPrimary) {
        this.id = id;
        this.row = row;
        this.col = col;
        this.length = length;
        this.isHorizontal = isHorizontal;
        this.isPrimary = isPrimary;
    }

    public char getId(){
        return this.id;
    }

    public void addLength(){
        this.length += 1;
    }

    public Piece move(String direction) { // return piece karena untuk
setiap state akan dibuat board baru
        int newRow = this.row;
        int newCol = this.col;
        if (isHorizontal) {
            if (direction.equals("L")) newCol--;
            else if (direction.equals("R")) newCol++;
        } else {
            if (direction.equals("U")) newRow--;
            else if (direction.equals("D")) newRow++;
        }
        return new Piece(id, newRow, newCol, length, isHorizontal,
isPrimary);
    }

    public void print() {
        String info = "Piece " + id +
            " at (" + row + "," + col + "), len=" + length +
            ", " + (isHorizontal ? "Horizontal" : "Vertical") +
            (isPrimary ? ", Primary" : "");
        if (isPrimary) {
            System.out.println(RED + info + RESET);
        } else {
            System.out.println(info);
        }
    }
}

```

2.2.11. Kelas Input

Kelas ini merupakan kelas utility yang menangani masukan berupa file .txt dari pengguna.

Kelas Input	
Metode	Penjelasan
<pre> public static Board readBoardFromFile(String filename) </pre>	<p>Method untuk membaca konfigurasi papan awal dari file .txt</p>
<pre> package utils; import java.io.*; import java.util.*; import java.util.function.BiConsumer; import object.Board; import object.Piece; public class Input { public static Board readBoardFromFile(String filename) throws IOException { BufferedReader br = new BufferedReader(new FileReader(filename)); String[] dim = br.readLine().trim().split("\\s+"); int row = Integer.parseInt(dim[0]); int col = Integer.parseInt(dim[1]); int nPieces = Integer.parseInt(br.readLine().trim()); char[][] grid = new char[row][col]; List<Piece> pieces = new ArrayList<>(); Map<Character, Piece> pieceMap = new HashMap<>(); int goalRow = -1, goalCol = -1; char exitDir = '?'; for (int i = 0; i < row; i++) { String line = br.readLine(); if (line.length() == col + 1 && line.charAt(0) == 'K') { exitDir = 'L'; goalRow = i; goalCol = -1; line = line.substring(1); // drop the K } else if (line.length() == col + 1 && line.charAt(col) == 'K') { exitDir = 'R'; goalRow = i; goalCol = col; line = line.substring(0, col); // keep only the board cells } for (int j = 0; j < col; j++) { grid[i][j] = line.charAt(j); } } } } </pre>	

```

        if (grid[i][j] != '.' && grid[i][j] != 'K'){
            Piece p = pieceMap.get(grid[i][j]);
            if (p == null) {
                boolean horizontal =
                    (j + 1 < col && line.charAt(j+1) ==
grid[i][j]) ||
                    (j - 1 >= 0 && line.charAt(j-1) ==
grid[i][j]);
                boolean isPrimary = (grid[i][j] == 'P');
                p = new Piece(grid[i][j], i, j, 1, horizontal,
isPrimary);
                pieceMap.put(grid[i][j], p);
            } else {
                p.addLength();
            }
        }
    }
    br.mark(2);
    String extra = br.readLine();
    if (extra != null && extra.indexOf('K') != -1) {
        int kPos = extra.indexOf('K');
        if (exitDir != '?')
            throw new IOException("Two exit gates found!");
        if (kPos >= 0 && kPos < col) {
            goalCol = kPos;
            if (extra.trim().equals(extra)) {
                exitDir = 'D'; goalRow = row;
            } else {
                exitDir = 'U'; goalRow = -1;
            }
        } else
            throw new IOException("Malformed K-line: "+extra);
    } else {
        br.reset();
    }
    br.close();

    pieces.addAll(pieceMap.values());
    return new Board(grid, pieces, goalRow, goalCol, exitDir);
}

```

2.2.12. Kelas RushHourApp

Kelas Input
Atribut: <pre>private static final int CELL = 60;</pre>

<pre> private Canvas boardCanvas; private Board currentBoard; private ComboBox<String> algoChoice; private ComboBox<String> heuristicChoice; private ListView<String> moveList; private Label visitedLabel; private Label timeLabel; private Label stepLabel; private SolveResult lastResult; private Button exportButton; </pre>	
Metode	Penjelasan
private void start(Stage stage)	Metode utama untuk inialisasi antarmuka pengguna aplikasi Rush Hour, meliputi komponen seperti tombol, canvas, dan layout.
private void loadBoard(Stage stage)	Metode untuk memuat papan permainan dari file teks yang dipilih pengguna melalui dialog pemilihan file.
private void runSolver()	Metode yang menjalankan algoritma pemecahan masalah Rush Hour berdasarkan pilihan algoritma dari pengguna.
private void drawBoard(Board b) private void drawBoard(Board b, boolean showPrimary)	Metode untuk menggambar representasi visual papan Rush Hour beserta piecenya pada canvas.
private void animate(List<Board> path)	Metode untuk menampilkan animasi langkah-langkah solusi dengan menggambar setiap papan dari jalur solusi secara berurutan.
private void exportToFile(Stage stage)	Metode untuk menyimpan file output.txt
private void showErr(String msg)	Metode utilitas untuk menampilkan pesan kesalahan kepada pengguna dalam bentuk dialog.
private void main(String[] args)	Metode entry point aplikasi JavaFX yang memanggil metode launch untuk memulai aplikasi.
<pre> package gui; import javafx.application.Application; import javafx.application.Platform; import javafx.concurrent.Task; import javafx.geometry.Insets; </pre>	


```

import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

import java.util.List;
import object.Board;
import object.Piece;
import algo.AStar;
import algo.GBFS;
import algo.Helper;
import algo.UCS;
import algo.IDS;
import algo.SolveResult;
import utils.Input;

import java.io.File;
import java.io.IOException;
import utils.Output;

public class RushHourApp extends Application {

    private static final int CELL = 60;
    private Canvas boardCanvas;
    private Board currentBoard;
    private ComboBox<String> algoChoice;
    private ComboBox<String> heuristicChoice;
    private ListView<String> moveList;
    private Label visitedLabel;
    private Label timeLabel;
    private Label stepLabel;
    private SolveResult lastResult;
    private Button exportButton;

    @Override
    public void start(Stage stage) {
        /* ----- Controls panel ----- */
        Button loadBtn = new Button("Load Board");
        Button solveBtn = new Button("Solve");
        algoChoice = new ComboBox<>();
        algoChoice.getItems().addAll("GBFS (Greedy)", "A*", "UCS",
"IDS");
        algoChoice.getSelectionModel().selectFirst();
        heuristicChoice = new ComboBox<>();
        heuristicChoice.getItems().addAll("Distance + Blockers",
"Distance Only", "Blockers + Second Blockers");
        heuristicChoice.getSelectionModel().select(0);

        exportButton = new Button("Export");
    }

```

```

        exportButton.setDisable(true);
        exportButton.setOnAction(e -> exportToFile(stage));
        HBox controlBar = new HBox(10, loadBtn, algoChoice,
heuristicChoice, solveBtn, exportButton);
        controlBar.setAlignment(Pos.TOP_CENTER);
        controlBar.setPrefWidth(140);

        visitedLabel = new Label("Visited: 0");
        timeLabel    = new Label("Time: 0 ms");
        stepLabel    = new Label("Step: 0");
        HBox statusBar = new HBox(20, visitedLabel, timeLabel,
stepLabel);
        statusBar.setAlignment(Pos.CENTER_LEFT);
        statusBar.setPadding(new Insets(5));

        /* ----- Drawing surface ----- */
        boardCanvas = new Canvas();
        StackPane boardPane = new StackPane(boardCanvas);
        boardPane.setMinSize(Region.USE_PREF_SIZE, Region.USE_PREF_SIZE);

        /* ----- Move list ----- */
        moveList = new ListView<>();

        BorderPane root = new BorderPane();
        root.setTop(controlBar);
        root.setCenter(boardPane);
        root.setRight(moveList);
        root.setBottom(statusBar);

        Scene scene = new Scene(root, 900, 600);
        stage.setTitle("Rush Hour Solver");
        stage.setScene(scene);
        stage.show();

        loadBtn.setOnAction(e -> loadBoard(stage));
        solveBtn.setOnAction(e -> runSolver());
    }

    private void loadBoard(Stage stage) {
        FileChooser fc = new FileChooser();
        fc.getExtensionFilters().add(new
FileChooser.ExtensionFilter("Text board", "*.txt"));
        File f = fc.showOpenDialog(stage);
        if (f == null) return;
        try {
            currentBoard = Input.readBoardFromFile(f.getAbsolutePath());
            drawBoard(currentBoard);
            moveList.getItems().clear();
            visitedLabel.setText("Visited: 0");
            timeLabel.setText("Time: 0 ms");
            stepLabel.setText("Step: 0");
        } catch (Exception ex) {
            showErr("Failed to load board:\n" + ex.getMessage());
        }
    }

```

```

}

private void runSolver() {
    if (currentBoard == null) {
        showError("Load a board first.");
        return;
    }
    moveList.getItems().setAll("🕒 solving ...");
    visitedLabel.setText("Visited: 0");
    timeLabel.setText("Time: 0 ms");
    stepLabel.setText("Step: 0");
    Task<SolveResult> task = new Task<>() {
        @Override protected SolveResult call() {
            String algo = algoChoice.getValue();
            String heuristic = heuristicChoice.getValue();
            int mode = 0;
            if (heuristic.equals("Blockers + Second Blockers")) {
                mode = 1;
            } else if (heuristic.equals("Distance Only")) {
                mode = 2;
            }
            long start = System.currentTimeMillis();
            SolveResult result = null;
            if (algo.equals("GBFS (Greedy)")) {
                result = GBFS.solve(currentBoard, mode);
            } else if (algo.equals("A*")) {
                result = AStar.solve(currentBoard, mode);
            } else if (algo.equals("UCS")) {
                result = UCS.solve(currentBoard, mode);
            } else if (algo.equals("IDS")) {
                result = IDS.solve(currentBoard, mode);
            }
            long end = System.currentTimeMillis();
            return new SolveResult(result.solution,
result.nodesVisited, end - start);
        }
    };
    task.setOnSucceeded(e -> {
        SolveResult result = task.getValue();
        lastResult = result;
        exportButton.setDisable(result.solution == null);
        if (result.solution == null) {
            moveList.getItems().setAll("❌ No solution");
            return;
        }
        List<object.Node> path = result.solution.getPath();
        List<Board> boards = path.stream().map(n ->
n.board).toList();
        java.util.List<String> steps = new java.util.ArrayList<>();
        for (int i = 0; i < path.size(); i++) {
            if (i == 0) {
                steps.add("Start");
            } else {
                steps.add(i + " : " + (path.get(i).move == null ?

```

```

"Unknown" : path.get(i).move));
        }
    }
    moveList.getItems().setAll(steps);
    animate(boards);
    visitedLabel.setText("Visited: " + result.nodesVisited);
    timeLabel.setText("Time: " + result.timeMs + " ms");
    stepLabel.setText("Step: " + (path.size() - 1));
    });
    task.setOnFailed(e -> showErr("Solver crashed: " +
task.getException()));
    new Thread(task).start();
}

// render
private void drawBoard(Board b) {
    drawBoard(b, false);
}

private void drawBoard(Board b, boolean hidePrimary) {
    int topExtra = (b.getGoalRow() < 0) ? 1 : 0;
    int bottomExtra = (b.getGoalRow() >= b.height) ? 1 : 0;
    int leftExtra = (b.getGoalCol() < 0) ? 1 : 0;
    int rightExtra = (b.getGoalCol() >= b.width) ? 1 : 0;
    double w = (b.width + leftExtra + rightExtra) * CELL;
    double h = (b.height + topExtra + bottomExtra) * CELL;
    boardCanvas.setWidth(w);
    boardCanvas.setHeight(h);

    GraphicsContext g = boardCanvas.getGraphicsContext2D();
    g.clearRect(0,0,w,h);

    double offsetX = leftExtra * CELL;
    double offsetY = topExtra * CELL;

    g.setStroke(Color.LIGHTGRAY);
    for (int r = 0; r <= b.height; r++) g.strokeLine(offsetX, offsetY
+ r * CELL, offsetX + b.width * CELL, offsetY + r * CELL);
    for (int c = 0; c <= b.width; c++) g.strokeLine(offsetX + c *
CELL, offsetY, offsetX + c * CELL, offsetY + b.height * CELL);

    int gateRow = b.getGoalRow() + topExtra;
    int gateCol = b.getGoalCol() + leftExtra;
    g.setFill(Color.DARKSEAGREEN);
    g.fillRect(gateCol * CELL, gateRow * CELL, CELL, CELL);
    g.setFill(Color.WHITE);
    g.setFont(javafx.scene.text.Font.font(24));
    g.setTextAlign(javafx.scene.text.TextAlignment.CENTER);
    g.setTextBaseline(javafx.geometry.VPos.CENTER);
    g.fillText("K", gateCol * CELL + CELL/2, gateRow * CELL +
CELL/2);

    for (Piece p : b.pieces) {
        if (hidePrimary && p.isPrimary) continue; // !! skip when

```

asked

```
        Color col = p.isPrimary ? Color.CRIMSON : Color.CADETBLUE;
        double x = offsetX + p.col * CELL + 2;
        double y = offsetY + p.row * CELL + 2;
        double pw = (p.isHorizontal ? p.length : 1) * CELL - 4;
        double ph = (p.isHorizontal ? 1 : p.length) * CELL - 4;

        g.setFill(col);
        g.fillRoundRect(x, y, pw, ph, 8, 8);

        g.setFill(Color.WHITE);
        g.setFont(javafx.scene.text.Font.font(18));
        g.setTextAlign(javafx.scene.text.TextAlignment.CENTER);
        g.setTextBaseline(javafx.geometry.VPos.CENTER);
        g.fillText(String.valueOf(p.id), x + pw / 2, y + ph / 2);
    }
}

// animasi steps
private void animate(java.util.List<object.Board> path) {
    new Thread(() -> {
        for (object.Board b : path) {
            Platform.runLater(() -> drawBoard(b));
            try { Thread.sleep(400); } catch (InterruptedException
ignored) {}
        }
        Board last = path.get(path.size() - 1);
        if (Helper.isGoal(last)) {
            Platform.runLater(() -> drawBoard(last, true));
        }
    }).start();
}

private void exportToFile(Stage stage) {
    if (lastResult == null || lastResult.solution == null) {
        showError("Solve dulu sebelum mengekspor.");
        return;
    }

    FileChooser fc = new FileChooser();
    fc.setTitle("Save Solution");
    fc.setInitialFileName("output.txt");
    fc.getExtensionFilters().add(new
FileChooser.ExtensionFilter("Text Files", "*.txt"));
    File f = fc.showSaveDialog(stage);
    if (f == null) return;

    try {
        utils.Output.write(f.getAbsolutePath(), lastResult);
        new Alert(Alert.AlertType.INFORMATION,
            "Solution saved to:\n" + f.getAbsolutePath(),
            ButtonType.OK).showAndWait();
    } catch (IOException ex) {
```

```

        showErr("Failed to save file:\n" + ex.getMessage());
    }
}
private void showErr(String msg) {
    new Alert(Alert.AlertType.ERROR, msg,
ButtonType.OK).showAndWait();
}

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

```

2.2.13. Kelas Output

Kelas Output	
Metode	Penjelasan
write(String filepath, SolveResult result) throws IOException	Metode untuk menulis hasil output ke file .txt.
<pre> package utils; import java.io.BufferedWriter; import java.io.FileWriter; import java.io.IOException; import java.util.List; import algo.SolveResult; import object.Board; import object.Node; public class Output { public static void write(String filepath, SolveResult result) throws IOException { try (BufferedWriter w = new BufferedWriter(new FileWriter(filepath))) { Node goal = result.solution; int visited = result.nodesVisited; w.write("==== RUSH HOUR SOLUTION ====="); w.newLine(); if (goal == null) { w.write("No solution found."); w.newLine(); w.write("Nodes visited: " + visited); w.newLine(); return; } List<Node> path = goal.getPath(); </pre>	

```

        int moves = path.size() - 1;

        w.write("Total moves    : " + moves);
        w.newLine();
        w.write("Nodes visited : " + visited);
        w.newLine();
        w.write("-----");
        w.newLine();

        for (int i = 0; i < path.size(); i++) {
            Node n = path.get(i);
            String moveDesc = (n.move == null ? "START" : n.move);
            w.write("Step " + i + " : " + moveDesc);
            w.newLine();

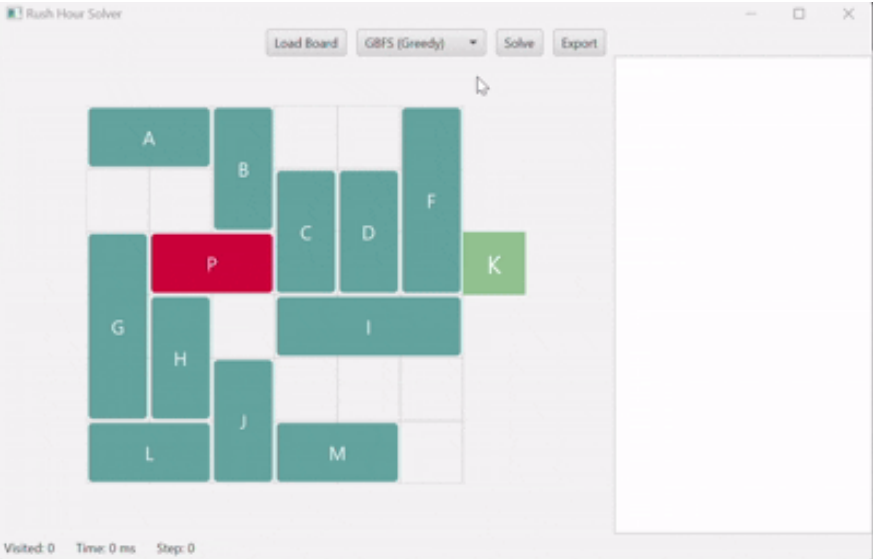
            Board b = n.board;
            for (int r = 0; r < b.height; r++) {
                w.write(new String(b.grid[r]));
                w.newLine();
            }
            w.write("-----");
            w.newLine();
        }
    }
}

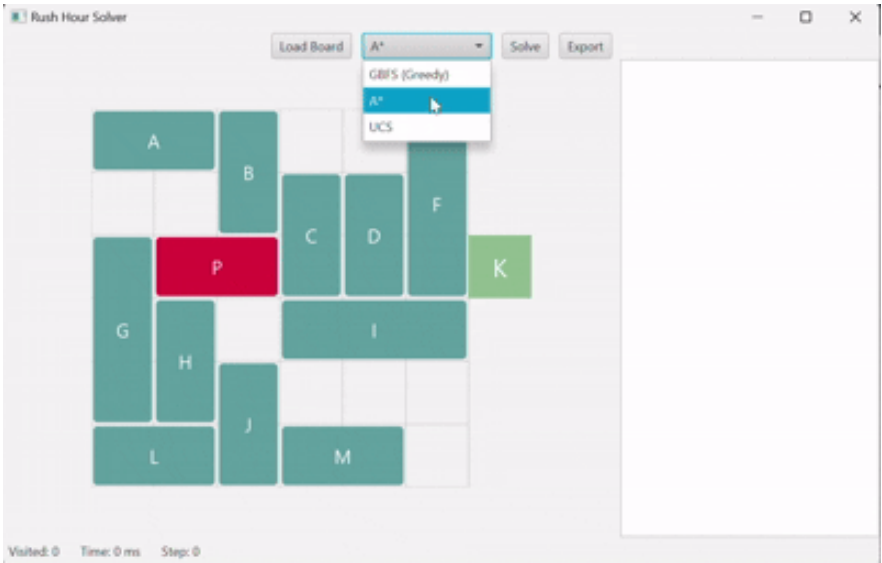
```

BAB III
EKSPERIMEN DAN ANALISIS

3.1. Test Case 1

Input1.txt	<pre>6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.</pre>
GBFS	

	<pre>==== RUSH HOUR SOLUTION ==== Total moves : 7 Nodes visited : 11 ----- Papan awal: AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. ----- Gerakan 1 : C U 1 AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. ----- Gerakan 2 : P R 1 AABC.F ..BCDF G.PPDFK GH.III GHJ... LLJMM. ----- Gerakan 5 : I L 1 AABCDF ..BCDF G..PPFK GHIII. GHJ... LLJMM. ----- Gerakan 6 : F D 3 AABCD. ..BCD. G..PP.K GHIIIF GHJ..F LLJMMF ----- Gerakan 7 : P R 1 AABCD. ..BCD. G..PPK GHIIIF GHJ..F LLJMMF -----</pre>
A*	

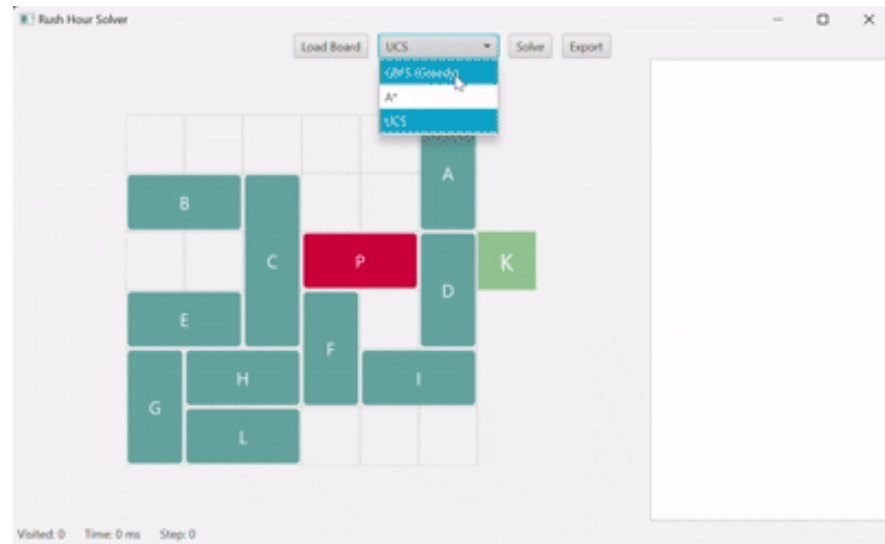
	<pre>===== RUSH HOUR SOLUTION ===== Total moves : 6 Nodes visited : 18 ----- Papan awal: AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. ----- Gerakan 1 : C U 1 AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. ----- Gerakan 2 : D U 1 AABCDF ..BCDF GPP..FK GH.III GHJ... LLJMM. ----- Gerakan 3 : P R 2 AABCDF ..BCDF G..PPFK GH.III GHJ... LLJMM. ----- Gerakan 4 : I L 1 AABCDF ..BCDF G..PPFK GH.III GHJ... LLJMM. ----- Gerakan 5 : F D 3 AABCD. ..BCD. G..PP.K GH.IIIF GHJ..F LLJMMF ----- Gerakan 6 : P R 1 AABCD. ..BCD. G...PPK GH.IIIF GHJ..F LLJMMF -----</pre>
UCS	

	<pre> ===== RUSH HOUR SOLUTION ===== Total moves : 5 Nodes visited : 188 ----- Papan awal: AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. ----- Gerakan 1 : C U 1 AABC.F ..BCDF GPP.DFK GH.III GHJ... LLJMM. ----- Gerakan 2 : D U 1 AABCDF ..BCDF GPP..FK GH.III GHJ... LLJMM. ----- Gerakan 3 : I L 1 AABCDF ..BCDF GPP..FK GH.III GHJ... LLJMM. ----- Gerakan 4 : F D 3 AABCD. ..BCD. GPP...K GH.III GHJ..F LLJMMF ----- Gerakan 5 : P R 3 AABCD. ..BCD. G...PPK GH.III GHJ..F LLJMMF ----- </pre>
--	---

3.2. Test Case 2

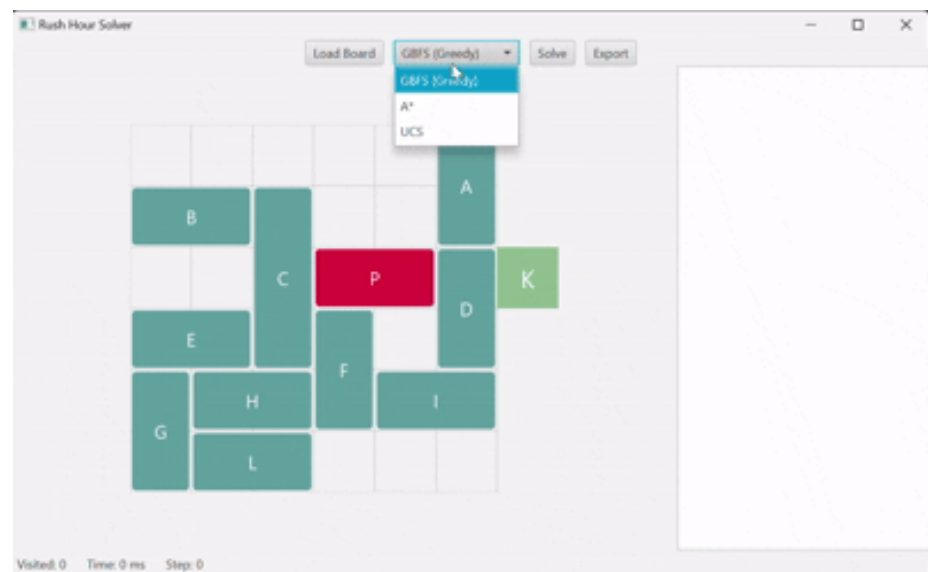
Input2.txt	<pre> 6 6 11A BBC..A ..CPPDK EECF.D GHHFII GLL... </pre>
------------	--

GBFS

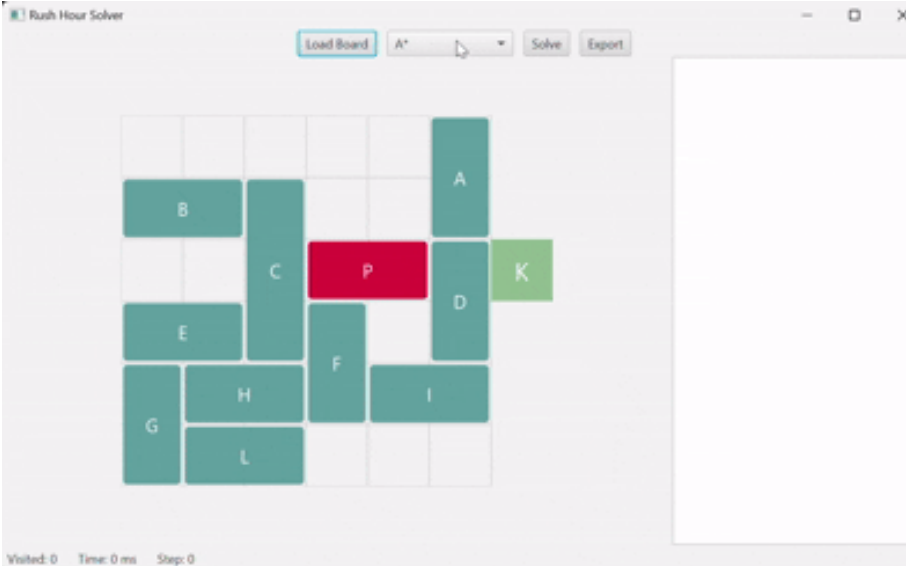


Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

A*



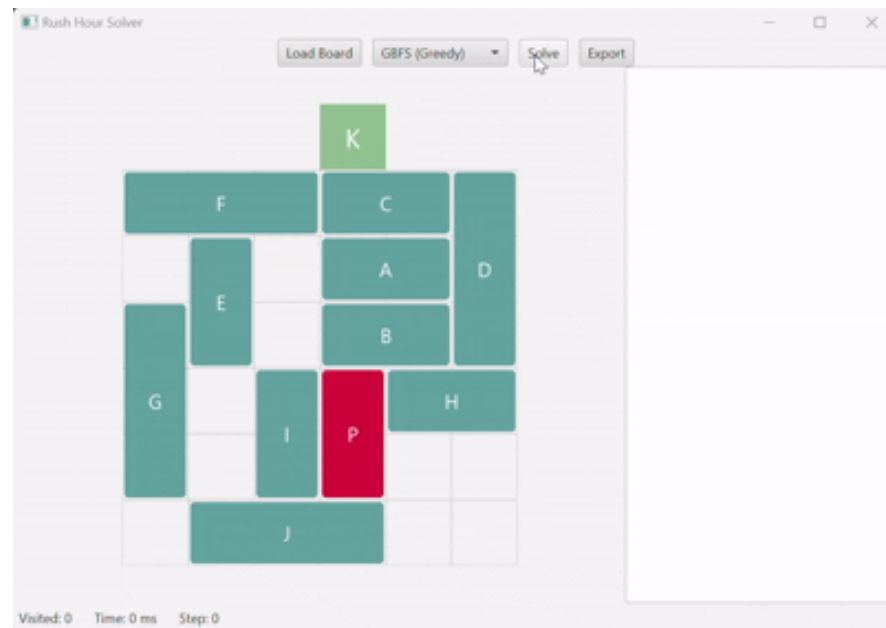
Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

UCS	 <p>Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.</p>
-----	--

3.3. Test Case 3

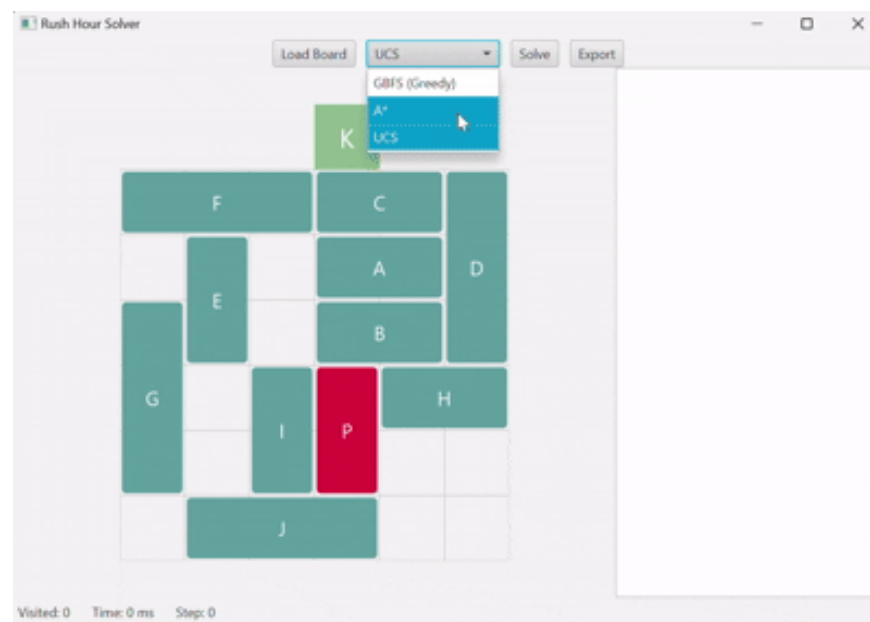
Input3.txt	<pre>6 6 12 K FFFCD .E.AAD GE.BBD G.IPHH G.IPK. .JJJK.</pre>
------------	--

GBFS

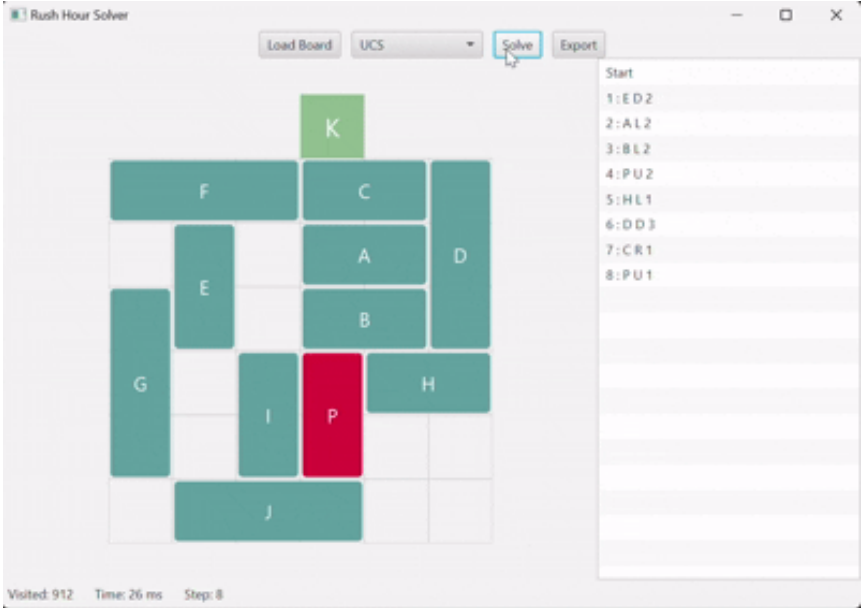


Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

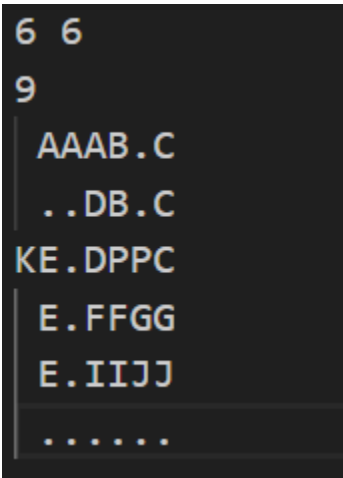
A*



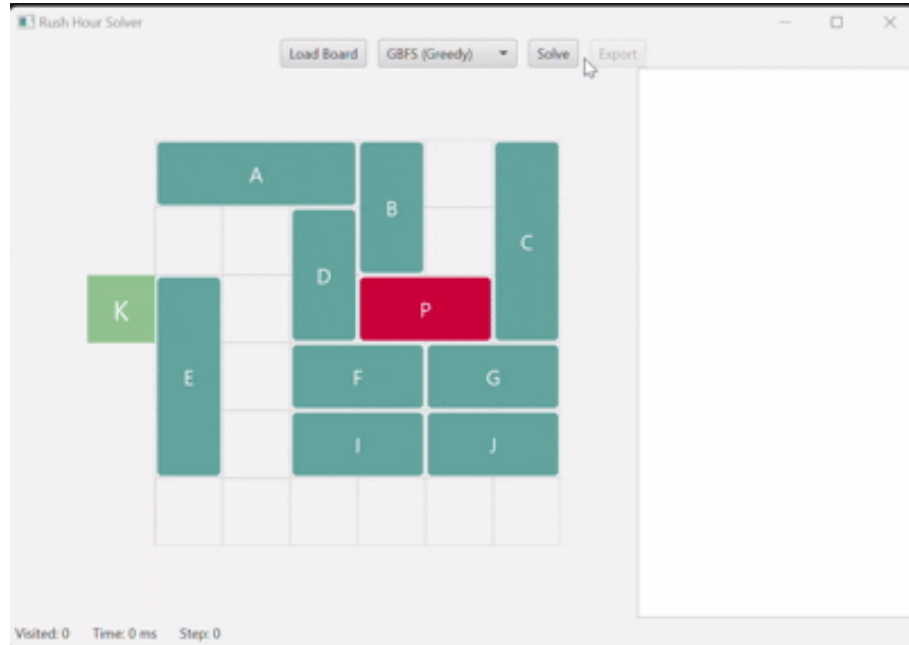
Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

<p>UCS</p>	 <p>Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.</p>
------------	--

3.4. Test Case 4

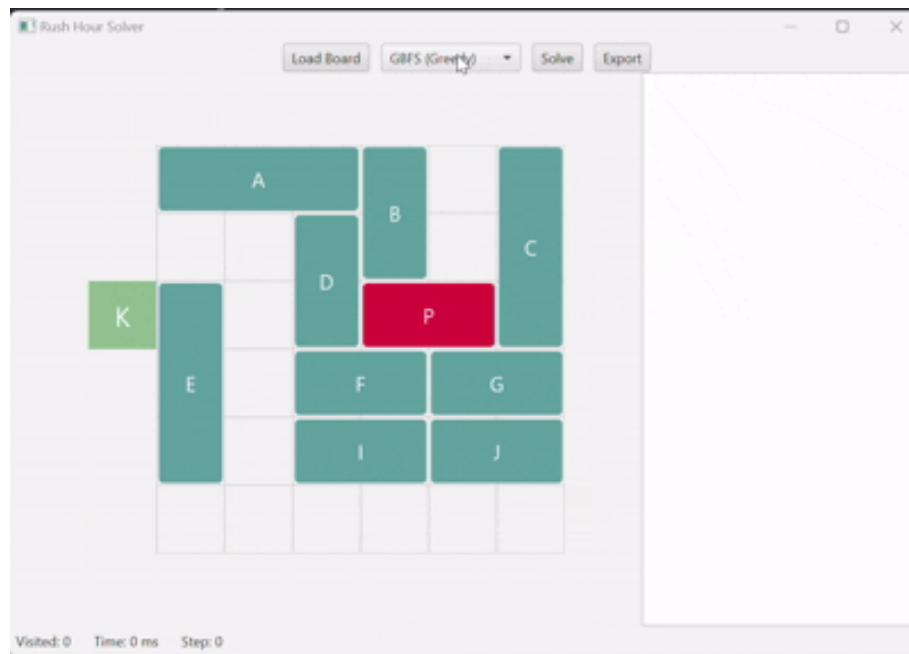
<p>Input4.txt</p>	
-------------------	--

GBFS

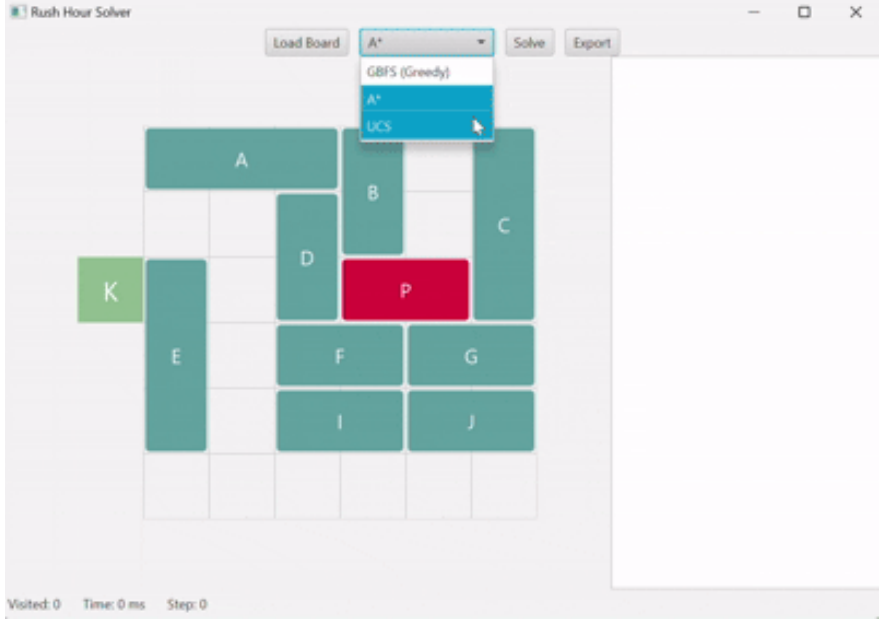


Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

A*



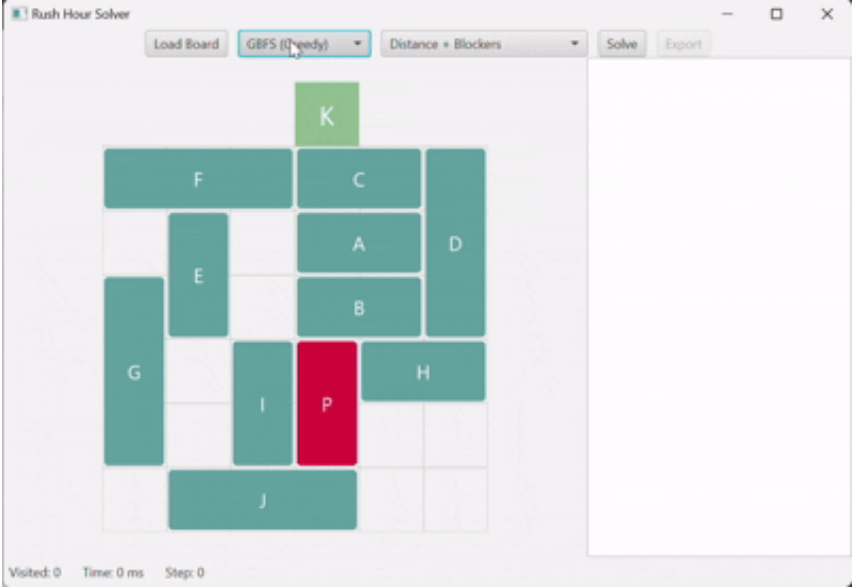
Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

UCS	 <p>Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.</p>
-----	--

3.5. Test Case 5 (Bonus)

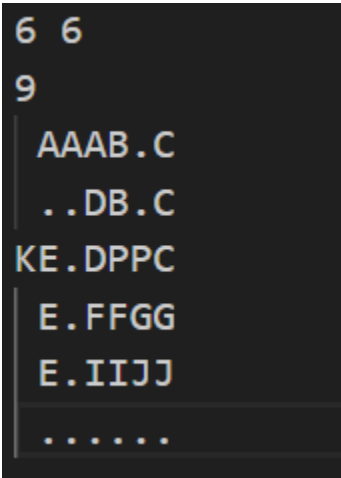
*Test Case tambahan ini ditujukan untuk menunjukkan hasil algoritma alternatif yaitu IDS (bonus). Test Case yang digunakan adalah Input3.txt

Input3.txt	<pre> 6 6 12 K FFFCCD .E.AAD GE.BBD G.IPHH G.IPK. .JJJK. </pre>
------------	--

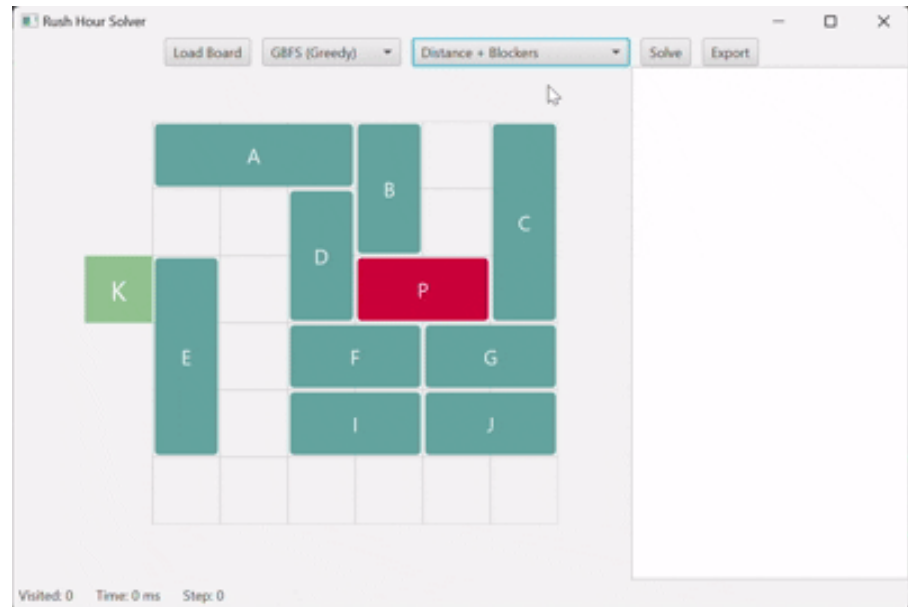
IDS	 <p>Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.</p>
-----	--

3.6. Test Case 6 (Bonus)

*Test Case tambahan ini ditujukan untuk menunjukkan pengaruh dari perbedaan nilai heuristik (bonus). Pada test case ini hanya akan digunakan GBFS yang $f(n)$ nya sangat bergantung pada $h(n)$. Test Case yang digunakan adalah Input4.txt

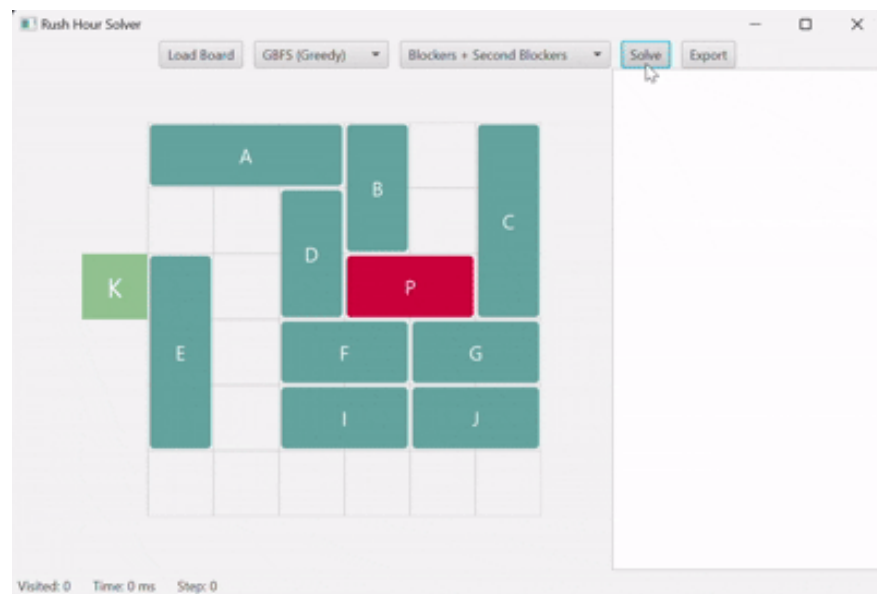
Input4.txt	 <pre> 6 6 9 AAAB.C ..DB.C KE.DPPC E.FFGG E.IIJJ </pre>
------------	---

Distance +
Blockers



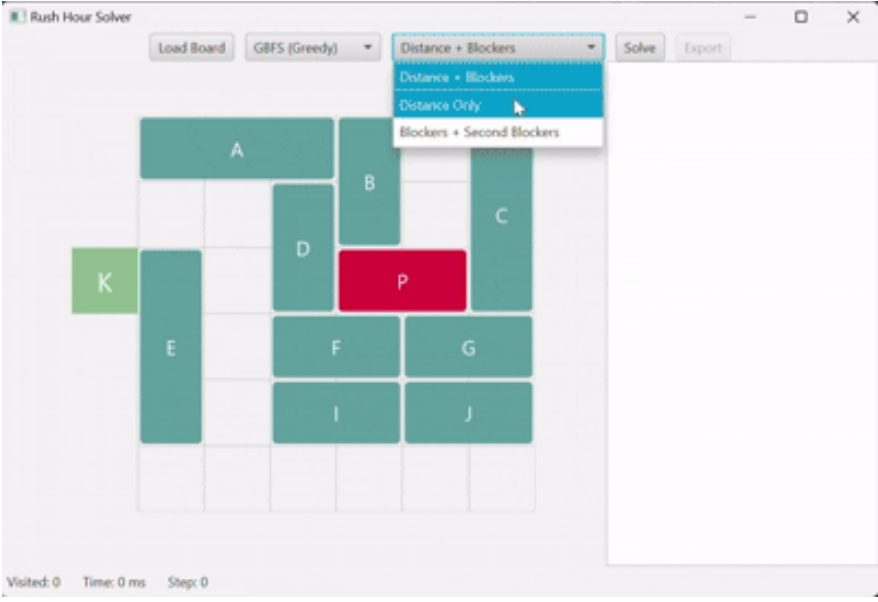
Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

Blockers +
2nd Blockers



Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

Distance Only



Output.txt lengkap dapat dilihat di repository beserta dengan output testcase lainnya.

3.7. Hasil Analisis Percobaan Algoritma Pathfinding

Berdasarkan hasil percobaan, dapat dilihat bahwa jumlah simpul yang dikunjungi untuk setiap algoritma berbeda-beda. Algoritma GBFS memiliki jumlah simpul yang dikunjungi yang paling sedikit. Hal ini terjadi karena GBFS hanya mengejar simpul dengan estimasi jarak ke tujuan (heuristik) terkecil tanpa mempertimbangkan biaya perjalanan sejauh ini ($g(n)$). Namun, algoritma greedy ini menyebabkan GBFS cenderung melewati solusi optimal dan menghasilkan jalur solusi yang lebih panjang dibandingkan algoritma lain terlihat dari jumlah step atau langkah yang paling banyak dibandingkan dengan algoritma lain. Karena algoritmanya yang langsung memilih langkah terbaik di sebuah state tertentu, waktu yang dibutuhkan algoritma GBFS beragam dari yang sangat cepat apabila kasusnya sederhana hingga membutuhkan waktu lebih lama untuk kasus yang lebih kompleks.

Sebaliknya, Uniform Cost Search (UCS) memiliki jumlah simpul yang dikunjungi paling banyak dan cenderung membutuhkan waktu eksekusi yang lebih lama. Secara umum, jumlah langkah yang dihasilkan algoritma UCS pada percobaan ini paling sedikit. Hal ini terjadi karena UCS selalu menjamin solusi optimal dengan mengevaluasi semua kemungkinan dengan mengukur jarak ke node tujuan melalui heuristik tertentu. Akan tetapi, karena mengevaluasi semua kemungkinan ini pula, jumlah simpul yang dikunjungi serta waktu eksekusinya lama.

Algoritma A* yang menggabungkan kedua algoritma tadi dengan memanfaatkan gabungan antara $g(n)$ dan $h(n)$, A* dapat menemukan solusi optimal apabila heuristiknya admissible. Secara keseluruhan, A* menghasilkan jalur solusi dengan jumlah langkah dan

visited node di antara GBFS dan UCS. Akan tetapi, di segi penggunaan waktu, algoritma A* merupakan algoritma yang paling cepat mendapatkan solusi.

Algoritma IDS memakan waktu yang sangat lama, mengunjungi lebih banyak simpul, dan menghasilkan lebih banyak langkah dibandingkan dengan algoritma lainnya. Hal ini terjadi karena simpul-simpul dekat akar akan terus-menerus dikunjungi kembali di setiap iterasi sehingga total simpul yang dikunjungi akan jauh lebih besar terutama saat solusi berada di kedalaman besar. Langkah yang ditemukan algoritma IDS tidak optimal karena IDS menggunakan DFS di setiap iterasi sehingga solusi yang ditemukan pertama kali belum tentu yang paling pendek atau paling murah, hanya yang ditemukan pertama pada batas kedalaman tersebut.

Berdasarkan hasil percobaan terhadap tiga heuristik berbeda, didapati bahwa ada perbedaan waktu eksekusi dan jumlah langkah solusi. Heuristik pertama, yang menghitung jumlah piece yang secara langsung menghalangi jalur primary piece, menghasilkan waktu eksekusi yang paling cepat karena perhitungannya ringan, tetapi jumlah langkah yang diambil tetap cukup tinggi karena heuristik ini tidak memperhatikan kerumitan dalam memindahkan penghalang tersebut. Sementara itu, heuristik kedua, yang menghitung jumlah piece yang menghalangi piece yang menghalangi primary piece, menghasilkan jumlah langkah yang paling sedikit meskipun waktu eksekusinya relatif lebih lama. Hal ini terjadi karena heuristik yang paling memberikan gambaran realistis terhadap kondisi papan. Heuristik ketiga, yaitu menghitung jarak langsung dari primary piece ke pintu keluar, cenderung menghasilkan waktu eksekusi yang lebih lama dan jumlah langkah yang paling banyak. Hal ini disebabkan karena heuristik tersebut sederhana dan tidak memperhitungkan adanya hambatan di jalur primary piece. Dengan demikian, heuristik kedua dapat dianggap sebagai yang paling efektif dalam menghasilkan solusi optimal tetapi membutuhkan waktu lebih banyak untuk perhitungan.

Dari hasil percobaan ini, dapat disimpulkan bahwa terdapat hal yang harus dikorbankan antara optimalitas solusi dan efisiensi waktu/ruang pencarian untuk tiap algoritma dan heuristik. Pemilihan algoritma yang digunakan akan tergantung pada kebutuhan apakah lebih mengutamakan kecepatan, jumlah node minimal, atau solusi paling pendek.

BAB IV

IMPLEMENTASI BONUS

5.1. GUI (Graphical User Interface)

Program Rush Hour Solver ini menggunakan JavaFX sebagai *framework* untuk menciptakan antarmuka pengguna yang informatif. GUI dirancang dengan struktur `BorderPane` yang membagi layar menjadi beberapa area fungsional, panel kontrol di bagian atas, board di bagian tengah, daftar langkah di bagian kanan, dan panel status di bagian bawah.

Panel kontrol program ini dilengkapi dengan tombol untuk me-load papan permainan, menjalankan algoritma penyelesaian, dan mengeksport hasil. Program ini juga menyediakan berbagai pilihan algoritma (GBFS, A*, UCS, dan IDS) dan heuristik ("Distance + Blockers", "Distance Only", "Blockers + Second Blockers") yang dapat disesuaikan berdasarkan kebutuhan pengguna. Fleksibilitas ini memungkinkan pengguna untuk membandingkan efektivitas berbagai pendekatan dalam menyelesaikan puzzle Rush Hour.

Visualisasi solusi pada program ini diimplementasikan dengan animasi langkah-per-langkah yang menunjukkan pergerakan *piece* pada *board*. Setelah ditemukan solusi, `statusBar` akan memberikan informasi berupa jumlah node yang dikunjungi, waktu pemrosesan, dan jumlah langkah solusi. Terakhir, fitur ekspor memungkinkan pengguna menyimpan solusi yang ditemukan ke dalam sebuah file teks.

5.2. Algoritma Alternatif: Iterative Deepening Search

Dalam implementasi di program ini, di dalam method `dls` pertama-tama dicek apakah string hash dari papan `current` sudah ada di `visited`. Jika ya, fungsi langsung return `null` untuk mencegah looping terus menerus, jika belum maka `visited.add(hash)`. Selanjutnya jika `current` sudah memenuhi kondisi goal, `dls` langsung return `current`. Apabila belum goal tapi `limit == 0`, artinya kedalaman maksimal untuk iterasi ini telah tercapai, sehingga fungsi juga mengembalikan `null`. Jika masih ada "jatah" kedalaman, `dls` memanggil `Helper.expand(current, false, false)` untuk menghasilkan semua child node valid, lalu memanggil dirinya sendiri dengan `limit-1`. Jika salah satu panggilan rekursif menemukan goal, hasilnya langsung di-"bubble up" hingga `solve()`.

Kembali ke `solve()`, setelah `dls` selesai, apabila hasilnya bukan `null` berarti solusi telah ditemukan pada kedalaman tersebut: algoritma segera mengembalikan sebuah `SolveResult` yang berisi simpul goal dan jumlah total simpul unik yang dieksplorasi, yaitu `visitedCount + visited.size()`. Jika `dls` belum menemukan solusi, maka `visitedCount += visited.size()` (mengakumulasi semua simpul yang dikunjungi pada level ini), `depthLimit++`, dan loop diulang dari awal.

```
function solve(Board start, int mode) → SolveResult
```

```
{ Mencari solusi jalur minimum dari papan awal menggunakan algoritma IDS }
```

Deklarasi

```
depthLimit    : int
visitedCount   : int
visited        : Set<String>
result         : Node
```

Algoritma:

```
depthLimit ← 0
visitedCount ← 0
```

loop

```
visited ← new HashSet<>()
result ← dls( new Node(start, null, null, 0, 0),
              depthLimit,
              Visited, mode)
```

```
if (result ≠ null) then
    → new SolveResult(result, visitedCount + visited.size())
```

```
visitedCount ← visitedCount + visited.size()
depthLimit ← depthLimit + 1
```

end loop

```
function dls(Node current, int limit, Set<String> visited, int mode) → Node
{ Depth-Limited Search rekursif hingga kedalaman "limit" }
```

Deklarasi

```
hash    : String
child    : Node
found    : Node
```

Algoritma:

```
hash ← current.board.toString()
if (visited.contains(hash)) then
    → null
visited.add(hash)
```

```
if (Helper.isGoal(current.board)) then
    → current
```

```
if (limit == 0) then
    → null
```

```
for each child in Helper.expand(current, false, false, mode) do
    found ← dls(child, limit - 1, visited)
    if (found ≠ null) then
        → found
end for
```

→ null

5.3. Heuristik Alternatif

Dalam program Rush Hour Solver ini, kami telah mengimplementasikan tiga jenis heuristik untuk membantu proses pencarian solusi. Heuristik pertama yaitu "Distance Only" merupakan pendekatan paling sederhana yang menghitung jarak langsung dari

primary piece ke pintu keluar. Implementasinya dilakukan melalui metode `distanceOnly()` yang menghitung selisih absolut antara posisi kendaraan utama dengan posisi pintu keluar, dengan mempertimbangkan orientasi kendaraan (horizontal atau vertikal) dan arah pintu keluar (kiri, kanan, atas, atau bawah).

Heuristik kedua, "Distance + Blockers" menggabungkan jarak *primary piece* ke pintu keluar dengan jumlah kendaraan lain yang menghalangi jalur. Implementasinya terdapat dalam metode `blockingDistance()` yang menjumlahkan hasil dari `distanceOnly()` dengan hasil dari `countDirectBlockers()`. Pendekatan ini *admissible* karena mempertimbangkan bahwa setiap penghalang akan memerlukan setidaknya satu langkah tambahan untuk dipindahkan.

Heuristik terakhir yaitu "Blockers + 2nd Blockers" diimplementasikan dalam metode `blockingPlusPlus()`. Selain menghitung jarak dan jumlah penghalang langsung, heuristik ini juga mempertimbangkan kendaraan penghalang yang juga terhalang sehingga tidak dapat langsung dipindahkan. Metode ini mengevaluasi setiap penghalang langsung dan menambahkan nilai heuristik jika penghalang tersebut tidak memiliki ruang untuk bergerak. Pendekatan ini memberikan estimasi yang lebih akurat tentang kompleksitas situasi dan berpotensi menghasilkan pencarian yang lebih efisien dalam kasus yang kompleks.

LAMPIRAN

6.1. Tautan *repository* Github

https://github.com/naylzhra/Tucil3_13523050_13523079

6.2. Tabel Checklist

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	✓	
2	Program berhasil dijalankan	✓	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4	Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	✓	
5	[Bonus] Implementasi algoritma pathfinding alternatif	✓	
6	[Bonus] Implementasi 2 atau lebih heuristik alternatif	✓	
7	[Bonus] Program memiliki GUI	✓	
8	Program dan laporan dibuat (kelompok) sendiri	✓	

DAFTAR PUSTAKA

- [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian1.pdf) (Diakses 19 Mei 2025)
- [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) (Diakses 19 Mei 2025)