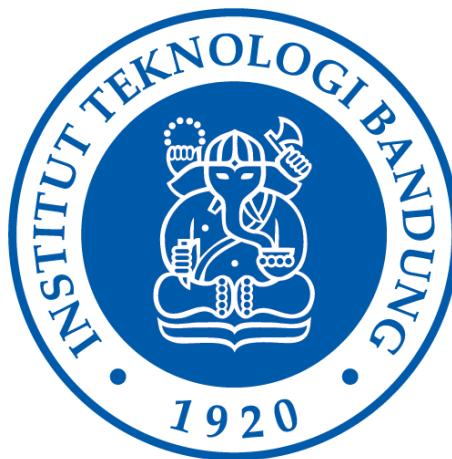


IF2211 - Strategi Algoritma

Laporan Tugas Kecil 3

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun Oleh:

Adinda Putri 13523071

Shanice Feodora Tjahjono 13523097

**Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung**

2025

DAFTAR ISI

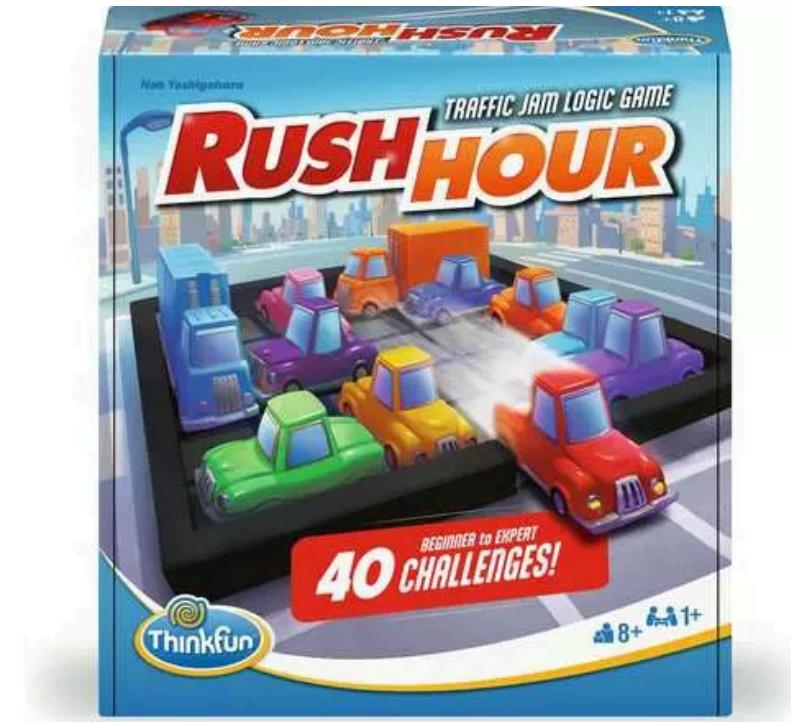
DAFTAR ISI.....	2
BAB 1.....	4
Pendahuluan.....	4
1.1. Deskripsi Tugas.....	4
Ilustrasi kasus :.....	5
1.2. Algoritma Pathfinding.....	8
1.2.1 Algoritma Uniform Cost Search (UCS).....	8
1.2.2 Algoritma Greedy Best First Search (GBFS).....	8
1.2.3 Algoritma A*.....	9
1.2.4 Algoritma Dijkstra.....	9
BAB 2.....	11
Algoritma Program.....	11
2.1 Algoritma Pathfinding.....	11
2.2 Pseudocode Algoritma.....	12
2.2.1 Algoritma Uniform Cost Search (UCS).....	12
2.2.2 Algoritma Greedy Best First Search (GBFS).....	13
2.2.3 Algoritma A*.....	15
2.2.4 Algoritma Dijkstra.....	18
BAB 3.....	22
Source Code Program.....	22
3.1 Board.java.....	23
3.2 GameState.java.....	34
3.3 Move.java.....	40
3.4 Piece.java.....	43
3.5 AStar.java.....	48
3.6 Dijkstra.java.....	53
3.7 GBFS.java.....	58
3.8 UCS.java.....	62
3.9 Heuristics.java.....	66
3.10 AboutPage.java.....	70
3.11 BoardPane.java.....	73
3.12 ControlPanel.java.....	83
3.13 CreatorPage.java.....	83
3.14 Renderer.java.....	87
3.15 WelcomePage.java.....	126
3.16 App.java.....	128
3.17 SaveSolutionHandler.java.....	135

3.18 Main.java.....	139
BAB 4.....	146
Pengujian.....	146
3.1. Algoritma A*	146
3.2. Algoritma Greedy Best First Search (GBFS).....	148
3.3. Algoritma Uniform Cost Search (UCS).....	150
3.4. Algoritma Dijkstra.....	153
BAB 5.....	156
Hasil Analisis.....	156
BAB 6.....	157
Implementasi Bonus.....	157
5.1. Bonus 1: Implementasi Algoritma Alternatif.....	157
5.2. Bonus 2: Implementasi Heuristic Alternatif.....	157
5.2.1. Manhattan Distance.....	158
5.2.2. Blocking Vehicles.....	158
5.2.3. Combined Heuristic.....	158
5.3. Bonus 3: Graphical User Interface.....	159
Lampiran.....	163
Pranala Repository.....	163
Daftar Pustaka.....	164

BAB 1

Pendahuluan

1.1. Deskripsi Tugas



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*.

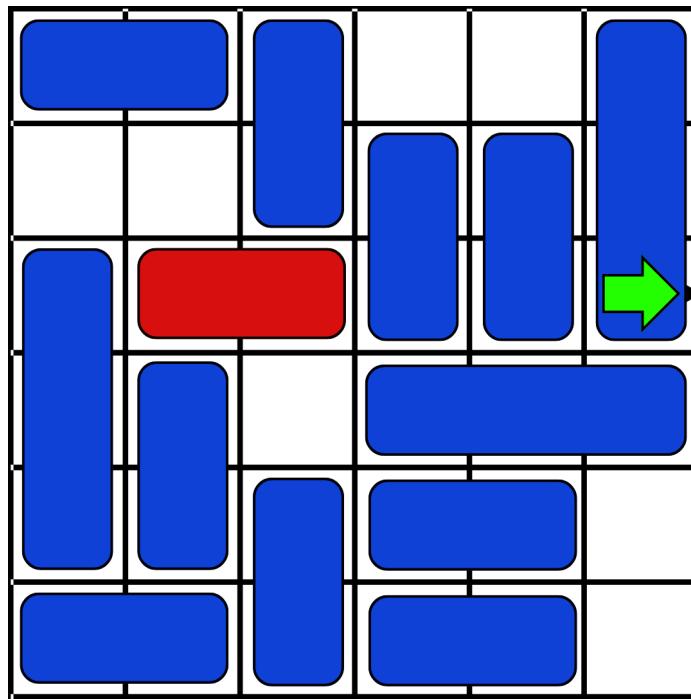
Hanya *primary piece* yang dapat digerakkan **keluar papan melewati pintu keluar**. *Piece* yang bukan *primary piece* tidak dapat digerakkan keluar papan. Papan memiliki

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.

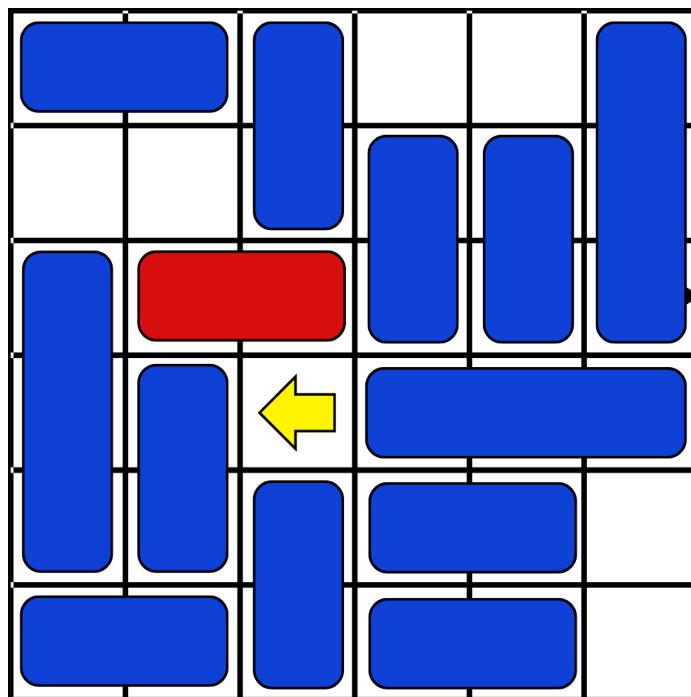
Ilustrasi kasus :

Diberikan sebuah *papan* berukuran 6 x 6 dengan 12 *piece* kendaraan dengan 1 *piece* merupakan *primary piece*. *Piece* ditempatkan pada *papan* dengan posisi dan orientasi sebagai berikut.

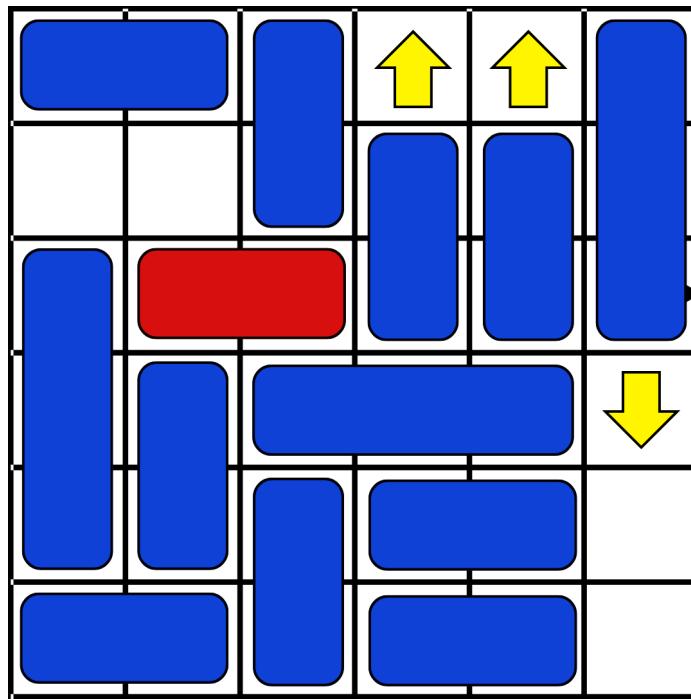


Gambar 2. Awal Permainan Game Rush Hour

Pemain dapat menggeser-geser *piece* (termasuk *primary piece*) untuk membentuk jalan lurus antara *primary piece* dan *pintu keluar*.

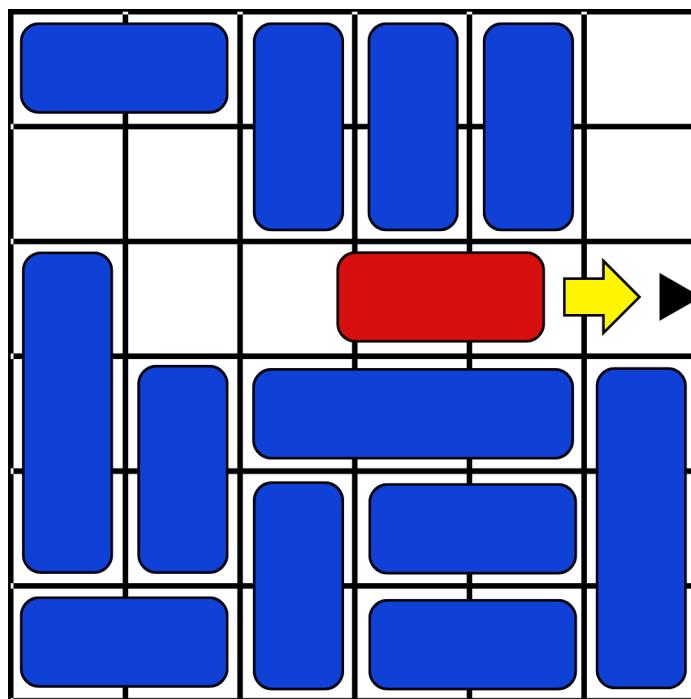


Gambar 3. Gerakan Pertama Game Rush Hour



Gambar 4. Gerakan Kedua Game Rush Hour

Puzzle berikut dinyatakan telah selesai apabila *primary piece* dapat digeser keluar papan melalui *pintu keluar*.



Gambar 5. Pemain Menyelesaikan Permainan

1.2. Algoritma Pathfinding

Algoritma pathfinding adalah metode komputasional yang digunakan untuk menemukan jalur optimal dari titik awal ke titik tujuan dalam suatu ruang pencarian, seperti graf atau peta. Jalur optimal ini biasanya meminimalkan biaya tertentu, seperti jarak, waktu, atau energi. Algoritma ini sangat penting dalam berbagai aplikasi, termasuk navigasi GPS, perencanaan rute robot, dan permainan komputer.

1.2.1 Algoritma Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian jalur berbasis graf yang termasuk dalam kategori *uninformed search*, di mana pemilihan simpul berikutnya didasarkan sepenuhnya pada biaya aktual dari simpul awal tanpa memperhitungkan estimasi ke tujuan. UCS mengembangkan simpul dengan biaya kumulatif terendah terlebih dahulu, menggunakan struktur data *priority queue* yang diurutkan berdasarkan fungsi biaya $g(n)$, yaitu total biaya jalur dari simpul awal ke simpul n .

Dalam implementasinya, UCS mirip dengan algoritma Dijkstra, namun UCS biasanya berhenti lebih awal, tepat setelah simpul tujuan di-*dequeue* dari *priority queue*, karena pada titik itu, jalur terpendek ke simpul tujuan sudah ditemukan. UCS menjamin jalur solusi yang optimal (biaya minimum) selama semua sisi pada graf memiliki bobot non-negatif. Untuk setiap ekspansi simpul, algoritma akan memeriksa tetangganya dan memperbarui antrian prioritas jika ditemukan jalur yang lebih murah menuju simpul tersebut. Karena tidak menggunakan heuristik, UCS memiliki kekurangan, yaitu kurang efisien pada graf besar atau ruang pencarian luas, terutama jika simpul tujuan berada jauh dari simpul awal. Namun, kelebihannya terletak pada jaminan optimalitas dan kesederhanaan logika secara algoritmik.

1.2.2 Algoritma Greedy Best First Search (GBFS)

Greedy Best First Search (GBFS) adalah algoritma pencarian jalur yang termasuk dalam kategori *informed search*, karena memanfaatkan fungsi heuristik $h(n)$ untuk memperkirakan jarak dari simpul n ke tujuan. Tidak seperti Uniform Cost Search yang mengevaluasi jalur berdasarkan biaya aktual dari awal ($g(n)$), GBFS hanya mempertimbangkan nilai heuristik, dan selalu memilih simpul yang tampaknya paling dekat ke tujuan secara estimatif.

Algoritma ini menggunakan *priority queue* yang diurutkan berdasarkan nilai $h(n)$ — semakin kecil nilai heuristiknya, semakin tinggi prioritas simpul tersebut untuk dieksplorasi. Karena hanya berfokus pada "kedekatan" ke tujuan, GBFS bisa sangat cepat dalam menemukan solusi, terutama pada graf besar, namun tidak menjamin bahwa solusi tersebut adalah jalur terpendek. GBFS rentan terhadap jebakan lokal (local minima), di mana simpul yang tampak menjanjikan justru mengarah ke jalan buntu atau jalur yang mahal. Oleh karena itu, performa GBFS sangat bergantung pada kualitas heuristik yang digunakan — semakin akurat estimasi $h(n)$, semakin efisien dan efektif jalur yang

ditemukan. Meskipun tidak optimal, GBFS sering dipakai dalam aplikasi *real-time* atau situasi di mana kecepatan lebih diutamakan dibanding optimalitas atau keefektifan.

1.2.3 Algoritma A*

A* (A-Star) Search adalah algoritma pencarian jalur yang termasuk dalam *informed search* dan dirancang untuk menyeimbangkan efisiensi dan optimalisasi. A* menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$ untuk menentukan urutan eksplorasi simpul, di mana $g(n)$ adalah biaya aktual dari simpul awal ke simpul n, dan $h(n)$ adalah fungsi heuristik yang memperkirakan biaya dari n ke simpul tujuan. Dengan menggabungkan kedua nilai ini, A* mempertimbangkan baik jalur yang sudah ditempuh maupun estimasi sisa jalur yang harus dilalui, menjadikannya strategi yang adaptif.

A* menggunakan *priority queue* untuk menyimpan simpul-simpul frontier, dengan selalu memilih simpul dengan nilai $f(n)$ terkecil untuk diekspansi. Jika heuristik $h(n)$ yang digunakan bersifat *admissible* (tidak pernah melebihi biaya sesungguhnya) dan *consistent* (memenuhi segitiga ketat), maka A* menjamin menemukan solusi optimal.

Secara teknis, A* merupakan perluasan dari Uniform Cost Search dan Greedy Best-First Search—jika $h(n) = 0$ maka A* menjadi UCS, dan jika $g(n) = 0$ maka A* menjadi GBFS. Dalam implementasinya, simpul-simpul disimpan dalam antrian prioritas berdasarkan nilai $f(n)$, dan simpul dengan $f(n)$ terkecil akan diekspansi lebih dahulu. A* bersifat lengkap (*complete*) dan optimal dalam kondisi tertentu, tetapi memiliki kompleksitas ruang dan waktu eksponensial dalam kasus terburuk karena perlu menyimpan seluruh simpul yang telah dikunjungi. Contoh klasik penggunaannya adalah pencarian rute kota seperti dari Arad ke Bucharest, dengan $g(n)$ sebagai jarak tempuh sejauh ini dan $h(n)$ berupa jarak garis lurus ke tujuan.

1.2.4 Algoritma Dijkstra

Algoritma Dijkstra adalah algoritma pencarian jalur terpendek dalam graf berbobot non-negatif yang bertujuan untuk menghitung biaya minimum dari satu simpul sumber ke semua simpul lainnya. Dijkstra bekerja dengan prinsip *greedy*, yaitu selalu memilih simpul dengan jarak sementara terkecil yang belum dikunjungi, lalu memperbarui jarak minimum ke simpul-simpul tetangganya jika ditemukan jalur yang lebih murah melalui simpul tersebut. Dalam implementasinya, Dijkstra menggunakan struktur data *priority queue* untuk memilih simpul dengan jarak terkecil secara efisien. Fungsi biaya yang digunakan adalah $g(n)$, yaitu total biaya aktual dari simpul sumber ke simpul n, tanpa menggunakan informasi heuristik apa pun. Karena itu, algoritma ini termasuk dalam kategori *uninformed search*.

Dijkstra menjamin menemukan solusi optimal asalkan semua bobot sisi dalam graf bernilai non-negatif. Meskipun serupa dengan Uniform Cost Search (UCS), perbedaan utamanya adalah Dijkstra menghitung jarak ke seluruh simpul (global), sedangkan UCS biasanya berhenti ketika simpul tujuan telah ditemukan (lebih lokal). Algoritma ini sangat berguna dalam aplikasi seperti pemetaan jaringan jalan, routing data

dalam jaringan komputer, dan optimisasi jalur logistik, terutama ketika tujuan tidak spesifik atau ketika semua jarak minimum perlu diketahui.

BAB 2

Algoritma Program

2.1 Algoritma Pathfinding

Program Rush Hour yang dikembangkan dalam tugas ini mengimplementasikan empat algoritma pencarian jalur (pathfinding), yaitu Uniform Cost Search (UCS), Greedy Best First Search (GBFS), A*, dan sebagai bonus, juga disertakan Dijkstra. Seluruh algoritma ini digunakan untuk menyelesaikan permainan Rush Hour, di mana tujuan utamanya adalah mencari urutan pergerakan mobil/motor pada papan sehingga primary piece (P) dapat keluar dari papan melalui pintu keluar (K) dengan cara yang paling efisien. Tiap algoritma menerapkan strategi pencarian berbeda: UCS dan Dijkstra menggunakan pencarian berbasis cost aktual ($g(n)$), GBFS mengandalkan prediksi heuristik ($h(n)$), sedangkan A* mengombinasikan keduanya dalam perhitungan total skor $f(n) = g(n) + h(n)$ untuk mendapatkan solusi optimal dengan eksplorasi yang efisien.

Dalam konteks pencarian jalur, fungsi biaya total $f(n)$ dan fungsi biaya lintasan sejauh ini $g(n)$ berperan penting dalam menilai seberapa menjajikan suatu simpul (node). $g(n)$ adalah total biaya dari simpul awal hingga simpul saat ini, sedangkan $f(n)$ merupakan estimasi total biaya dari simpul awal hingga simpul tujuan melalui simpul saat ini, yaitu $f(n) = g(n) + h(n)$, dengan $h(n)$ adalah fungsi heuristik. Pada algoritma A*, nilai $f(n)$ digunakan untuk menentukan prioritas eksplorasi state, sehingga memungkinkan kombinasi antara pencarian berdasarkan biaya nyata dan estimasi ke depan.

Heuristik yang digunakan dalam implementasi A* maupun GBFS di program ini adalah estimasi berbasis posisi dan penghalang di depan primary piece (P). Contohnya adalah jumlah kendaraan yang menghalangi jalur P ke pintu keluar atau jarak terpendek yang masih harus ditempuh. Jika heuristik yang digunakan tidak pernah melebihi cost sesungguhnya dari node ke goal, maka heuristik tersebut dikatakan admissible. Berdasarkan implementasi program, heuristik yang hanya menghitung jumlah blocker di depan P atau jarak langsung tanpa memperhitungkan langkah kompleks dapat dianggap admissible, karena nilainya tidak melebihi biaya aktual penyelesaian.

Pada kasus penyelesaian Rush Hour, UCS tidak sama dengan BFS, meskipun keduanya tidak menggunakan heuristik. BFS mengeksplorasi semua simpul pada level yang sama sebelum melanjutkan ke level berikutnya, sedangkan UCS mengeksplorasi berdasarkan biaya aktual ($g(n)$). Karena langkah di Rush Hour bisa memiliki cost berbeda jika langkah dinilai berdasarkan jumlah grid yang ditempuh, maka jalur yang dihasilkan UCS bisa berbeda dengan BFS. Namun, jika setiap langkah dianggap memiliki cost seragam, hasil path UCS dan BFS bisa identik secara struktur.

Secara teoritis, A* dapat lebih efisien dibanding UCS pada penyelesaian Rush Hour, karena A* menggunakan heuristik untuk mengarahkan pencarian ke arah goal, sehingga mempercepat eksplorasi dan mengurangi jumlah node yang dikunjungi. UCS, meskipun optimal, harus menjelajahi banyak simpul karena tidak memiliki panduan menuju goal. Dengan heuristik

yang baik dan admissible, A* akan menemukan solusi optimal dengan waktu yang lebih cepat dari UCS.

Sebaliknya, Greedy Best First Search (GBFS) hanya mempertimbangkan nilai heuristik $h(n)$ tanpa memperhatikan cost aktual ($g(n)$), sehingga meskipun GBFS bisa sangat cepat, ia tidak menjamin solusi optimal. GBFS bisa terjebak dalam solusi suboptimal (local optimum) apabila heuristik yang digunakan menyesatkan. Oleh karena itu, secara teoritis GBFS lebih cocok untuk kasus ketika kecepatan lebih penting daripada optimalitas solusi, namun tidak dapat diandalkan untuk menjamin solusi terbaik dalam penyelesaian Rush Hour.

2.2 Pseudocode Algoritma

2.2.1 Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search (UCS) yang diimplementasikan menjamin jalur solusi dengan total cost terkecil. Setiap langkah pergeseran satu petak dihitung sebagai cost = 1, sehingga UCS akan menelusuri semua kemungkinan gerakan secara bertahap berdasarkan total cost kumulatif (fungsi $g(n)$), dan selalu memilih konfigurasi papan (state) dengan $g(n)$ terkecil terlebih dahulu.

Algoritma dimulai dari sebuah GameState awal yang dibentuk dari objek Board hasil pembacaan konfigurasi input papan permainan. State awal ini dibungkus ke dalam objek Node, yang disimpan dalam sebuah PriorityQueue (open list) dengan prioritas berdasarkan total cost-nya. Kemudian, dalam setiap iterasi, UCS akan mengambil node dengan cost paling kecil, memeriksa apakah node tersebut sudah merupakan tujuan (yakni ketika primary piece bisa keluar papan melalui pintu keluar), dan jika belum, akan diekspansi ke semua state valid berikutnya (successor states). Setiap successor yang belum pernah dikunjungi akan dihitung total cost-nya dan ditambahkan kembali ke queue untuk diproses kemudian.

Selama proses pencarian, UCS menyimpan seluruh state yang sudah dikunjungi di dalam closedList untuk menghindari eksplorasi ulang terhadap konfigurasi yang sama. Dengan demikian, UCS menjelajahi ruang kemungkinan konfigurasi papan dengan cara yang sistematis dan menjamin bahwa solusi pertama yang ditemukan adalah solusi optimal (yakni dengan jumlah langkah minimum). Ketika solusi ditemukan, jalur solusi tersebut direkonstruksi dari state akhir hingga awal menggunakan informasi pergerakan yang tersimpan dalam setiap GameState. Berikut pseudocode dari algoritma UCS yang diimplementasikan,

```
function UCS(Board initialBoard) -> GameState: # Inisialisasi struktur data
  openList = PriorityQueue ordered by node.cost
  closedList = Empty Set of GameStates

  # Inisialisasi metrik pencarian
  nodesVisited = 0
  startTime = getCurrentTime()

  # Buat state awal dan simpul awal
  initialState = new GameState(initialBoard, "none")
  initialNode = new Node(initialState, 0, null)
  openList.add(initialNode)
```

```

# Perulangan utama pencarian
while openList is not empty:
    currentNode = openList.poll() # Ambil simpul dengan biaya terendah
    currentState = currentNode.state
    nodesVisited = nodesVisited + 1

    # Cek apakah tujuan tercapai
    if currentState.isGoal():
        executionTime = getCurrentTime() - startTime
        return currentState

    # Lewati jika state sudah dikunjungi
    if currentState in closedList:
        continue

    # Tandai state sebagai sudah dikunjungi
    closedList.add(currentState)

    # Hasilkann dan proses successors
    for each successor in currentState.getSuccessors():
        if successor not in closedList:
            successorNode = new Node(successor, successor.getG(), currentNode)
            openList.add(successorNode)

    # Tidak ada solusi yang ditemukan
executionTime = getCurrentTime() - startTime
return null

function printSolution(GameState solution, Board initialBoard): if solution is null:
print "Tidak ada solusi yang ditemukan!" return

moves = solution.getMoves()

print "Papan Awal:"
initialBoard.printBoard(null)

currentBoard = initialBoard
for i = 0 to moves.size() - 1:
    move = moves[i]
    currentBoard = currentBoard.applyMove(move)
    print "Gerakan " + (i + 1) + ":" + move
    currentBoard.printBoard(move)

print "Solusi ditemukan dalam " + moves.size() + " langkah"
print "Jumlah node yang diperiksa: " + nodesVisited
print "Waktu eksekusi: " + executionTime + " ms"

```

2.2.2 Algoritma Greedy Best First Search (GBFS)

Algoritma Greedy Best-First Search (GBFS) pada kelas GBFS digunakan untuk menyelesaikan permainan Rush Hour dengan pendekatan yang memprioritaskan

kedekatan terhadap tujuan berdasarkan nilai heuristik. Tujuan permainan ini adalah untuk mengeluarkan primary piece P dari papan menuju pintu keluar K, melalui serangkaian langkah minimum yang valid. Tidak seperti UCS yang mempertimbangkan total cost sejauh ini ($g(n)$), GBFS hanya mempertimbangkan seberapa dekat sebuah konfigurasi terhadap solusi menggunakan fungsi heuristik ($h(n)$), tanpa memperhitungkan langkah-langkah yang telah dilakukan sebelumnya. Hal ini membuat GBFS lebih cepat dalam eksplorasi, meskipun tidak selalu menghasilkan solusi optimal.

Proses pencarian dimulai dengan menginisialisasi GameState awal dari objek papan (Board) yang diberikan. Algoritma menggunakan PriorityQueue (open list) yang menyimpan node berdasarkan nilai heuristik $h(n)$ paling kecil. Pada setiap iterasi, algoritma memilih state dengan nilai heuristik terkecil dari queue dan memeriksa apakah konfigurasi tersebut merupakan tujuan. Jika tidak, maka semua successor valid dari state tersebut akan dihasilkan, dikonversi menjadi node baru, dan dimasukkan kembali ke queue jika belum pernah dikunjungi sebelumnya (dicek menggunakan representasi String papan sebagai key dalam closedList).

Heuristik yang digunakan dipilih melalui parameter heuristicName, dan nilai $h(n)$ dihitung oleh masing-masing state GameState. Implementasi ini mendukung fleksibilitas untuk mencoba beberapa fungsi heuristik berbeda yang relevan untuk Rush Hour (misalnya jumlah mobil penghalang di depan P, atau jarak bebas ke pintu keluar). Berikut pseudocode dari algoritma GBFS yang diimplementasikan,

```
function GBFS(Board initialBoard, String heuristicName) -> GameState: # Inisialisasi struktur data openList = PriorityQueue ordered by node.state.getH() closedList = Empty Set of Strings

# Inisialisasi metrik pencarian
nodesVisited = 0
startTime = getCurrentTime()

# Buat state awal dan simpul awal
initialState = new GameState(initialBoard, heuristicName)
initialNode = new Node(initialState, null)
openList.add(initialNode)

# Perulangan utama pencarian
while openList is not empty:
    currentNode = openList.poll() # Ambil simpul dengan nilai heuristik terendah
    currentState = currentNode.state
    nodesVisited = nodesVisited + 1

    # Buat kunci unik untuk papan saat ini
    currentKey = getBoardKey(currentState.getBoard())

    # Cek apakah tujuan tercapai
    if currentState.isGoal():
        executionTime = getCurrentTime() - startTime
        return currentState

    # Lewati jika papan sudah dikunjungi
```

```

if currentKey in closedList:
    continue

# Tandai papan sebagai sudah dikunjungi
closedList.add(currentKey)

# Hasilkan dan proses successors
for each successor in currentState.getSuccessors():
    successorKey = getBoardKey(successor.getBoard())
    if successorKey not in closedList:
        successorNode = new Node(successor, currentNode)
        openList.add(successorNode)

# Tidak ada solusi yang ditemukan
executionTime = getCurrentTime() - startTime
return null

function getBoardKey(Board board) -> String: key = "" grid = board.getGrid() for i =
0 to board.getRows() - 1: for j = 0 to board.getCols() - 1: key = key + grid[i][j]
return key

function printSolution(GameState solution, Board initialBoard, String heuristicName):
if solution is null: print "Tidak ada solusi yang ditemukan!" return

print "Menggunakan heuristic: " + heuristicName

moves = solution.getMoves()

print "Papan Awal:"
initialBoard.printBoard(null)

currentBoard = initialBoard
for i = 0 to moves.size() - 1:
    move = moves[i]
    currentBoard = currentBoard.applyMove(move)
    print "Gerakan " + (i + 1) + ":" + move
    currentBoard.printBoard(move)

print "Solusi ditemukan dalam " + moves.size() + " langkah"
print "Jumlah node yang diperiksa: " + nodesVisited
print "Waktu eksekusi: " + executionTime + " ms"

```

2.2.3 Algoritma A*

Algoritma A* (A-Star) adalah salah satu algoritma pencarian jalur yang paling cerdas dan efisien karena menggabungkan kekuatan dari dua pendekatan: Uniform Cost Search (berbasis cost sejauh ini) dan Greedy Best-First Search (berbasis prediksi ke tujuan). Pada implementasi di kelas AStar, permainan Rush Hour diperlakukan sebagai ruang state, di mana setiap konfigurasi papan merupakan sebuah state, dan gerakan valid dari mobil/motor menghasilkan state baru. Tujuan algoritma adalah menemukan urutan

langkah minimal yang dapat mengeluarkan primary piece (P) menuju pintu keluar (K) dengan menggunakan kombinasi dari biaya aktual ($g(n)$) dan nilai heuristik ($h(n)$) untuk menilai setiap state.

Algoritma memulai pencarian dari state awal yang dibentuk dari objek Board, kemudian memasukkan state tersebut ke dalam sebuah priority queue (open set) yang memprioritaskan elemen berdasarkan total skor $f(n) = g(n) + h(n)$. Di setiap iterasi, A* akan mengambil state dengan nilai $f(n)$ terkecil dari open set dan memeriksa apakah state tersebut sudah mencapai kondisi tujuan (`isGoal()`). Jika belum, semua state turunan (successors) dari state tersebut akan dihitung nilai $g(n)$ -nya (jumlah langkah dari awal), kemudian dibandingkan dengan skor terbaik sebelumnya (`bestGScore`). Jika ditemukan jalur baru yang lebih baik menuju state tersebut, maka A* akan memperbarui datanya dan memasukkan state baru ke open set.

State yang telah diproses akan dimasukkan ke dalam closed set berdasarkan representasi string unik dari papan (`getBoardKey()`), sehingga tidak akan dikunjungi ulang. Dengan pendekatan ini, A* mengeksplorasi hanya jalur-jalur yang menjanjikan, sekaligus memastikan solusi yang ditemukan adalah optimal — yakni dengan jumlah langkah minimum berdasarkan kombinasi cost sejauh ini dan prediksi ke depan. Berikut pseudocode dari algoritma A* yang diimplementasikan,

```
function AStar(Board initialBoard, String heuristicName) -> GameState:
    # Inisialisasi metrik pencarian
    nodesVisited = 0
    startTime = getCurrentTime()

    # Inisialisasi struktur data
    openSet = PriorityQueue ordered by state.getF()
    closedSet = Empty Set of Strings
    bestGScore = Empty Map of <String, Double>

    # Buat state awal
    startState = new GameState(initialBoard, heuristicName)
    openSet.add(startState)

    # Simpan skor G terbaik untuk papan awal
    startStateKey = getBoardKey(startState.getBoard())
    bestGScore.put(startStateKey, 0.0)

    # Perulangan utama pencarian
    while openSet is not empty:
        current = openSet.poll() # Ambil state dengan nilai f terendah
        nodesVisited = nodesVisited + 1

        # Cek apakah tujuan tercapai
        if current.isGoal():
            executionTime = getCurrentTime() - startTime
            return current

        # Buat kunci unik untuk papan saat ini
        currentKey = getBoardKey(current.getBoard())
```

```

# Lewati jika papan sudah dikunjungi
if currentKey in closedSet:
    continue

# Tandai papan sebagai sudah dikunjungi
closedSet.add(currentKey)

# Hasilkan dan proses successors
successors = current.getSuccessors()
for each successor in successors:
    successorKey = getBoardKey(successor.getBoard())

    # Lewati jika successor sudah dikunjungi
    if successorKey in closedSet:
        continue

    # Hitung skor G baru
    tentativeG = successor.getG()

    # Jika ditemukan jalur yang lebih baik
    if successorKey not in bestGScore OR tentativeG <
bestGScore.get(successorKey):
        bestGScore.put(successorKey, tentativeG)

    # Tambahkan ke openSet jika belum ada
    if not isInOpenSet(openSet, successorKey):
        openSet.add(successor)

# Tidak ada solusi yang ditemukan
executionTime = getCurrentTime() - startTime
return null

function isInOpenSet(PriorityQueue openSet, String key) -> boolean:
    for each state in openSet:
        if getBoardKey(state.getBoard()) equals key:
            return true
    return false

function getBoardKey(Board board) -> String:
    key = ""
    grid = board.getGrid()
    for i = 0 to board.getRows() - 1:
        for j = 0 to board.getCols() - 1:
            key = key + grid[i][j]
    return key

function printSolution(GameState solution, Board initialBoard, String heuristicName):
    if solution is null:
        print "Tidak ada solusi yang ditemukan!"
        return

    print "Menggunakan heuristic: " + heuristicName

```

```

moves = solution.getMoves()

print "Papan Awal"
initialBoard.printBoard(null)

currentBoard = initialBoard
for i = 0 to moves.size() - 1:
    move = moves[i]
    currentBoard = currentBoard.applyMove(move)
    print "Gerakan " + (i + 1) + ": " + move
    currentBoard.printBoard(move)

print "Solusi ditemukan dalam " + moves.size() + " langkah"
print "Jumlah node yang diperiksa: " + nodesVisited
print "Waktu eksekusi: " + executionTime + " ms"

```

2.2.4 Algoritma Dijkstra

Algoritma Dijkstra merupakan algoritma pencarian jalur yang berfokus untuk menemukan rute dengan total biaya minimum dari titik awal menuju titik tujuan, tanpa menggunakan heuristik. Dalam konteks permainan Rush Hour, algoritma ini digunakan untuk mencari solusi berupa urutan langkah pergerakan mobil sehingga primary piece P dapat keluar dari papan melalui pintu keluar K. Implementasi algoritma ini bekerja dengan cara menjelajahi ruang state, di mana setiap konfigurasi papan adalah sebuah GameState, dan cost $g(n)$ menyatakan jumlah langkah yang telah ditempuh dari konfigurasi awal.

Pencarian dimulai dari state awal (GameState awal yang dibentuk dari Board), yang dimasukkan ke dalam sebuah priority queue (openSet) berdasarkan nilai $g(n)$ — yakni jumlah langkah dari awal. Setiap iterasi, state dengan $g(n)$ terkecil diambil dari queue. Jika state tersebut merupakan solusi (diperiksa dengan `isGoal()`), maka pencarian dihentikan dan solusi dikembalikan. Jika belum, semua successor valid dari state saat ini akan diperiksa, dan jika ditemukan cost yang lebih baik menuju state tersebut (lebih kecil dari sebelumnya), maka state baru tersebut akan dimasukkan ke openSet, dan bestCost akan diperbarui.

Setiap state yang telah diproses akan disimpan di dalam closedSet untuk menghindari eksplorasi ulang. Identitas unik setiap state ditentukan dari representasi string papan (`getBoardKey()`), yang menggabungkan seluruh karakter papan menjadi satu string sebagai kunci. Jika sebuah state ditemukan di openSet tetapi memiliki jalur baru dengan cost lebih baik, maka queue akan diperbarui dengan versi baru state tersebut melalui metode `updateOpenSet()`. Berikut pseudocode dari algoritma Djikstra yang diimplementasikan,

```

function Dijkstra(Board initialBoard) -> GameState:
    # Inisialisasi metrik pencarian
    nodesVisited = 0
    startTime = getCurrentTime()

```

```

# Inisialisasi struktur data
openSet = PriorityQueue ordered by state.getG()
closedSet = Empty Set of Strings
bestCost = Empty Map of <String, Double>

# Buat state awal
startState = new GameState(initialBoard)
openSet.add(startState)

# Simpan biaya terbaik untuk papan awal
startStateKey = getBoardKey(startState.getBoard())
bestCost.put(startStateKey, 0.0)

# Perulangan utama pencarian
while openSet is not empty:
    current = openSet.poll() # Ambil state dengan nilai g terendah
    nodesVisited = nodesVisited + 1

    # Buat kunci unik untuk papan saat ini
    currentKey = getBoardKey(current.getBoard())

    # Cek apakah tujuan tercapai
    if current.isGoal():
        executionTime = getCurrentTime() - startTime
        return current

    # Lewati jika papan sudah dikunjungi
    if currentKey in closedSet:
        continue

    # Tandai papan sebagai sudah dikunjungi
    closedSet.add(currentKey)

    # Hasilkan dan proses successors
    successors = current.getSuccessors()
    for each successor in successors:
        successorKey = getBoardKey(successor.getBoard())

        # Lewati jika successor sudah dikunjungi
        if successorKey in closedSet:
            continue

        # Hitung biaya baru
        newCost = successor.getG()

        # Jika ditemukan jalur yang lebih baik
        if successorKey not in bestCost OR newCost < bestCost.get(successorKey):
            bestCost.put(successorKey, newCost)

        # Jika belum ada di openSet, tambahkan ke openSet
        if not isInOpenSet(openSet, successorKey):
            openSet.add(successor)

```

```

        # Jika sudah ada di openSet, update dengan yang lebih baik
        else:
            updateOpenSet(openSet, successor, successorKey)

    # Tidak ada solusi yang ditemukan
    executionTime = getCurrentTime() - startTime
    return null

function isInOpenSet(PriorityQueue openSet, String key) -> boolean:
    for each state in openSet:
        if getBoardKey(state.getBoard()) equals key:
            return true
    return false

function updateOpenSet(PriorityQueue openSet, GameState newState, String key):
    tempStates = new ArrayList()

    # Keluarkan semua state dari openSet
    while openSet is not empty:
        state = openSet.poll()
        # Simpan semua state kecuali yang ingin diupdate
        if not getBoardKey(state.getBoard()) equals key:
            tempStates.add(state)

    # Kembalikan semua state yang disimpan ke openSet
    openSet.addAll(tempStates)
    #Tambahkan state baru dengan biaya yang lebih baik
    openSet.add(newState)

function getBoardKey(Board board) -> String:
    key = ""
    grid = board.getGrid()
    for i = 0 to board.getRows() - 1:
        for j = 0 to board.getCols() - 1:
            key = key + grid[i][j]
    return key

function printSolution(GameState solution, Board initialBoard):
    if solution is null:
        print "No solution found!"
        return

    moves = solution.getMoves()

    print "Papan Awal"
    initialBoard.printBoard(null)

    currentBoard = initialBoard
    for i = 0 to moves.size() - 1:
        move = moves[i]
        currentBoard = currentBoard.applyMove(move)
        print "Gerakan " + (i + 1) + ":" + move

```

```
currentBoard.printBoard(move)

print "Solusi ditemukan dalam " + moves.size() + " langkah"
print "Jumlah node yang diperiksa: " + nodesVisited
print "Waktu eksekusi: " + executionTime + " ms"
```

BAB 3

Source Code Program

The screenshot shows a file explorer window with the following folder structure:

- EXPLORER ...
- └ TUCIL3_13523071_13523097
 - └ .vscode
 - {} launch.json
 - > bin
 - > doc
 - > lib
 - > resources
 - └ src
 - └ algorithm
 - J AStar.java
 - J Dijkstra.java
 - J GBFS.java
 - J Heuristics.java
 - J UCS.java
 - └ core
 - J Board.java
 - J GameState.java
 - J Move.java
 - J Piece.java
 - └ gui
 - J AboutPage.java
 - J App.java
 - J BoardPane.java
 - J ControlPanel.java
 - J CreatorPage.java
 - J Renderer.java
 - # style.css
 - J WelcomePage.java
 - J Main.java
 - > test
 - ↳ .gitignore
 - M Makefile
 - I README.md

Gambar 6. Folder Structure
(Sumber: Penulis)

3.1 Board.java

Berikut ini merupakan source code beserta tabel attribute dan method yang terdapat dalam struct di Board.java.

Board.java

```
package core;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Board {
    private int rows;
    private int cols;
    private char[][] grid;
    private List<Piece> pieces;
    private Piece primaryPiece;
    private int exitRow;
    private int exitCol;

    public Board(String filename) throws IOException {
        pieces = new ArrayList<>();
        exitRow = -1;
        exitCol = -1;
        readBoardFromFile(filename);
    }

    public Board(int rows, int cols, char[][] grid, List<Piece> pieces, Piece
primaryPiece, int exitRow, int exitCol) {
        this.rows = rows;
        this.cols = cols;
        this.grid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            System.arraycopy(grid[i], 0, this.grid[i], 0, cols);
        }
    }
}
```

```
    }

    this.pieces = new ArrayList<>();
    for (Piece p : pieces) {
        this.pieces.add(new Piece(p.getId(), p.getRow(), p.getCol(),
p.getSize(), p.isHorizontal(), p.isPrimary()));
    }
    this.primaryPiece = new Piece(primaryPiece.getId(),
primaryPiece.getRow(), primaryPiece.getCol(),
primaryPiece.getSize(), primaryPiece.isHorizontal(),
primaryPiece.isPrimary());
    this.exitRow = exitRow;
    this.exitCol = exitCol;
}

private void readBoardFromFile(String filename) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    String[] dimensions = reader.readLine().trim().split("\\s+");
    rows = Integer.parseInt(dimensions[0]);
    cols = Integer.parseInt(dimensions[1]);
    grid = new char[rows][cols];

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

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

    int kCount = 0;

    List<String> allLines = new ArrayList<>();
    String line;
    while ((line = reader.readLine()) != null) {
        allLines.add(line);
        for (int i = 0; i < line.length(); i++) {
            if (line.charAt(i) == 'K') {
                kCount++;
            }
        }
    }
}
```

```
reader.close();
reader = new BufferedReader(new FileReader(filename));

reader.readLine();
reader.readLine();

for (int i = 0; i < rows; i++) {
    line = i < allLines.size() ? allLines.get(i) : "";
    line = line.trim();

    if (line.length() > cols && line.charAt(cols) == 'K') {
        exitRow = i;
        exitCol = cols;
        System.out
            .println("Debug: Found exit at row " + exitRow + ", col
" + exitCol + " (outside right grid)");
    }

    for (int j = 0; j < cols; j++) {
        char c = j < line.length() ? line.charAt(j) : '.';
        if (c == 'K') {
            exitRow = i;
            exitCol = j;
            grid[i][j] = 'K';
            System.out.println("Debug: Found exit at row " + exitRow +
", col " + exitCol + " (inside grid)");
        } else {
            grid[i][j] = c;
            if (c != '.') {
                pieceCells.computeIfAbsent(c, k -> new
ArrayList<>()).add(new int[] { i, j });
            }
        }
    }
}

if (rows < allLines.size()) {
    line = allLines.get(rows);
```

```
        int kIndex = line.indexOf('K');
        if (kIndex != -1) {
            exitRow = rows;
            exitCol = kIndex;
            System.out.println("Debug: Found exit at row " + exitRow + ", "
col " + exitCol + " (below grid)");
        }
    }

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

        int size = cells.size();
        int minRow = cells.get(0)[0];
        int minCol = cells.get(0)[1];
        boolean isHorizontal = cells.get(0)[0] == cells.get(1)[0];
        for (int[] cell : cells) {
            minRow = Math.min(minRow, cell[0]);
            minCol = Math.min(minCol, cell[1]);
        }
        boolean isPrimary = (id == 'P');
        Piece piece = new Piece(id, minRow, minCol, size, isHorizontal,
isPrimary);
        pieces.add(piece);
        if (isPrimary) {
            primaryPiece = piece;
        }
    }

    if (primaryPiece == null) {
        throw new IllegalArgumentException("No primary piece found");
    }

    int actualNonPrimary = pieces.size() - 1;
    if (actualNonPrimary != numNonPrimaryPieces) {
        throw new IllegalArgumentException(
            "Expected " + numNonPrimaryPieces + " non-primary pieces,"
```

```
found " + actualNonPrimary);
}

    System.out.println("Debug: Primary piece at col " +
primaryPiece.getCol() + ", row " + primaryPiece.getRow() +
        ", isHorizontal: " + primaryPiece.isHorizontal());
    System.out.println("Debug: Exit at col " + exitCol + ", row " +
exitRow);

    if (primaryPiece.isHorizontal()) {
        if (exitRow != primaryPiece.getRow()) {
            System.out.println("Debug: Exit and primary piece not aligned
horizontally");
            System.out.println("Debug: Exit row: " + exitRow + ", Primary
piece row: " + primaryPiece.getRow());
            throw new IllegalArgumentException("Exit not aligned with
horizontal primary piece");
        }
    } else {
        if (exitCol != primaryPiece.getCol()) {
            System.out.println("Debug: Exit and primary piece not aligned
vertically");
            System.out.println("Debug: Exit col: " + exitCol + ", Primary
piece col: " + primaryPiece.getCol());
            throw new IllegalArgumentException("Exit not aligned with
vertical primary piece");
        }
    }
}

public int getRows() {
    return rows;
}

public int getCols() {
    return cols;
}
```

```
public Piece getPrimaryPiece() {
    return primaryPiece;
}

public int getExitRow() {
    return exitRow;
}

public int getExitCol() {
    return exitCol;
}

public boolean isEmpty(int row, int col) {
    if (row < 0 || row >= rows || col < 0 || col >= cols) {
        return false;
    }
    return grid[row][col] == '.' || grid[row][col] == 'K';
}

public boolean isSolved() {
    for (int[] cell : primaryPiece.getOccupiedCells()) {
        if (cell[0] == exitRow && cell[1] == exitCol) {
            return true;
        }
    }

    int pRow = primaryPiece.getRow();
    int pCol = primaryPiece.getCol();
    int pSize = primaryPiece.getSize();

    if (primaryPiece.isHorizontal() && exitCol == cols) {
        return pRow == exitRow && pCol + pSize == cols;
    }

    if (!primaryPiece.isHorizontal() && exitRow == rows) {
        return pCol == exitCol && pRow + pSize == rows;
    }
}
```

```
        return false;
    }

    public List<Move> getAllPossibleMoves() {
        List<Move> moves = new ArrayList<>();
        for (Piece piece : pieces) {
            for (Move move : piece.getPossibleMoves(this)) {
                moves.add(move);
            }
        }
        return moves;
    }

    public Board applyMove(Move move) {
        char pieceId = move.getPieceId();
        Piece oldPiece = null;
        for (Piece p : pieces) {
            if (p.getId() == pieceId) {
                oldPiece = p;
                break;
            }
        }
        if (oldPiece == null) {
            throw new IllegalArgumentException("Piece not found: " + pieceId);
        }

        Piece newPiece = oldPiece.applyMove(move);
        List<Piece> newPieces = new ArrayList<>();
        for (Piece p : pieces) {
            if (p.getId() == pieceId) {
                newPieces.add(newPiece);
            } else {
                newPieces.add(p);
            }
        }

        char[][] newGrid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
```

```
        System.arraycopy(grid[i], 0, newGrid[i], 0, cols);
    }

    for (int[] cell : oldPiece.getOccupiedCells()) {
        if (cell[0] >= 0 && cell[0] < rows && cell[1] >= 0 && cell[1] <
cols) {
            newGrid[cell[0]][cell[1]] = '.';
        }
    }

    for (int[] cell : newPiece.getOccupiedCells()) {
        if (cell[0] >= 0 && cell[0] < rows && cell[1] >= 0 && cell[1] <
cols) {
            newGrid[cell[0]][cell[1]] = newPiece.getId();
        }
    }

    Piece newPrimaryPiece = primaryPiece.getId() == pieceId ? newPiece :
primaryPiece;
    return new Board(rows, cols, newGrid, newPieces, newPrimaryPiece,
exitRow, exitCol);
}

public void printBoard(Move move) {
    char movedPieceId = move != null ? move.getId() : 0;
    System.out.println("Papan:");

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            char c = grid[i][j];
            if (c == 'P') {
                System.out.print("\u001B[31m" + c + "\u001B[0m ");
            } else if (c == movedPieceId && c != '.') {
                System.out.print("\u001B[34m" + c + "\u001B[0m ");
            } else {
                System.out.print(c + " ");
            }
        }
    }
}
```

```
        if (i == exitRow && exitCol == cols) {
            System.out.print("\u001B[32mK\u001B[0m");
        }

        System.out.println();
    }

    System.out.println();
}

public char[][] getGrid() {
    char[][] copy = new char[rows][cols];
    for (int i = 0; i < rows; i++) {
        System.arraycopy(grid[i], 0, copy[i], 0, cols);
    }
    return copy;
}

public List<Piece> getPieces() {
    List<Piece> copy = new ArrayList<>();
    for (Piece p : pieces) {
        copy.add(new Piece(p.getId(), p.getRow(), p.getCol(), p.getSize(),
p.isHorizontal(), p.isPrimary())));
    }
    return copy;
}

public Board reverseMove(Move move) {
    char pieceId = move.getPieceId();
    String direction = move.getDirection();

    String oppositeDirection;
    switch (direction) {
        case "up":
            oppositeDirection = "down";
            break;
        case "down":
```

```

        oppositeDirection = "up";
        break;
    case "left":
        oppositeDirection = "right";
        break;
    case "right":
        oppositeDirection = "left";
        break;
    default:
        throw new IllegalArgumentException("Unknown direction: " +
direction);
    }

    Move reverseMove = new Move(pieceId, oppositeDirection);

    return applyMove(reverseMove);
}
}

```

Attribute	Tipe Data	Keterangan
rows	private int	Atribut baris
cols	private int	Atribut kolom
grid	private char[][]	Atribut matriks untuk merepresentasikan isi papan
pieces	private List<Piece>	Atribut daftar objek Piece yang ada pada papan
primaryPiece	private Piece	Atribut <i>piece</i> utama
exitRow	private int	Atribut indeks baris dari lokasi pintu keluar K
exitCol	private int	Atribut indeks kolom dari lokasi pintu keluar K

Method	Returned Data Type	Keterangan
--------	--------------------	------------

Board	-	Konstruktor untuk membaca papan dari file konfigurasi
Board	-	Konstruktor dengan parameter
readBoardFromFile	-	Metode untuk memuat informasi papan, kendaraan, dan pintu keluar dari file
getRows	int	Metode untuk mengembalikan jumlah baris papan
getCols	int	Metode untuk mengembalikan jumlah kolom papan
getPrimaryPiece	Piece	Metode untuk mengembalikan objek Piece utama
getExitRow	int	Metode untuk mengembalikan indeks baris pintu keluar K
getExitCol	int	Metode untuk mengembalikan indeks kolom pintu keluar K
isCellEmpty	boolean	Metode untuk mengecek apakah sel pada baris dan kolom tertentu kosong atau tidak
isSolved	boolean	Metode untuk mengecek apakah primary piece sudah berada di pintu keluar (selesai)
getAllPossibleMoves	List<Move>	Metode untuk menghasilkan semua gerakan valid dari setiap kendaraan pada papan
applyMove	Board	Metode untuk mengembalikan papan setelah satu Move diterapkan pada papan

printBoard	-	Metode untuk menampilkan papan
getGrid	char[][]	Metode untuk mengembalikan salinan 2D array papan saat ini
getPieces	List<Piece>	Metode untuk mengembalikan salinan daftar semua kendaraan (Piece) di papan
reverseMove	Board	Metode untuk mengembalikan papan baru setelah membalikkan arah dari suatu Move

3.2 GameState.java

Berikut ini merupakan source code beserta tabel attribute dan method yang terdapat dalam class di GameState.java

GameState.java

```
package core;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import algorithm.Heuristics;

public class GameState {
    private final Board board;
    private final List<Move> moves;
    private final double g;
    private final double h;
    private final double f;
    private final String heuristicName;

    public GameState(Board board) {
        this(board, "manhattan");
    }
}
```

```

public GameState(Board board, String heuristicName) {
    this.board = board;
    this.moves = new ArrayList<>();
    this.g = 0;
    this.heuristicName = heuristicName;
    this.h = heuristicName.equals("none") ? 0 : computeHeuristic();
    this.f = g + h;
}

public GameState(Board board, List<Move> parentMoves, Move newMove, double parentG, boolean isUCS) {
    this.board = board;
    this.moves = new ArrayList<>(parentMoves);
    this.moves.add(newMove);
    this.g = parentG + 1;
    this.heuristicName = "none";
    this.h = isUCS ? 0 : computeHeuristic();
    this.f = g + h;
}

public GameState(Board board, List<Move> parentMoves, Move newMove, double parentG) {
    this(board, parentMoves, newMove, parentG, "manhattan");
}

public GameState(Board board, List<Move> parentMoves, Move newMove, double parentG, String heuristicName) {
    this.board = board;
    this.moves = new ArrayList<>(parentMoves);
    this.moves.add(newMove);
    this.g = parentG + 1;
    this.heuristicName = heuristicName;
    this.h = heuristicName.equals("none") ? 0 : computeHeuristic();
    this.f = g + h;
}

private double computeHeuristic() {

```

```
        if (heuristicName == null || heuristicName.equals("none")) {
            return 0;
        } else if (heuristicName.equals("manhattan")) {
            Piece primary = board.getPrimaryPiece();
            int exitRow = board.getExitRow();
            int exitCol = board.getExitCol();

            if (primary.isHorizontal()) {
                int pieceEndCol = primary.getCol() + primary.getSize() - 1;
                return Math.abs(pieceEndCol - exitCol);
            } else {
                int pieceEndRow = primary.getRow() + primary.getSize() - 1;
                return Math.abs(pieceEndRow - exitRow);
            }
        } else {
            return Heuristics.getHeuristic(board, heuristicName);
        }
    }

    // Get successor states
    public List<GameState> getSuccessors() {
        List<GameState> successors = new ArrayList<>();
        for (Move move : board.getAllPossibleMoves()) {
            Board newBoard = board.applyMove(move);
            boolean isUCS = heuristicName.equals("none");
            if (isUCS) {
                GameState successor = new GameState(newBoard, moves, move, g,
true);
                successors.add(successor);
            } else {
                GameState successor = new GameState(newBoard, moves, move, g,
heuristicName);
                successors.add(successor);
            }
        }
        return successors;
    }
}
```

```
public Board getBoard() {
    return board;
}

public List<Move> getMoves() {
    return Collections.unmodifiableList(moves);
}

public double getG() {
    return g;
}

public double getH() {
    return h;
}

public double getF() {
    return f;
}

public String getHeuristicName() {
    return heuristicName;
}

public boolean isGoal() {
    return board.isSolved();
}

@Override
public boolean equals(Object o) {
    if (this == o)
        return true;
    if (!(o instanceof GameState))
        return false;
    GameState other = (GameState) o;
    char[][] thisGrid = board.getGrid();
    char[][] otherGrid = other.board.getGrid();
    for (int i = 0; i < board.getRows(); i++) {
```

```

        for (int j = 0; j < board.getCols(); j++) {
            if (thisGrid[i][j] != otherGrid[i][j]) {
                return false;
            }
        }
    }
    return true;
}

@Override
public int hashCode() {
    char[][] grid = board.getGrid();
    int hash = 0;
    for (int i = 0; i < board.getRows(); i++) {
        for (int j = 0; j < board.getCols(); j++) {
            hash = 31 * hash + grid[i][j];
        }
    }
    return hash;
}
}

```

Attribute	Tipe Data	Keterangan
board	private final Board	Atribut papan
moves	private final List<Move>	Atribut daftar Move
g	private final double	Atribut g(n)
h	private final double	Atribut h(n)
f	private final double	Atribut f(n) = g(n) + h(n)
heuristicName	private final String	Atribut nama heuristic yang digunakan

Method	Returned Data Type	Keterangan
GameState	-	Konstruktor default dengan

		heuristic "manhattan" dan tanpa langkah ($g = 0$, moves kosong)
GameState	-	Konstruktor dengan papan dan heuristic yang ditentukan oleh pengguna
GameState	-	Konstruktor untuk UCS; membuat GameState baru dengan langkah sebelumnya + 1 dan $h = 0$
GameState	-	Konstruktor untuk A* dengan heuristic default "manhattan"
GameState	-	Konstruktor untuk A* dengan heuristic tertentu; digunakan dalam GBFS dan A*
computeHeuristic	double	Metode internal untuk menghitung nilai $h(n)$ berdasarkan nama heuristic
getSuccessors	List<GameState>	Metode untuk mengembalikan successors
getBoard	Board	Metode untuk mengembalikan papan
getMoves	List<Move>	Metode untuk mengembalikan daftar moves
getG	double	Metode untuk mengembalikan $g(n)$
getH	double	Metode untuk mengembalikan $h(n)$
getF	double	Metode untuk mengembalikan $f(n)$
getHeuristicName	String	Metode untuk mengembalikan nama heuristic
isGoal	boolean	Metode untuk mengecek

		apakah papan atau permainan sudah diselesaikan
equals	boolean	Override untuk membandingkan dua GameState berdasarkan konfigurasi papan
hashCode	int	Override untuk menghitung hash berdasarkan isi papan untuk struktur hash

3.3 Move.java

Berikut ini merupakan source code beserta tabel attribute dan method yang terdapat dalam class di Move.java

```
package core;

public class Move {
    private char pieceId;
    private String direction;
    private int steps;

    public Move(char pieceId, String direction) {
        this(pieceId, direction, 1);
    }

    public Move(char pieceId, String direction, int steps) {
        if (!isValidDirection(direction)) {
            throw new IllegalArgumentException("Invalid direction: " +
direction);
        }
        if (steps < 1) {
            throw new IllegalArgumentException("Steps must be at least 1");
        }
        this.pieceId = pieceId;
        this.direction = direction;
        this.steps = steps;
    }
}
```

```
public Move(String moveString) {
    String[] parts = moveString.split("-");
    if (parts.length < 2 || parts.length > 3 || parts[0].length() != 1) {
        throw new IllegalArgumentException("Invalid move format: " +
moveString);
    }

    this.pieceId = parts[0].charAt(0);
    this.direction = parts[1];

    if (!isValidDirection(direction)) {
        throw new IllegalArgumentException("Invalid direction in move: " +
direction);
    }

    this.steps = (parts.length == 3) ? Integer.parseInt(parts[2]) : 1;

    if (steps < 1) {
        throw new IllegalArgumentException("Steps must be at least 1");
    }
}

private boolean isValidDirection(String direction) {
    return direction.equals("kiri") || direction.equals("kanan") ||
    direction.equals("atas") || direction.equals("bawah");
}

public char getPieceId() {
    return pieceId;
}

public String getDirection() {
    return direction;
}

public int getSteps() {
    return steps;
}
```

```

@Override
public String toString() {
    if (steps == 1) {
        return pieceId + "-" + direction;
    } else {
        return pieceId + "-" + direction + "-" + steps;
    }
}

```

Attribute	Tipe Data	Keterangan
pieceId	private char	Atribut ID Piece
direction	private String	Atribut yang merepresentasikan arah gerak Piece
steps	private int	Atribut jumlah langkah

Method	Returned Data Type	Keterangan
Move	-	Konstruktor untuk membuat objek Move dengan pieceId dan arah default 1 langkah
Move	-	Konstruktor untuk membuat Move dengan pieceId, arah, dan jumlah langkah
Move	-	Konstruktor untuk parsing string format "A-kiri-2" menjadi objek Move
isValidDirection	boolean	Metode untuk mengecek apakah arah (direction) valid (hanya "atas", "bawah", "kiri", "kanan")
getPieceId	char	Metode untuk mengembalikan ID Piece

getDirection	String	Metode untuk mengembalikan arah gerak Piece
getSteps	int	Metode untuk mengembalikan jumlah langkah gerakan
toString	String	Metode untuk mengubah objek Move menjadi string format A-kiri atau A-kiri-2

3.4 Piece.java

Berikut ini merupakan source code beserta tabel attribute dan method yang terdapat dalam class di Piece.java

```
package core;

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

public class Piece {
    private char id;
    private int row;
    private int col;
    private int size;
    private boolean isHorizontal;
    private boolean isPrimary;

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

    public char getId() {
```

```
        return id;
    }

    public int getRow() {
        return row;
    }

    public int getCol() {
        return col;
    }

    public int getSize() {
        return size;
    }

    public boolean isHorizontal() {
        return isHorizontal;
    }

    public boolean isPrimary() {
        return isPrimary;
    }

    public Move[] getPossibleMoves(Board board) {
        List<Move> moves = new ArrayList<>();

        if (isHorizontal) {
            int maxStepsLeft = 0;
            for (int step = 1; col - step >= 0; step++) {
                if (board.isCellEmpty(row, col - step)) {
                    maxStepsLeft++;
                } else {
                    break;
                }
            }

            for (int step = 1; step <= maxStepsLeft; step++) {
                moves.add(new Move(id, "kiri", step));
            }
        }
    }
}
```

```
    }

    int maxStepsRight = 0;
    for (int step = 1; col + size - 1 + step < board.getCols(); step++) {
        if (board.isEmpty(row, col + size - 1 + step)) {
            maxStepsRight++;
        } else {
            break;
        }
    }

    for (int step = 1; step <= maxStepsRight; step++) {
        moves.add(new Move(id, "kanan", step));
    }

} else {
    int maxStepsUp = 0;
    for (int step = 1; row - step >= 0; step++) {
        if (board.isEmpty(row - step, col)) {
            maxStepsUp++;
        } else {
            break;
        }
    }

    for (int step = 1; step <= maxStepsUp; step++) {
        moves.add(new Move(id, "atas", step));
    }

    int maxStepsDown = 0;
    for (int step = 1; row + size - 1 + step < board.getRows(); step++) {
        if (board.isEmpty(row + size - 1 + step, col)) {
            maxStepsDown++;
        } else {
            break;
        }
    }
}
```

```
    }

    for (int step = 1; step <= maxStepsDown; step++) {
        moves.add(new Move(id, "bawah", step));
    }
}

return moves.toArray(new Move[0]);
}

public Piece applyMove(Move move) {
    String direction = move.getDirection();
    int steps = move.getSteps();
    int newRow = row;
    int newCol = col;

    if (isHorizontal) {
        if (direction.equals("kiri")) {
            newCol -= steps;
        } else if (direction.equals("kanan")) {
            newCol += steps;
        }
    } else {
        if (direction.equals("atas")) {
            newRow -= steps;
        } else if (direction.equals("bawah")) {
            newRow += steps;
        }
    }
}

return new Piece(id, newRow, newCol, size, isHorizontal, isPrimary);
}

public int[][] getOccupiedCells() {
    int[][] cells = new int[size][2];
    for (int i = 0; i < size; i++) {
        if (isHorizontal) {
            cells[i][0] = row;
        }
    }
}
```

```

        cells[i][1] = col + i;
    } else {
        cells[i][0] = row + i;
        cells[i][1] = col;
    }
}
return cells;
}
}

```

Attribute	Tipe Data	Keterangan
id	private char	Atribut ID piece
row	private int	Atribut baris
col	private int	Atribut kolom
size	private int	Atribut ukuran piece
isHorizontal	private boolean	Atribut untuk membedakan piece horizontal dengan yang vertikal
isPrimary	private boolean	Atribut untuk menentukan primary piece

Method	Returned Data Type	Keterangan
Piece	-	Konstruktor
getId	char	Metode untuk mengembalikan ID piece
getRow	int	Metode untuk mengembalikan baris
getCol	int	Metode untuk mengembalikan kolom
getSize	int	Metode untuk mengembalikan ukuran piece
isHorizontal	boolean	Metode yang mengembalikan

		true apabila piece horizontal
isPrimary	boolean	Metode yang mengembalikan true apabila piece merupakan primary piece
getPossibleMoves	Move[]	Metode yang mengembalikan semua Move yang mungkin untuk dilakukan piece
applyMove	Piece	Metode yang menggerakkan Piece
getOccupiedCells	int[][]	Metode yang mengembalikan posisi-posisi pada papan yang sudah ditempati piece

3.5 AStar.java

Berikut ini merupakan attribute dan method yang terdapat dalam class AStar.java

```
package algorithm;

import core.Board;
import core.GameState;
import core.Move;

import java.util.*;

public class AStar {
    private int nodesVisited;
    private double executionTime;
    private String heuristicName;
    private Board initialBoard;

    public AStar(String heuristicName) {
        this.nodesVisited = 0;
        this.executionTime = 0.0;
        this.heuristicName = heuristicName;
    }
}
```

```
public GameState solve(Board initialBoard) {
    nodesVisited = 0;
    this.initialBoard = initialBoard;
    long startTime = System.nanoTime();

    PriorityQueue<GameState> openSet = new
PriorityQueue<>(Comparator.comparingDouble(GameState::getF));

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

    Map<String, Double> bestGScore = new HashMap<>();

    GameState startState = new GameState(initialBoard, heuristicName);
    openSet.add(startState);

    String startStateKey = getBoardKey(startState.getBoard());
    bestGScore.put(startStateKey, 0.0);

    while (!openSet.isEmpty()) {
        GameState current = openSet.poll();
        nodesVisited++;

        if (current.isGoal()) {
            executionTime = (System.nanoTime() - startTime) / 1_000_000.0;
            return current;
        }

        String currentKey = getBoardKey(current.getBoard());

        if (closedSet.contains(currentKey)) {
            continue;
        }

        closedSet.add(currentKey);

        List<GameState> successors = current.getSuccessors();

        for (GameState successor : successors) {
```

```
String successorKey = getBoardKey(successor.getBoard());

    if (closedSet.contains(successorKey)) {
        continue;
    }

    double tentativeG = successor.getG();

    if (!bestGScore.containsKey(successorKey) || tentativeG <
bestGScore.get(successorKey)) {
        bestGScore.put(successorKey, tentativeG);

        if (!isOpenSet(openSet, successorKey)) {
            openSet.add(successor);
        }
    }
}

executionTime = (System.nanoTime() - startTime) / 1_000_000.0;
return null;
}

private boolean isOpenSet(PriorityQueue<GameState> openSet, String key) {
    for (GameState state : openSet) {
        if (getBoardKey(state.getBoard()).equals(key)) {
            return true;
        }
    }
    return false;
}

private String getBoardKey(Board board) {
    StringBuilder key = new StringBuilder();
    char[][] grid = board.getGrid();
    for (int i = 0; i < board.getRows(); i++) {
        for (int j = 0; j < board.getCols(); j++) {
            key.append(grid[i][j]);
        }
    }
    return key.toString();
}
```

```
        }

    }

    return key.toString();
}

public void printSolution(GameState solution) {
    if (solution == null) {
        System.out.println("Tidak ada solusi yang ditemukan!");
        return;
    }

    System.out.println("Menggunakan heuristic: " + heuristicName);

    List<Move> moves = solution.getMoves();

    System.out.println("Papan Awal");
    initialBoard.printBoard(null);

    Board currentBoard = initialBoard;
    for (int i = 0; i < moves.size(); i++) {
        Move move = moves.get(i);
        currentBoard = currentBoard.applyMove(move);
        System.out.println("Gerakan " + (i + 1) + ": " + move);
        currentBoard.printBoard(move);
    }

    System.out.println("Solusi ditemukan dalam " + moves.size() + " langkah");
    System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
    System.out.println("Waktu eksekusi: " + executionTime + " ms");
}

public int getNodesVisited() {
    return nodesVisited;
}

public double getExecutionTime() {
    return executionTime;
```

```

    }

    public String getHeuristicName() {
        return heuristicName;
    }
}

```

Attribute	Tipe Data	Keterangan
nodesVisited	private int	Atribut simpul yang dikunjungi
executionTime	private double	Atribut waktu eksekusi
heuristicName	private String	Atribut nama heuristic
initialBoard	private Board	Atribut papan awal

Method	Returned Data type	Keterangan
AStar	-	Konstruktor default
solve	GameState	Metode untuk menyelesaikan puzzle Rush Hour menggunakan algoritma A* dan mengembalikan solusi akhir
isInOpenSet	boolean	Metode untuk mengecek apakah suatu konfigurasi papan sudah berada di antrian (openSet)
getBoardKey	String	Metode untuk mengubah konfigurasi grid papan menjadi string sebagai kunci pencarian unik
printSolution	-	Metode untuk menampilkan solusi secara visual beserta langkah-langkah dan papan setelah tiap langkah
getNodesVisited	int	Metode untuk

		mengembalikan jumlah node yang dikunjungi selama proses pencarian solusi
getExecutionTime	double	Metode untuk mengembalikan waktu eksekusi algoritma dalam satuan milidetik
getHeuristicName	String	Metode untuk mengembalikan nama heuristic yang digunakan pada eksekusi algoritma A*

3.6 Dijkstra.java

Berikut ini merupakan attribute dan method yang terdapat dalam class Dijkstra.java

```
package algorithm;

import core.Board;
import core.GameState;
import core.Move;

import java.util.*;

public class Dijkstra {
    private int nodesVisited;
    private double executionTime;
    private Board initialBoard;

    public Dijkstra() {
        this.nodesVisited = 0;
        this.executionTime = 0.0;
    }

    public GameState solve(Board initialBoard) {
        this.initialBoard = initialBoard;
        nodesVisited = 0;
        long startTime = System.nanoTime();
    }
}
```

```
PriorityQueue<GameState> openSet = new PriorityQueue<>(
    Comparator.comparingDouble(GameState::getG));

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

Map<String, Double> bestCost = new HashMap<>();

GameState startState = new GameState(initialBoard);
openSet.add(startState);

String startStateKey = getBoardKey(startState.getBoard());
bestCost.put(startStateKey, 0.0);

while (!openSet.isEmpty()) {
    GameState current = openSet.poll();
    nodesVisited++;

    String currentKey = getBoardKey(current.getBoard());

    if (current.isGoal()) {
        executionTime = (System.nanoTime() - startTime) / 1_000_000.0;
        return current;
    }

    if (closedSet.contains(currentKey)) {
        continue;
    }

    closedSet.add(currentKey);

    List<GameState> successors = current.getSuccessors();

    for (GameState successor : successors) {
        String successorKey = getBoardKey(successor.getBoard());

        if (closedSet.contains(successorKey)) {
            continue;
        }
    }
}
```

```
        double newCost = successor.getG();

        if (!bestCost.containsKey(successorKey) || newCost <
bestCost.get(successorKey)) {
            bestCost.put(successorKey, newCost);

            if (!isOpenSet(openSet, successorKey)) {
                openSet.add(successor);
            } else {
                updateOpenSet(openSet, successor, successorKey);
            }
        }
    }

    executionTime = (System.nanoTime() - startTime) / 1_000_000.0;
    return null;
}

private boolean isOpenSet(PriorityQueue<GameState> openSet, String key) {
    for (GameState state : openSet) {
        if (getBoardKey(state.getBoard()).equals(key)) {
            return true;
        }
    }
    return false;
}

private void updateOpenSet(PriorityQueue<GameState> openSet, GameState
newState, String key) {
    List<GameState> tempStates = new ArrayList<>();

    while (!openSet.isEmpty()) {
        GameState state = openSet.poll();
        if (!getBoardKey(state.getBoard()).equals(key)) {
            tempStates.add(state);
        }
    }
}
```

```
    }

    openSet.addAll(tempStates);
    openSet.add(newState);
}

private String getBoardKey(Board board) {
    StringBuilder key = new StringBuilder();
    char[][] grid = board.getGrid();
    for (int i = 0; i < board.getRows(); i++) {
        for (int j = 0; j < board.getCols(); j++) {
            key.append(grid[i][j]);
        }
    }
    return key.toString();
}

public void printSolution(GameState solution) {
    if (solution == null) {
        System.out.println("No solution found!");
        return;
    }

    List<Move> moves = solution.getMoves();

    System.out.println("Papan Awal");
    initialBoard.printBoard(null);

    Board currentBoard = initialBoard;
    for (int i = 0; i < moves.size(); i++) {
        Move move = moves.get(i);
        currentBoard = currentBoard.applyMove(move);
        System.out.println("Gerakan " + (i + 1) + ": " + move);
        currentBoard.printBoard(move);
    }

    System.out.println("Solusi ditemukan dalam " + moves.size() + " langkah");
}
```

```

        System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
        System.out.println("Waktu eksekusi: " + executionTime + " ms");
    }

    public int getNodesVisited() {
        return nodesVisited;
    }

    public double getExecutionTime() {
        return executionTime;
    }
}

```

Attribute	Tipe Data	Keterangan
nodesVisited	private int	Atribut simpul yang dikunjungi
executionTime	private double	Atribut waktu eksekusi
initialBoard	private Board	Atribut papan awal

Method	Returned Data type	Keterangan
Dijkstra	-	Konstruktor default
solve	GameState	Metode untuk menyelesaikan puzzle Rush Hour menggunakan algoritma Dijkstra dan mengembalikan solusi akhir
isInOpenSet	boolean	Metode untuk mengecek apakah suatu konfigurasi papan sudah berada di antrian (openSet)
getBoardKey	String	Metode untuk mengubah konfigurasi grid papan menjadi string sebagai kunci pencarian unik
printSolution	-	Metode untuk menampilkan

		solusi secara visual beserta langkah-langkah dan papan setelah tiap langkah
getNodesVisited	int	Metode untuk mengembalikan jumlah node yang dikunjungi selama proses pencarian solusi
getExecutionTime	double	Metode untuk mengembalikan waktu eksekusi algoritma dalam satuan milidetik

3.7 GBFS.java

Berikut ini merupakan attribute dan method yang terdapat dalam class GBFS.java

```
ackage algorithm;

import core.Board;
import core.GameState;
import core.Move;

import java.util.*;

public class GBFS {
    private static class Node {
        GameState state;

        Node(GameState state, Node parent) {
            this.state = state;
        }
    }

    private int nodesVisited;
    private double executionTime;
    private String heuristicName;
    private Board initialBoard;

    public GBFS(String heuristicName) {
        this.nodesVisited = 0;
```

```
        this.executionTime = 0.0;
        this.heuristicName = heuristicName;
    }

    public GameState solve(Board initialBoard) {
        this.initialBoard = initialBoard;
        nodesVisited = 0;
        long startTime = System.nanoTime();

        PriorityQueue<Node> openList = new PriorityQueue<>(
            Comparator.comparingDouble(n -> n.state.getH())
        );
        Set<String> closedList = new HashSet<>();

        GameState initialState = new GameState(initialBoard, heuristicName);
        Node initialNode = new Node(initialState, null);
        openList.add(initialNode);

        while (!openList.isEmpty()) {
            Node currentNode = openList.poll();
            GameState currentState = currentNode.state;
            nodesVisited++;

            String currentKey = getBoardKey(currentState.getBoard());

            if (currentState.isGoal()) {
                executionTime = (System.nanoTime() - startTime) / 1_000_000.0;
// Convert to milliseconds
                return currentState;
            }

            if (closedList.contains(currentKey)) {
                continue;
            }

            closedList.add(currentKey);

            for (GameState successor : currentState.getSuccessors()) {
```

```
        String successorKey = getBoardKey(successor.getBoard());
        if (!closedList.contains(successorKey)) {
            Node successorNode = new Node(successor, currentNode);
            openList.add(successorNode);
        }
    }

    executionTime = (System.nanoTime() - startTime) / 1_000_000.0;
    return null;
}

private String getBoardKey(Board board) {
    StringBuilder key = new StringBuilder();
    char[][] grid = board.getGrid();
    for (int i = 0; i < board.getRows(); i++) {
        for (int j = 0; j < board.getCols(); j++) {
            key.append(grid[i][j]);
        }
    }
    return key.toString();
}

public void printSolution(GameState solution) {
    if (solution == null) {
        System.out.println("Tidak ada solusi yang ditemukan!");
        return;
    }

    System.out.println("Menggunakan heuristic: " + heuristicName);

    List<Move> moves = solution.getMoves();

    System.out.println("Papan Awal:");
    initialBoard.printBoard(null);

    Board currentBoard = initialBoard;
    for (int i = 0; i < moves.size(); i++) {
```

```

        Move move = moves.get(i);
        currentBoard = currentBoard.applyMove(move);
        System.out.println("Gerakan " + (i + 1) + ": " + move);
        currentBoard.printBoard(move);
    }

    System.out.println("Solusi ditemukan dalam " + moves.size() + " langkah");
    System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
    System.out.println("Waktu eksekusi: " + executionTime + " ms");
}

public int getNodesVisited() {
    return nodesVisited;
}

public double getExecutionTime() {
    return executionTime;
}

public String getHeuristicName() {
    return heuristicName;
}
}

```

Attribute	Tipe Data	Keterangan
nodesVisited	private int	Atribut simpul yang dikunjungi
executionTime	private double	Atribut waktu eksekusi
heuristicName	private String	Atribut nama heuristic
initialBoard	private Board	Atribut papan awal

Method	Returned Data type	Keterangan
GBFS	-	Konstruktor default

solve	GameState	Metode untuk menyelesaikan puzzle Rush Hour menggunakan algoritma GBFS dan mengembalikan solusi akhir
isInOpenSet	boolean	Metode untuk mengecek apakah suatu konfigurasi papan sudah berada di antrian (openSet)
getBoardKey	String	Metode untuk mengubah konfigurasi grid papan menjadi string sebagai kunci pencarian unik
printSolution	-	Metode untuk menampilkan solusi secara visual beserta langkah-langkah dan papan setelah tiap langkah
getNodesVisited	int	Metode untuk mengembalikan jumlah node yang dikunjungi selama proses pencarian solusi
getExecutionTime	double	Metode untuk mengembalikan waktu eksekusi algoritma dalam satuan milidetik
getHeuristicName	String	Metode untuk mengembalikan nama heuristic yang digunakan pada eksekusi algoritma A*

3.8 UCS.java

Berikut ini merupakan attribute dan method yang terdapat dalam class UCS.java

```
package algorithm;

import core.Board;
import core.GameState;
import core.Move;
```

```
import java.util.*;  
  
public class UCS {  
    private static class Node {  
        GameState state;  
        double cost;  
  
        Node(GameState state, double cost, Node parent) {  
            this.state = state;  
            this.cost = cost;  
        }  
    }  
  
    private int nodesVisited;  
    private double executionTime;  
  
    private Board initialBoard;  
  
    public UCS() {  
        this.nodesVisited = 0;  
        this.executionTime = 0.0;  
    }  
  
    public GameState solve(Board board) {  
        this.initialBoard = board;  
  
        PriorityQueue<Node> openList = new  
PriorityQueue<>(Comparator.comparingDouble(n -> n.cost));  
        Set<GameState> closedList = new HashSet<>();  
  
        GameState initialState = new GameState(board, "none");  
        Node initialNode = new Node(initialState, 0, null);  
        openList.add(initialNode);  
  
        nodesVisited = 0;  
        long startTime = System.nanoTime();  
  
        while (!openList.isEmpty()) {
```

```
Node currentNode = openList.poll();
GameState currentState = currentNode.state;
nodesVisited++;

if (currentState.isGoal()) {
    executionTime = (System.nanoTime() - startTime) / 1_000_000.0;
    return currentState;
}

if (closedList.contains(currentState)) {
    continue;
}

closedList.add(currentState);

for (GameState successor : currentState.getSuccessors()) {
    if (!closedList.contains(successor)) {
        Node successorNode = new Node(successor, successor.getG(),
currentNode);
        openList.add(successorNode);
    }
}
}

executionTime = (System.nanoTime() - startTime) / 1_000_000.0;
return null;
}

public void printSolution(GameState solution) {
    if (solution == null) {
        System.out.println("Tidak ada solusi yang ditemukan!");
        return;
    }

    List<Move> moves = solution.getMoves();

    System.out.println("Papan Awal:");
}
```

```

        initialBoard.printBoard(null);

        Board currentBoard = initialBoard;
        for (int i = 0; i < moves.size(); i++) {
            Move move = moves.get(i);
            currentBoard = currentBoard.applyMove(move);
            System.out.println("Gerakan " + (i + 1) + ": " + move);
            currentBoard.printBoard(move);
        }
    }

    public int getNodesVisited() {
        return nodesVisited;
    }

    public double getExecutionTime() {
        return executionTime;
    }
}

```

Attribute	Tipe Data	Keterangan
nodesVisited	private int	Atribut simpul yang dikunjungi
executionTime	private double	Atribut waktu eksekusi
initialBoard	private Board	Atribut papan awal

Method	Returned Data type	Keterangan
UCS	-	Konstruktor default
solve	GameState	Metode untuk menyelesaikan puzzle Rush Hour menggunakan algoritma UCS dan mengembalikan solusi akhir
isInOpenSet	boolean	Metode untuk mengecek apakah suatu konfigurasi

		papan sudah berada di antrian (openSet)
getBoardKey	String	Metode untuk mengubah konfigurasi grid papan menjadi string sebagai kunci pencarian unik
printSolution	-	Metode untuk menampilkan solusi secara visual beserta langkah-langkah dan papan setelah tiap langkah
getNodesVisited	int	Metode untuk mengembalikan jumlah node yang dikunjungi selama proses pencarian solusi
getExecutionTime	double	Metode untuk mengembalikan waktu eksekusi algoritma dalam satuan milidetik

3.9 Heuristics.java

Berikut ini merupakan attribute dan method yang terdapat dalam class Heuristics.java

```
algorithm;

import core.Board;
import core.Piece;

public class Heuristics {
    public static double manhattanDistance(Board board) {
        Piece primary = board.getPrimaryPiece();
        int exitRow = board.getExitRow();
        int exitCol = board.getExitCol();

        if (primary.isHorizontal()) {
            int pieceEndCol = primary.getCol() + primary.getSize() - 1;

            if (exitCol >= board.getCols()) {
                return board.getCols() - 1 - pieceEndCol;
            } else {

```

```
        return Math.abs(pieceEndCol - exitCol);
    }
} else {
    int pieceEndRow = primary.getRow() + primary.getSize() - 1;

    if (exitRow >= board.getRows()) {
        return board.getRows() - 1 - pieceEndRow;
    } else {
        return Math.abs(pieceEndRow - exitRow);
    }
}

public static double blockingVehicles(Board board) {
    Piece primary = board.getPrimaryPiece();
    int exitRow = board.getExitRow();
    int exitCol = board.getExitCol();
    int count = 0;

    if (primary.isHorizontal()) {
        int primaryRow = primary.getRow();
        int primaryEndCol = primary.getCol() + primary.getSize() - 1;

        int startCol, endCol;

        if (exitCol >= board.getCols()) {
            startCol = primaryEndCol + 1;
            endCol = board.getCols() - 1;
        } else if (exitCol < 0) {
            startCol = 0;
            endCol = primary.getCol() - 1;
        } else if (exitCol > primaryEndCol) {
            startCol = primaryEndCol + 1;
            endCol = exitCol - 1;
        } else {
            startCol = exitCol + 1;
            endCol = primary.getCol() - 1;
        }
    }
}
```

```
startCol = Math.max(0, startCol);
endCol = Math.min(board.getCols() - 1, endCol);

for (int col = startCol; col <= endCol; col++) {
    if (primaryRow >= 0 && primaryRow < board.getRows() && col >= 0
&& col < board.getCols()) {
        char cell = board.getGrid()[primaryRow][col];
        if (cell != '.' && cell != 'K') {
            count++;
        }
    }
} else {
    int primaryCol = primary.getCol();
    int primaryEndRow = primary.getRow() + primary.getSize() - 1;

    int startRow, endRow;

    if (exitRow >= board.getRows()) {
        startRow = primaryEndRow + 1;
        endRow = board.getRows() - 1;
    } else if (exitRow < 0) {
        startRow = 0;
        endRow = primary.getRow() - 1;
    } else if (exitRow > primaryEndRow) {
        startRow = primaryEndRow + 1;
        endRow = exitRow - 1;
    } else {
        startRow = exitRow + 1;
        endRow = primary.getRow() - 1;
    }

    startRow = Math.max(0, startRow);
    endRow = Math.min(board.getRows() - 1, endRow);

    for (int row = startRow; row <= endRow; row++) {
```

```

        if (row >= 0 && row < board.getRows() && primaryCol >= 0 &&
primaryCol < board.getCols()) {
            char cell = board.getGrid()[row][primaryCol];
            if (cell != '.' && cell != 'K') {
                count++;
            }
        }
    }

    return count;
}

public static double combined(Board board) {
    return manhattanDistance(board) + 2 * blockingVehicles(board);
}

public static double getHeuristic(Board board, String heuristicName) {
    switch (heuristicName.toLowerCase()) {
        case "manhattan":
            return manhattanDistance(board);
        case "blocking":
            return blockingVehicles(board);
        case "combined":
            return combined(board);
        default:
            return manhattanDistance(board);
    }
}
}

```

Method	Returned Data Type	Keterangan
manhattanDistance	static double	Metode untuk menghitung jarak Manhattan dari ujung piece utama ke exit dan digunakan sebagai heuristic dasar

blockingVehicles	static double	Metode untuk menghitung jumlah kendaraan yang menghalangi jalur piece utama ke exit
combined	static double	Metode yang menggabungkan manhattanDistance dan dua kali blockingVehicles untuk heuristic gabungan.
getHeuristic	static double	Metode untuk memanggil metode heuristic sesuai nama (manhattan, blocking, combined) dengan default ke manhattan

3.10 AboutPage.java

Berikut ini merupakan attribute dan method yang terdapat dalam class AboutPage.java

```
package gui;

import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextFlow;

public class AboutPage extends VBox {

    private Button backButton;

    public AboutPage() {

        this.setSpacing(20);
        this.setAlignment(Pos.CENTER);
        this.setPadding(new Insets(30));
    }
}
```

```
this.getStyleClass().add("about-page");

Text title = new Text("What's Under the Hood?");
title.setFont(Font.font("Arial", FontWeight.BOLD, 32));
title.getStyleClass().add("page-title");

TextFlow descriptionFlow = createDescriptionText();

ScrollPane scrollPane = new ScrollPane(descriptionFlow);
scrollPane.setFitToWidth(true);
scrollPane.setPrefHeight(350);
scrollPane.getStyleClass().add("about-scroller");

Label versionLabel = new Label("Version 1.0 - May 2025");
versionLabel.getStyleClass().add("version-label");

backButton = new Button("Back to Garage");
backButton.getStyleClass().add("back-button");
backButton.setPrefWidth(150);

this.getChildren().addAll(
    title,
    scrollPane,
    versionLabel,
    backButton
);
}

private TextFlow createDescriptionText() {
    TextFlow textFlow = new TextFlow();
    textFlow.setLineSpacing(5);
    textFlow.setPadding(new Insets(10));

    Text intro = new Text(
        "Ever found yourself stuck in a cartoon traffic jam, boxed in by
        cars that just won't budge? "
        + "Welcome to Rush Hour, your intelligent parking-lot escape
        assistant!\n\n"
    );
}
```

```
);

intro.setFont(Font.font("Arial", 14));

Text algorithmTitle = new Text("Implemented Algorithms\n");
algorithmTitle.setFont(Font.font("Arial", FontWeight.BOLD, 16));

Text algorithmDesc = new Text(
    "• A* (A-Star)\n"
    + "A balanced navigator that considers both how far you've driven
and how close you are to the exit. It's like having a built-in GPS that
actually knows what it's doing.\n\n"
    + "• Greedy Best First Search (GBFS)\n"
    + "Picks the most promising-looking path – the one that seems
closest to the goal. Fast, a little reckless, but great when you're in a
hurry!\n\n"
    + "• Uniform Cost Search (UCS)\n"
    + "The methodical one. It explores every possible route based on
the cost to get there. Like a careful driver watching fuel efficiency, it
guarantees the optimal solution, even if it takes a few extra turns.\n\n"
);
algorithmDesc.setFont(Font.font("Arial", 14));

Text worksTitle = new Text("How it Works\n");
worksTitle.setFont(Font.font("Arial", FontWeight.BOLD, 16));

Text worksInfo = new Text(
    "Just upload a .txt file containing the puzzle configuration, "
    + "our mechanics will read the map, shuffle the cars, and guide the
hero straight to the exit.\n\n"
);
worksInfo.setFont(Font.font("Arial", 14));

Text licenseTitle = new Text("Licenses\n");
licenseTitle.setFont(Font.font("Arial", FontWeight.BOLD, 16));

Text courseInfo = new Text(
    "This project was developed as part of the IF2211 Strategi
Algoritma "
```

```

        + "course at Institut Teknologi Bandung, Semester II 2024/2025."
    );
    courseInfo.setFont(Font.font("Arial", 14));

    textFlow.getChildren().addAll(intro, algorithmTitle, algorithmDesc,
worksTitle, worksInfo, licenseTitle, courseInfo);
    return textFlow;
}

public Button getBackButton() {
    return backButton;
}
}

```

Attribute	Tipe Data	Keterangan
backButton	private Button	Atribut tombol (<i>button</i>) kembali

Method	Returned Data Type	Keterangan
AboutPage	-	Konstruktor
createDescriptionText	TextFlow	Metode untuk membuat teks deskripsi
getBackButton	Button	Metode untuk mengembalikan <i>back button</i>

3.11 BoardPane.java

Berikut ini merupakan attribute dan method yang terdapat dalam class BoardPane.java

```

package gui;

import javafx.animation.TranslateTransition;
import javafx.geometry.Insets;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;

```

```
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.util.Duration;
import javafx.scene.text.Text;

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

public class BoardPane extends GridPane {

    private int rows = 6;
    private int cols = 6;
    private double cellSize = 60;
    private Map<Character, StackPane> pieceMap;
    private char[][] board;

    private int exitRow = -1;
    private int exitCol = -1;

    private int gridRows;
    private int gridCols;

    public BoardPane() {
        this.getStyleClass().add("board-pane");
        pieceMap = new HashMap<>();

        setHgap(2);
        setVgap(2);
        setPadding(new Insets(15));

        this.setMaxWidth((cols + 1) * (cellSize + 2) + 30);
        this.setMaxHeight((rows + 1) * (cellSize + 2) + 30);

        System.out.println("Working directory: " +
System.getProperty("user.dir"));
    }
}
```

```
public void initializeBoard(char[][] board) {
    this.board = board;
    this.rows = board.length;
    this.cols = board[0].length;

    this.getChildren().clear();
    pieceMap.clear();

    try {
        findExitPosition();
        calculateGridDimensions();

        System.out.println("Board contents:");
        for (int i = 0; i < board.length; i++) {
            System.out.println(new String(board[i]));
        }
        System.out.println("Exit position: row=" + exitRow + ", col=" +
exitCol);
        System.out.println("Grid dimensions: rows=" + gridRows + ", cols=" +
gridCols);

        createEmptyCells();
        placePieces();

        if (exitRow >= 0 && exitCol >= 0) {
            placeExitDoor();
        } else {
            System.err.println("WARNING: No exit door (K) found in the
board!");
        }
    } catch (RuntimeException e) {
        System.err.println(e.getMessage());
    }
}

private void findExitPosition() {
    exitRow = -1;
    exitCol = -1;
```

```
        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board[row].length; col++) {
                if (board[row][col] == 'K') {
                    exitRow = row;
                    exitCol = col;
                    System.out.println("Exit found in board at [" + row + ", " +
col + "]");
                    return;
                }
            }
        }

        System.out.println("Exit not found in board matrix");
    }

    private void calculateGridDimensions() {
        gridRows = rows;
        gridCols = cols;
    }

    private void createEmptyCells() {
        for (int row = 0; row < gridRows; row++) {
            for (int col = 0; col < gridCols; col++) {
                Rectangle cell = new Rectangle(cellSize, cellSize);
                cell.getStyleClass().add("rectangle-cell");
                cell.setFill(Color.web("#f5f5f5"));
                cell.setStroke(Color.web("#e0e0e0"));
                cell.setStrokeWidth(1);
                cell.setArcWidth(5);
                cell.setArcHeight(5);
                add(cell, col, row);
            }
        }
    }

    private void placeExitDoor() {
```

```
        if (exitRow < 0 || exitCol < 0 || exitRow >= board.length || exitCol >= board[0].length) {
            System.err.println("ERROR: Exit position [" + exitRow + "," + exitCol + "] is out of bounds!");
            return;
        }

        StackPane exitNode = createExitNode();

        getChildren().removeIf(node -> {
            Integer colIndex = GridPane.getColumnIndex(node);
            Integer rowIndex = GridPane.getRowIndex(node);
            return colIndex != null && rowIndex != null &&
                colIndex == exitCol && rowIndex == exitRow;
        });

        add(exitNode, exitCol, exitRow);
        System.out.println("Placed EXIT door at row=" + exitRow + ", col=" + exitCol);

        pieceMap.put('K', exitNode);
    }

    private StackPane createExitNode() {
        StackPane exitPane = new StackPane();
        Rectangle exitRect = new Rectangle(cellSize, cellSize);
        exitRect.getStyleClass().add("exit-door");

        exitRect.setFill(Color.web("#FF5722"));
        exitRect.setStroke(Color.web("#BF360C"));
        exitRect.setStrokeWidth(2);
        exitRect.setArcWidth(10);
        exitRect.setArcHeight(10);

        Text exitLetter = new Text("K");
        exitLetter.setFill(Color.WHITE);
        exitLetter.setFont(Font.font("Arial", FontWeight.BOLD, 18));
    }
}
```

```

        exitPane.getChildren().addAll(exitRect, exitLetter);
    }

    private void placePieces() {
        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                if (row < board.length && col < board[row].length) {
                    char piece = board[row][col];
                    if (piece != '.' && piece != 'K') {
                        placePiece(piece, row, col);
                    } else if (piece == '.') {
                        Text dotText = new Text(".");
                        dotText.setFill(Color.LIGHTGRAY);
                        dotText.setFont(Font.font("Arial", FontWeight.NORMAL,
12));
                        add(dotText, col, row);
                    }
                }
            }
        }
    }

    private void placePiece(char piece, int row, int col) {
        if (!pieceMap.containsKey(piece)) {
            StackPane pieceNode = createPieceNode(piece);
            add(pieceNode, col, row);

            pieceMap.put(piece, pieceNode);

            if (isHorizontalPiece(piece, row, col)) {
                int size = getPieceSize(piece, row, col, true);
                Rectangle pieceRect = (Rectangle)
pieceNode.getChildren().get(0);
                pieceRect.setWidth(cellSize * size + (size - 1) * getHgap());
                GridPane.setColumnSpan(pieceNode, size);
            } else if (isVerticalPiece(piece, row, col)) {
                int size = getPieceSize(piece, row, col, false);
            }
        }
    }
}

```

```
        Rectangle pieceRect = (Rectangle)
pieceNode.getChildren().get(0);
        pieceRect.setHeight(cellSize * size + (size - 1) * getVgap());
        GridPane.setRowSpan(pieceNode, size);
    }
}
}

private StackPane createPieceNode(char piece) {
    StackPane piecePane = new StackPane();
    Rectangle rect = createPieceRectangle(piece);

    Text pieceText = new Text(String.valueOf(piece));
    pieceText.setFill(Color.WHITE);
    pieceText.setFont(Font.font("Arial", FontWeight.BOLD, 18));

    piecePane.getChildren().addAll(rect, pieceText);
    return piecePane;
}

private Rectangle createPieceRectangle(char piece) {
    Rectangle rect = new Rectangle(cellSize, cellSize);

    if (piece == 'P') {
        rect.getStyleClass().add("primary-piece");
        rect.setFill(Color.web("#F25E59"));
    } else {
        rect.getStyleClass().add("piece");
        rect.getStyleClass().add("piece-" + piece);

        String[] colorPalette = {
            "#FAD390", "#F8C291", "#6A89CC", "#82CCDD",
            "#B8E994", "#F6B93B", "#78E08F", "#E55039",
            "#fa8231", "#f7b731", "#BDC581", "#A3CB38"
        };
        int colorIndex = (piece - 'A') % colorPalette.length;
        if (colorIndex < 0) colorIndex += colorPalette.length;
        rect.setFill(Color.web(colorPalette[colorIndex]));
    }
}
```

```
    }

    rect.setStroke(Color.web("#725861"));
    rect.setStrokeWidth(1.5);
    rect.setArcWidth(10);
    rect.setArcHeight(10);

    return rect;
}

private boolean isHorizontalPiece(char piece, int row, int col) {
    return col + 1 < cols && row < board.length && col + 1 <
board[row].length && board[row][col + 1] == piece;
}

private boolean isVerticalPiece(char piece, int row, int col) {
    return row + 1 < rows && row + 1 < board.length && col < board[row +
1].length && board[row + 1][col] == piece;
}

private int getPieceSize(char piece, int row, int col, boolean horizontal)
{
    int size = 1;
    if (horizontal) {
        for (int c = col + 1; c < cols && c < board[row].length &&
board[row][c] == piece; c++) {
            size++;
        }
    } else {
        for (int r = row + 1; r < rows && r < board.length && col <
board[r].length && board[r][col] == piece; r++) {
            size++;
        }
    }
    return size;
}

public void movePiece(char piece, String direction, int steps) {
```

```

        if (!pieceMap.containsKey(piece)) return;

        StackPane pieceNode = pieceMap.get(piece);
        TranslateTransition transition = new
TranslateTransition(Duration.millis(500), pieceNode);

        switch (direction) {
            case "up":
                transition.setByY(-(cellSize + getVgap()) * steps);
                break;
            case "down":
                transition.setByY((cellSize + getVgap()) * steps);
                break;
            case "left":
                transition.setByX(-(cellSize + getHgap()) * steps);
                break;
            case "right":
                transition.setByX((cellSize + getHgap()) * steps);
                break;
        }

        transition.play();
    }

    public void updateBoard(char[][][] newBoard) {
        this.board = newBoard;
        this.getChildren().clear();
        pieceMap.clear();
        initializeBoard(newBoard);
    }
}

```

Attribute	Tipe Data	Keterangan
rows	private int	Atribut baris
cols	private int	Atribut kolom
cellSize	private double	Atribut ukuran sel

pieceMap	private Map<Character, StackPane>	Atribut map untuk piece
board	private char[][]	Atribut papan
exitRow	private int	Atribut indeks baris keluar K
exitCol	private int	Atribut indeks kolom keluar K
gridRows	private int	Atribut jumlah baris grid
gridCols	private int	Atribut jumlah kolom grid

Method	Returned Data Type	Keterangan
BoardPane	-	Konstruktor
initializeBoard	-	Metode untuk menginisialisasi papan
findExitPosition	-	Metode untuk menemukan posisi keluar K
calculateGridDimensions	-	Metode untuk menghitung dimensi grid
createEmptyCells	-	Metode untuk membentuk/membuat sel kosong
placeExitDoor	-	Metode untuk menempatkan ‘pintu’ keluar K pada pieceMap
createExitNode	StackPane	Metode untuk membuat simpul ‘pintu’ keluar K
placePieces	-	Metode untuk meletakkan semua potongan kendaraan (selain K dan titik kosong) ke dalam grid berdasarkan papan saat ini
placePiece	-	Metode untuk meletakkan satu potongan kendaraan ke posisi tertentu pada grid dan

		menyesuaikan ukuran serta orientasinya
createPieceNode	StackPane	Metode untuk membuat node StackPane dari sebuah kendaraan, berisi bentuk persegi panjang dan teks identifikasi
createPieceRectangle	Rectangle	Metode untuk membuat bentuk persegi panjang (Rectangle) dari kendaraan, termasuk pewarnaan dan styling berdasarkan jenisnya
isHorizontalPiece	boolean	Metode yang mengembalikan true jika piece horizontal
isVerticalPiece	boolean	Metode yang mengembalikan true jika piece vertical
getPieceSize	int	Metode untuk mengembalikan ukuran piece
movePiece	-	Metode untuk memindahkan atau menggeser piece
updateBoard	-	Metode untuk memperbarui papan

3.12 ControlPanel.java

Berikut ini merupakan attribute dan method yang terdapat dalam class ControlPanel.java

a

Method	Returned Data Type	Keterangan
saveGIF	static boolean	Untuk menyimpan urutan frame gambar hasil dalam format .gif

3.13 CreatorPage.java

Berikut ini merupakan attribute dan method yang terdapat dalam class CreatorPage.java

package gui;

```
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class CreatorPage extends VBox {

    private Button backButton;

    public CreatorPage() {

        this.setSpacing(30);
        this.setAlignment(Pos.CENTER);
        this.setPadding(new Insets(30));
        this.getStyleClass().add("creator-page");

        Text title = new Text("Meet the Mechanics");
        title.setFont(Font.font("Arial", FontWeight.BOLD, 32));
        title.getStyleClass().add("page-title");

        Text description = new Text("Under the hood of this solver are two
engineers fine-tuning every move. \nThey don't fix real engines, they fix logic
puzzles!");
        description.setFont(Font.font("Arial", FontWeight.NORMAL, 16));
        description.getStyleClass().add("description-text");

        HBox creatorLayout = createCreatorLayout();

        backButton = new Button("Back to Garage");
        backButton.getStyleClass().add("back-button");
    }
}
```

```
backButton.setPrefWidth(150);

this.getChildren().addAll(
    title,
    description,
    creatorLayout,
    backButton
);
}

private HBox createCreatorLayout() {
    HBox layout = new HBox(30);
    layout.setAlignment(Pos.CENTER);

    VBox leftCreator = new VBox(10);
    leftCreator.setAlignment(Pos.CENTER_RIGHT);

    Label nameLeft = new Label("Shanice Feodora");
    nameLeft.setFont(Font.font("Arial", FontWeight.BOLD, 18));
    nameLeft.getStyleClass().add("creator-name");

    Label nimLeft = new Label("13523097");
    nimLeft.getStyleClass().add("creator-details");

    leftCreator.getChildren().addAll(nameLeft, nimLeft);

    ImageView creatorImage = createCreatorImageView();

    VBox rightCreator = new VBox(10);
    rightCreator.setAlignment(Pos.CENTER_LEFT);

    Label nameRight = new Label("Adinda Putri");
    nameRight.setFont(Font.font("Arial", FontWeight.BOLD, 18));
    nameRight.getStyleClass().add("creator-name");

    Label nimRight = new Label("13523071");
    nimRight.getStyleClass().add("creator-details");
```

```

        rightCreator.getChildren().addAll(nameRight, nimRight);

        layout.getChildren().addAll(leftCreator, creatorImage, rightCreator);

        return layout;
    }

    private ImageView createCreatorImageView() {
        try {
            Image creatorImage = new
Image(getClass().getResourceAsStream("/resources/images/creator.png"));
            ImageView creatorView = new ImageView(creatorImage);
            creatorView.setFitWidth(450);
            creatorView.setPreserveRatio(true);
            return creatorView;
        } catch (Exception e) {
            System.err.println("Error loading image: " + e.getMessage());
            return null;
        }
    }

    public Button getBackButton() {
        return backButton;
    }
}

```

Attribute	Tipe Data	Keterangan
backButton	private Button	Atribut tombol (<i>button</i>) kembali

Method	Returned Data Type	Keterangan
CreatorPage	-	Konstruktor
createCreatorLayout	HBox	Metode untuk membuat layout untuk halaman Creator
createCreatorImageView	ImageView	Metode untuk memuat dan

		menampilkan gambar kreator sebagai objek ImageView dengan ukuran yang disesuaikan
getBackButton	Button	Metode untuk mengembalikan <i>back button</i>

3.14 Renderer.java

Berikut ini merupakan attribute dan method yang terdapat dalam class Renderer.java

```
package gui;

import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.util.Duration;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import algorithm.AStar;
import algorithm.Dijkstra;
import algorithm.GBFS;
import algorithm.UCS;
import core.Board;
import core.GameState;
import core.Move;

public class Renderer {

    public interface StepChangeListener {
        void onStepChange(int stepIndex);
    }
}
```

```
private List<StepChangeListener> stepChangeListeners = new ArrayList<>();

public void addStepChangeListener(StepChangeListener listener) {
    stepChangeListeners.add(listener);
}

private BoardPane boardPane;
private char[][] currentBoard;
private List<MoveStep> solutionSteps;
private int currentStepIndex = -1;
private Timeline animation;

private String lastUsedAlgorithm = null;
private String lastUsedHeuristic = null;

private int totalMoves = 0;
private int nodesVisited = 0;
private long executionTime = 0;

public Renderer(BoardPane boardPane) {
    this.boardPane = boardPane;
    this.solutionSteps = new ArrayList<>();
    setupAnimation();
}

private void setupAnimation() {
    animation = new Timeline(
        new KeyFrame(Duration.millis(1000), e -> showNextMove())
    );
    animation.setCycleCount(Timeline.INDEFINITE);
}

public void loadPuzzleFromFile(File file) {
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(file));
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    String line;
    while ((line = reader.readLine()) != null) {
        currentBoard[rows - 1 - currentStepIndex][line.length() - 1] = line.charAt(0);
    }
}
```

```
// Parse dimensions
String dimensionsLine = reader.readLine();
if (dimensionsLine == null) {
    throw new IllegalArgumentException("File is empty or cannot be
read.");
}

String[] dimensions = dimensionsLine.split("\\s+");
if (dimensions.length < 2) {
    throw new IllegalArgumentException("Invalid dimensions
format");
}

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

System.out.println("Loading board with dimensions: " + rows + "x"
+ cols);

String secondLine = reader.readLine();
boolean isNumericLine = true;

try {
    Integer.parseInt(secondLine.trim());
} catch (NumberFormatException e) {
    isNumericLine = false;
}

List<String> boardLines = new ArrayList<>();
String line;
int leftPadding = 0;
boolean hasLeftK = false;
boolean hasTopK = false;
boolean exitExists = false;
int exitRow = -1, exitCol = -1;
int kCount = 0;
```

```

        while ((line = reader.readLine()) != null) {
            for (int i = 0; i < line.length(); i++) {
                if (line.charAt(i) == 'K') {
                    kCount++;
                }
            }

            String trimmed = line.trim();
            if (boardLines.isEmpty() && (trimmed.equals("K") ||
(trimmed.length() > 1 && trimmed.replace("K", "").trim().isEmpty())))) {
                hasTopK = true;
                exitCol = line.indexOf('K');
                System.out.println("K found in top row at col: " +
exitCol);
                continue;
            }
            boardLines.add(line);
        }

        if (kCount == 0) {
            throw new IllegalArgumentException("Error: No exit (K) found
in the puzzle.");
        } else if (kCount > 1) {
            throw new IllegalArgumentException("Error: Multiple exits (K)
found. Only one exit is allowed.");
        }
    }

    int minLeadingSpace = Integer.MAX_VALUE;
    for (String boardLine : boardLines) {
        if (boardLine.trim().isEmpty()) continue;
        int leadingSpaces =
boardLine.indexOf(boardLine.trim().charAt(0));
        minLeadingSpace = Math.min(minLeadingSpace, leadingSpaces);
    }
    if (minLeadingSpace == Integer.MAX_VALUE) minLeadingSpace = 0;
    leftPadding = minLeadingSpace;
}

```

```
System.out.println("Minimum leading spaces: " + leftPadding);

boolean foundPrimaryPiece = false;
boolean isHorizontal = false;
int pRow = -1, pCol = -1;
int pCount = 0;

for (int i = 0; i < boardLines.size(); i++) {
    String currentLine = boardLines.get(i);
    for (int j = 0; j < currentLine.length(); j++) {
        if (j < currentLine.length() && currentLine.charAt(j) ==
'P') {
            pCount++;
            if (!foundPrimaryPiece) {
                pRow = i;
                pCol = j - leftPadding;
                foundPrimaryPiece = true;
            }
        }
    }
}

if (pCount == 0) {
    throw new IllegalArgumentException("Error: No primary piece
(P) found in the puzzle.");
}

if (boardLines.size() > 0) {
    String firstLine = boardLines.get(0);
    if (firstLine.contains("K") && firstLine.indexOf('K') <=
firstLine.length() - 1) {
        exitExists = true;
        exitRow = 0;
        exitCol = firstLine.indexOf('K') - leftPadding;
        hasTopK = true;
    }
}
```

```
        System.out.println("K found in first row at col " +
exitCol);
    }
}

if (!exitExists) {
    for (int i = 0; i < Math.min(rows, boardLines.size()); i++) {
        String currentLine = boardLines.get(i);
        if (currentLine.trim().startsWith("K")) {
            exitExists = true;
            exitRow = i;

            int kPos = currentLine.indexOf('K');
            if (kPos - leftPadding <= 0) {
                exitCol = 0;
                hasLeftK = true;
                System.out.println("K found at start of row " + i
+ " with left padding " + leftPadding);
            } else {
                exitCol = kPos - leftPadding;
                System.out.println("K found in row " + i + " at
col " + exitCol);
            }
            break;
        }
    }
}

if (!exitExists) {
    for (int i = 0; i < Math.min(rows, boardLines.size()); i++) {
        String currentLine = boardLines.get(i);
        int kIndex = currentLine.indexOf('K');
        if (kIndex != -1) {
            exitExists = true;
            exitRow = i;
            exitCol = kIndex - leftPadding;
```

```

        if (kIndex - leftPadding <= 0) {
            hasLeftK = true;
            exitCol = 0;
        }

        System.out.println("K found in board at row " + i + ", "
col " + exitCol);
        break;
    }
}

if (!exitExists && boardLines.size() > rows) {

    for (int i = rows; i < boardLines.size(); i++) {
        String currentLine = boardLines.get(i);
        String trimmed = currentLine.trim();
        if (trimmed.contains("K")) {
            exitExists = true;
            exitRow = rows;

            if (!isHorizontal && foundPrimaryPiece) {
                exitCol = pCol;
            } else {
                exitCol = currentLine.indexOf('K') - leftPadding;
                if (exitCol < 0) exitCol = 0;
            }

            System.out.println("Found K in line " + i + " (after
board rows) at position [" + exitRow + "," + exitCol + "']");
            break;
        }

        int kIndex = currentLine.indexOf('K');
        if (kIndex != -1) {
            exitExists = true;
            exitRow = rows;
        }
    }
}

```

```

        exitCol = kIndex - leftPadding;
        if (exitCol < 0) exitCol = 0;

        System.out.println("Found K character in line after
board at col " + exitCol);
        break;
    }
}

if (!exitExists) {
    for (int i = 0; i < Math.min(rows, boardLines.size()); i++) {
        String currentLine = boardLines.get(i);
        if (currentLine.length() > cols + leftPadding) {
            int endIndex = cols + leftPadding;
            if (endIndex < currentLine.length() &&
currentLine.charAt(endIndex) == 'K') {
                exitExists = true;
                exitRow = i;
                exitCol = cols;
                System.out.println("K found at end of row " + i +
" after board width");
                break;
            }
        }
    }
}

for (int i = 0; i < Math.min(rows, boardLines.size()); i++) {
    String currentLine = boardLines.get(i);
    if (currentLine.length() <= leftPadding) continue;

    for (int j = leftPadding; j < currentLine.length(); j++) {
        char c = currentLine.charAt(j);

        if (c == 'P') {
            if (!foundPrimaryPiece) {

```

```

        pRow = i;
        pCol = j - leftPadding;
        foundPrimaryPiece = true;
    }

    if (j > leftPadding && currentLine.charAt(j-1) == 'P')
isHorizontal = true;
    if (j < currentLine.length()-1 &&
currentLine.charAt(j+1) == 'P') isHorizontal = true;
}
}

if (foundPrimaryPiece && !isHorizontal) {
    for (int i = 0; i < Math.min(rows, boardLines.size()); i++) {
        if (i == pRow) continue;

        String currentLine = boardLines.get(i);
        int adjustedCol = pCol + leftPadding;

        if (adjustedCol < currentLine.length() &&
currentLine.charAt(adjustedCol) == 'P') {
            isHorizontal = false;
            break;
        }
    }
}

System.out.println("Primary piece found at [" + pRow + "," + pCol
+ "], Orientation: " +
(isHorizontal ? "Horizontal" : "Vertical"));

if (exitExists) {
    System.out.println("Exit found at [" + exitRow + "," + exitCol
+ "] ");
    System.out.println("hasLeftK: " + hasLeftK + ", hasTopK: " +
hasTopK);
}

```

```

        boolean validOrientation = false;
        if (isHorizontal) {
            if (exitRow == pRow) {
                validOrientation = true;
            }
        } else {
            if (exitCol == pCol) {
                validOrientation = true;
            }
        }

        if (!validOrientation) {
            throw new IllegalArgumentException(
                "Error: Exit door (K) is not aligned with primary piece "
                + (P) + " +
                    "For horizontal primary pieces, exit must be in the
same row. " +
                    "For vertical primary pieces, exit must be in the same
column.");
        }
    } else {
        System.out.println("Warning: Exit exists but not detected in
parsing stage.");
    }

    int finalRows = rows;
    int finalCols = cols;

    if (exitExists && exitRow >= rows) {
        finalRows = exitRow + 1;
        System.out.println("Expanding rows for bottom exit: Final rows
= " + finalRows);
    }

    if (exitExists && exitCol >= cols) {

```

```

        finalCols = exitCol + 1;

        System.out.println("Expanding cols for right exit: Final cols
= " + finalCols);
    }

    if (hasLeftK) {
        finalCols += 1;

        System.out.println("Adding column for left K: Final cols = " +
finalCols);
    }

    if (!exitExists && foundPrimaryPiece) {
        if (isHorizontal) {
            exitRow = pRow;
            exitCol = cols;
            finalCols += 1;
            exitExists = true;
            System.out.println("Added automatic exit for horizontal at
[ " + exitRow + "," + exitCol + " ]");
        } else {
            exitRow = rows;
            exitCol = pCol;
            finalRows += 1;
            exitExists = true;
            System.out.println("Added automatic exit for vertical at
[ " + exitRow + "," + exitCol + " ]");
        }
    }

    currentBoard = new char[finalRows][finalCols];

    if (hasTopK) {
        for (int i = 0; i < finalRows; i++) {
            for (int j = 0; j < finalCols; j++) {
                currentBoard[i][j] = '.';
            }
        }
    }
}

```

```
        int adjustedExitCol = exitCol;
        if (hasLeftK) adjustedExitCol += 1;
        currentBoard[0][adjustedExitCol] = 'K';
        System.out.println("Placed K at top row [0," + adjustedExitCol
+ "]\n");

        for (int i = 0; i < Math.min(rows, boardLines.size()); i++) {
            String currentLine = boardLines.get(i);
            if (currentLine.length() <= leftPadding) continue;

            for (int j = leftPadding; j < currentLine.length(); j++) {
                char c = currentLine.charAt(j);
                if (c == ' ' || c == 'K') continue;

                int destCol = j - leftPadding;

                if (i+1 < finalRows && destCol >= 0 && destCol <
finalCols) {
                    currentBoard[i+1][destCol] = c;
                }
            }
        } else {
            for (int i = 0; i < finalRows; i++) {
                for (int j = 0; j < finalCols; j++) {
                    currentBoard[i][j] = '.';
                }
            }

            for (int i = 0; i < Math.min(rows, boardLines.size()); i++) {
                String currentLine = boardLines.get(i);
                if (currentLine.length() <= leftPadding) continue;

                for (int j = leftPadding; j < currentLine.length(); j++) {
                    char c = currentLine.charAt(j);
                    if (c == ' ') continue;
                }
            }
        }
    }
}
```

```

        int destCol = j - leftPadding;

        if (c == 'K' && hasLeftK && j - leftPadding <= 0) {
            destCol = 0;
        }

        if (i >= 0 && i < finalRows && destCol >= 0 && destCol
< finalCols) {
            currentBoard[i][destCol] = c;
        }
    }
}

if (exitExists) {
    if (hasLeftK) {
        currentBoard[exitRow][0] = 'K';
        System.out.println("Placed K at left [" + exitRow +
", 0]");
    }
}

if (exitRow >= rows) {
    int adjustedExitCol = exitCol;
    if (hasLeftK) adjustedExitCol += 1;

    if (!isHorizontal && foundPrimaryPiece) {
        adjustedExitCol = pCol + (hasLeftK ? 1 : 0);
    }
}

if (exitRow >= 0 && exitRow < finalRows &&
adjustedExitCol >= 0 && adjustedExitCol < finalCols) {
    currentBoard[exitRow][adjustedExitCol] = 'K';
    System.out.println("Placed K at bottom [" +
exitRow + "," + adjustedExitCol + "]");
}
}

```

```

        if (exitCol >= cols && !hasLeftK) {
            int adjustedExitCol = finalCols - 1;

            if (exitRow >= 0 && exitRow < finalRows) {
                currentBoard[exitRow][adjustedExitCol] = 'K';
                System.out.println("Placed K at right [" + exitRow
+ "," + adjustedExitCol + "]");
            }
        }
    }

boolean finalFoundP = false;
boolean finalFoundK = false;
boolean finalIsHorizontal = false;
int finalPRow = -1, finalPCol = -1;
int finalKRow = -1, finalKCol = -1;

for (int i = 0; i < currentBoard.length; i++) {
    for (int j = 0; j < currentBoard[i].length; j++) {
        if (currentBoard[i][j] == 'P') {
            finalFoundP = true;
            if (finalPRow == -1) {
                finalPRow = i;
                finalPCol = j;
            }

            if (j > 0 && j < currentBoard[i].length &&
currentBoard[i][j-1] == 'P') finalIsHorizontal = true;
            if (j < currentBoard[i].length-1 &&
currentBoard[i][j+1] == 'P') finalIsHorizontal = true;
        }

        if (currentBoard[i][j] == 'K') {
            finalFoundK = true;
            finalKRow = i;
            finalKCol = j;
        }
    }
}

```

```

        }

    }

    if (!finalFoundP) {
        throw new IllegalArgumentException("Error: Primary piece (P)
missing from final board.");
    }

    if (!finalFoundK) {
        throw new IllegalArgumentException("Error: Exit door (K)
missing from final board.");
    }

    boolean finalValidOrientation = false;
    if (finalIsHorizontal) {
        if (finalKRow == finalPRow) finalValidOrientation = true;
    } else {
        if (finalKCol == finalPCol) finalValidOrientation = true;
    }

    if (!finalValidOrientation) {
        throw new IllegalArgumentException(
            "Error: In final board, exit door (K) is not aligned with
primary piece (P).");
    }
}

System.out.println("Final board contents (" + finalRows + "x" +
finalCols + ")");
for (int i = 0; i < currentBoard.length; i++) {
    System.out.println(new String(currentBoard[i]));
}

boardPane.updateBoard(currentBoard);

solutionSteps.clear();
currentStepIndex = -1;

```

```
        solutionSteps.add(new MoveStep(copyBoard(currentBoard), null,
null, 0));

        System.out.println("File berhasil dimuat. Board " + finalRows +
"x" + finalCols);

    } catch (IOException e) {
        System.err.println("Error loading puzzle file: " +
e.getMessage());
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        System.err.println(e.getMessage());
        showErrorDialog("Puzzle Error", e.getMessage());
        e.printStackTrace();
    } catch (Exception e) {
        System.err.println("Error loading puzzle file: " +
e.getMessage());
        showErrorDialog("Puzzle Error", "Unexpected error: " +
e.getMessage());
        e.printStackTrace();
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                System.err.println("Error closing reader: " +
e.getMessage());
            }
        }
    }
}

public void showErrorDialog(String title, String message) {
    System.err.println("[ERROR DIALOG] " + title + ":" + message);

    javafx.application.Platform.runLater(() -> {
        javafx.stage.Stage dialogStage = new javafx.stage.Stage();
```

```
dialogStage.initModality(javafx.stage.Modality.APPLICATION_MODAL);
dialogStage.setTitle(title);
dialogStage.setResizable(false);

String primaryColor = "#fbb2a2";
String textColor = "#5a3e36";
String bgColor = "#ffffff";

javafx.scene.layout.VBox root = new javafx.scene.layout.VBox(15);
root.setPadding(new javafx.geometry.Insets(20));
root.setAlignment(javafx.geometry.Pos.CENTER);
root.setStyle("-fx-background-color: " + bgColor + "; " +
              "-fx-border-color: " + primaryColor + "; " +
              "-fx-border-width: 2px; " +
              "-fx-border-radius: 5px;");

javafx.scene.text.Text titleText = new
javafx.scene.text.Text(title);
titleText.setFont(javafx.scene.text.Font.font("Arial",
javafx.scene.FontWeight.BOLD, 16));
titleText.setFill(javafx.scene.paint.Color.web(textColor));

javafx.scene.layout.HBox titleBox = new
javafx.scene.layout.HBox(10);
titleBox.setAlignment(javafx.geometry.Pos.CENTER);
titleBox.setPadding(new javafx.geometry.Insets(0, 0, 5, 0));

javafx.scene.layout.StackPane iconPane = new
javafx.scene.layout.StackPane();
javafx.scene.shape.Circle circle = new
javafx.scene.shape.Circle(12);
circle.setFill(javafx.scene.paint.Color.web(primaryColor));

javafx.scene.text.Text icon = new javafx.scene.text.Text("!");
icon.setFont(javafx.scene.text.Font.font("Arial",
javafx.scene.FontWeight.BOLD, 16));
icon.setFill(javafx.scene.paint.Color.WHITE);
```

```
iconPane.getChildren().addAll(circle, icon);
titleBox.getChildren().addAll(iconPane, titleText);

javafx.scene.shape.Line separator = new javafx.scene.shape.Line();
separator.setStartX(0);
separator.setEndX(280);
separator.setStroke(javafx.scene.paint.Color.web(primaryColor));
separator.setStrokeWidth(1);

javafx.scene.text.Text messageText = new
javafx.scene.text.Text(message);
messageText.setFont(javafx.scene.text.Font.font("Arial", 14));
messageText.setFill(javafx.scene.paint.Color.web(textColor));

messageText.setTextAlignment(javafx.scene.text.TextAlignment.CENTER);
messageText.setWrappingWidth(280);

javafx.scene.control.Button okButton = new
javafx.scene.control.Button("OK");
okButton.setPrefSize(80, 30);
okButton.setStyle("-fx-background-color: " + primaryColor + "; " +
                  "-fx-text-fill: white; " +
                  "-fx-font-weight: bold; " +
                  "-fx-font-size: 14px; " +
                  "-fx-background-radius: 4px;");

okButton.setOnMouseEntered(e ->
    okButton.setStyle("-fx-background-color: #faa18d; " + // Sedikit lebih terang
                      "-fx-text-fill: white; " +
                      "-fx-font-weight: bold; " +
                      "-fx-font-size: 14px; " +
                      "-fx-background-radius: 4px;"))

);

okButton.setOnMouseExited(e ->
```

```
        okButton.setStyle("-fx-background-color: " + primaryColor + ",  
" +  
                "-fx-text-fill: white; " +  
                "-fx-font-weight: bold; " +  
                "-fx-font-size: 14px; " +  
                "-fx-background-radius: 4px;")  
    );  
  
    okButton.setOnAction(e -> dialogStage.close());  
  
    root.getChildren().addAll(titleBox, separator, messageText,  
okButton);  
  
    javafx.scene.Scene dialogScene = new javafx.scene.Scene(root, 320,  
180);  
    dialogStage.setScene(dialogScene);  
  
    root.setEffect(new javafx.scene.effect.DropShadow(5,  
javafx.scene.paint.Color.rgb(0, 0, 0, 0.2)));  
  
    javafx.animation.FadeTransition fadeIn = new  
javafx.animation.FadeTransition(javafx.util.Duration.millis(100), root);  
    fadeIn.setFromValue(0.0);  
    fadeIn.setToValue(1.0);  
    dialogStage.setOnShown(e -> fadeIn.play());  
  
    dialogStage.showAndWait();  
});  
}  
  
public void showSuccessDialog(String title, String message) {  
    System.out.println("[SUCCESS DIALOG] " + title + ":" + message);  
  
    javafx.application.Platform.runLater(() -> {  
        javafx.stage.Stage dialogStage = new javafx.stage.Stage();  
        dialogStage.initModality(javafx.stage.Modality.APPLICATION_MODAL);  
        dialogStage.setTitle(title);  
    });  
}
```

```
dialogStage.setResizable(false);

String primaryColor = "#82BF6E";
String textColor = "#5a3e36";
String bgColor = "#ffffff";

javafx.scene.layout.VBox root = new javafx.scene.layout.VBox(15);
root.setPadding(new javafx.geometry.Insets(20));
root.setAlignment(javafx.geometry.Pos.CENTER);
root.setStyle("-fx-background-color: " + bgColor + "; " +
    "-fx-border-color: " + primaryColor + "; " +
    "-fx-border-width: 2px; " +
    "-fx-border-radius: 5px;");

javafx.scene.text.Text titleText = new
javafx.scene.text.Text(title);
    titleText.setFont(javafx.scene.text.Font.font("Arial",
javafx.scene.FontWeight.BOLD, 16));
    titleText.setFill(javafx.scene.paint.Color.web(textColor));

javafx.scene.layout.HBox titleBox = new
javafx.scene.layout.HBox(10);
    titleBox.setAlignment(javafx.geometry.Pos.CENTER);
    titleBox.setPadding(new javafx.geometry.Insets(0, 0, 5, 0));

javafx.scene.layout.StackPane iconPane = new
javafx.scene.layout.StackPane();
    javafx.scene.shape.Circle circle = new
javafx.scene.shape.Circle(12);
    circle.setFill(javafx.scene.paint.Color.web(primaryColor));

javafx.scene.text.Text icon = new javafx.scene.text.Text("✓");
icon.setFont(javafx.scene.text.Font.font("Arial",
javafx.scene.FontWeight.BOLD, 16));
    icon.setFill(javafx.scene.paint.Color.WHITE);

iconPane.getChildren().addAll(circle, icon);
```

```
titleBox.getChildren().addAll(iconPane, titleText);

javafx.scene.shape.Line separator = new javafx.scene.shape.Line();
separator.setStartX(0);
separator.setEndX(380);
separator.setStroke(javafx.scene.paint.Color.web(primaryColor));
separator.setStrokeWidth(1);

javafx.scene.text.Text messageText = new
javafx.scene.text.Text(message);
messageText.setFont(javafx.scene.text.Font.font("Arial", 14));
messageText.setFill(javafx.scene.paint.Color.web(textColor));

messageText.setTextAlignment(javafx.scene.text.TextAlignment.CENTER);
messageText.setWrappingWidth(380);

javafx.scene.control.Button okButton = new
javafx.scene.control.Button("OK");
okButton.setPrefSize(80, 30);
okButton.setStyle("-fx-background-color: " + primaryColor + "; " +
                  "-fx-text-fill: white; " +
                  "-fx-font-weight: bold; " +
                  "-fx-font-size: 14px; " +
                  "-fx-background-radius: 4px;");

okButton.setOnMouseEntered(e ->
    okButton.setStyle("-fx-background-color: #95CA83; " +
                      "-fx-text-fill: white; " +
                      "-fx-font-weight: bold; " +
                      "-fx-font-size: 14px; " +
                      "-fx-background-radius: 4px;"))

);

okButton.setOnMouseExited(e ->
    okButton.setStyle("-fx-background-color: " + primaryColor + ";

" +
                  "-fx-text-fill: white; " +
```

```

        "-fx-font-weight: bold; " +
        "-fx-font-size: 14px; " +
        "-fx-background-radius: 4px;")  

    );  
  

    okButton.setOnAction(e -> dialogStage.close());  
  

    root.getChildren().addAll(titleBox, separator, messageText,  

okButton);  
  

    javafx.scene.Scene dialogScene = new javafx.scene.Scene(root, 420,  

200);  

    dialogStage.setScene(dialogScene);  
  

    root.setEffect(new javafx.scene.effect.DropShadow(5,  

javafx.scene.paint.Color.rgb(0, 0, 0, 0.2)));  
  

    javafx.animation.FadeTransition fadeIn = new  

javafx.animation.FadeTransition(javafx.util.Duration.millis(100), root);  

    fadeIn.setFromValue(0.0);  

    fadeIn.setToValue(1.0);  

    dialogStage.setOnShown(e -> fadeIn.play());  
  

    dialogStage.showAndWait();  

});  

}  
  

public boolean solvePuzzle(String algorithm, String heuristic) {  

    System.out.println("Start solving puzzle with algorithm: " + algorithm  

+  

        ", heuristic: " + (heuristic != null ? heuristic :  

"N/A"));  

    lastUsedAlgorithm = algorithm;  

    lastUsedHeuristic = heuristic;  

    try {  

        debugBoard(currentBoard);  

    }
}

```

```
File tempFile = File.createTempFile("rushHourBoard", ".txt");
tempFile.deleteOnExit();

try (FileWriter writer = new FileWriter(tempFile)) {
    writer.write(currentBoard.length + " " +
currentBoard[0].length + "\n");

    int numNonPrimaryPieces = countNonPrimaryPieces();
    writer.write(numNonPrimaryPieces + "\n");

    for (int i = 0; i < currentBoard.length; i++) {
        writer.write(new String(currentBoard[i]) + "\n");
    }

    writer.write(algorithm + "\n");
    if (!"dijkstra".equals(algorithm.toLowerCase()) &&
        !"ucs".equals(algorithm.toLowerCase()) &&
        heuristic != null) {
        writer.write(heuristic + "\n");
    }
}

System.out.println("Temporary file created: " +
tempFile.getAbsolutePath();

debugReadFile(tempFile);

Board coreBoard = new Board(tempFile.getAbsolutePath());
System.out.println("Core board loaded");

coreBoard.printBoard(null);

GameState solution = null;
String algorithmLower = algorithm.toLowerCase().trim();

System.out.println("Running algorithm: " + algorithmLower);
```

```

    if ("astar".equals(algorithmLower)) {
        System.out.println("Using A* algorithm with " +
                           (heuristic != null ? heuristic : "manhattan") +
                           " heuristic");
        AStar astar = new AStar(heuristic != null ? heuristic :
                           "manhattan");
        solution = astar.solve(coreBoard);
        nodesVisited = astar.getNodesVisited();
        executionTime = (long) astar.getExecutionTime();

    } else if ("dijkstra".equals(algorithmLower)) {
        System.out.println("Using Dijkstra algorithm");
        Dijkstra dijkstra = new Dijkstra();
        solution = dijkstra.solve(coreBoard);
        nodesVisited = dijkstra.getNodesVisited();
        executionTime = (long) dijkstra.getExecutionTime();

    } else if ("ucs".equals(algorithmLower)) {
        System.out.println("Using UCS algorithm");
        UCS ucs = new UCS();
        solution = ucs.solve(coreBoard);
        nodesVisited = ucs.getNodesVisited();
        executionTime = (long) ucs.getExecutionTime();

    } else if ("gbfs".equals(algorithmLower) ||
               "greedy".equals(algorithmLower)) {
        System.out.println("Using GBFS algorithm with " +
                           (heuristic != null ? heuristic : "manhattan") +
                           " heuristic");
        GBFS gbfs = new GBFS(heuristic != null ? heuristic :
                           "manhattan");
        solution = gbfs.solve(coreBoard);
        nodesVisited = gbfs.getNodesVisited();
        executionTime = (long) gbfs.getExecutionTime();

    } else {

```

```

        System.out.println("Unknown algorithm: " + algorithmLower + ".");
Using A* as fallback;

        AStar astarFallback = new AStar("manhattan");
        solution = astarFallback.solve(coreBoard);
        nodesVisited = astarFallback.getNodesVisited();
        executionTime = (long) astarFallback.getExecutionTime();
    }

    if (solution != null) {
        System.out.println("Solution found with " +
solution.getMoves().size() + " moves!");
        processAlgorithmSolution(solution);
        return true;
    } else {
        System.out.println("No solution found!");
        return false;
    }

} catch (Exception e) {
    System.err.println("Error solving puzzle: " + e.getMessage());
    e.printStackTrace();
    return false;
}
}

private void debugBoard(char[][] board) {
    System.out.println("DEBUG: Board state (" + board.length + "x" +
board[0].length + "):");
    for (int i = 0; i < board.length; i++) {
        System.out.println(new String(board[i]));
    }

    boolean foundP = false;
    boolean foundK = false;
    int pRow = -1, pCol = -1;
    int kRow = -1, kCol = -1;
    boolean isHorizontal = false;
}

```

```

        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                if (board[i][j] == 'P') {
                    foundP = true;
                    if (pRow == -1) {
                        pRow = i;
                        pCol = j;
                    }
                    if (j > 0 && j < board[i].length && board[i][j-1] == 'P')
isHorizontal = true;
                    if (j < board[i].length-1 && board[i][j+1] == 'P')
isHorizontal = true;
                }
                if (board[i][j] == 'K') {
                    foundK = true;
                    kRow = i;
                    kCol = j;
                }
            }
        }

        System.out.println("Primary piece (P): " + (foundP ? "Found at [" +
pRow + "," + pCol + "]" : "NOT FOUND!"));
        System.out.println("Primary piece orientation: " + (isHorizontal ?
"Horizontal" : "Vertical"));
        System.out.println("Exit (K): " + (foundK ? "Found at [" + kRow + "," +
+ kCol + "]" : "NOT FOUND!"));

        if (foundP && foundK) {
            boolean validExit = false;
            if (isHorizontal && kRow == pRow) validExit = true;
            if (!isHorizontal && kCol == pCol) validExit = true;

            System.out.println("Exit alignment with primary piece: " +
(validExit ? "Valid" : "INVALID!"));
        }
    }
}

```

```
}

private void debugReadFile(File file) {
    System.out.println("DEBUG: Reading file " + file.getAbsolutePath());
    try (BufferedReader reader = new BufferedReader(new FileReader(file)))
    {
        String line;
        int lineNumber = 0;
        while ((line = reader.readLine()) != null) {
            System.out.println("Line " + (++lineNumber) + ": " + line);
        }
    } catch (IOException e) {
        System.err.println("Error reading file: " + e.getMessage());
    }
}

private int countNonPrimaryPieces() {
    Set<Character> uniquePieces = new HashSet<>();
    for (int i = 0; i < currentBoard.length; i++) {
        for (int j = 0; j < currentBoard[0].length; j++) {
            char piece = currentBoard[i][j];
            if (piece != '.' && piece != 'K' && piece != 'P') {
                uniquePieces.add(piece);
            }
        }
    }
    return uniquePieces.size();
}

private void processAlgorithmSolution(GameState solution) {
    System.out.println("Processing solution with " +
solution.getMoves().size() + " moves");

    solutionSteps.clear();
    currentStepIndex = -1;
```

```

solutionSteps.add(new MoveStep(copyBoard(currentBoard), null, null,
0));

char[][] boardState = copyBoard(currentBoard);
List<Move> moves = solution.getMoves();

for (int i = 0; i < moves.size(); i++) {
    Move move = moves.get(i);
    char piece = move.getPieceId();
    String direction = convertDirectionToGui(move.getDirection());
    int steps = 1;

    try {
        java.lang.reflect.Method getStepsMethod =
move.getClass().getMethod("getSteps");
        steps = (int) getStepsMethod.invoke(move);
    } catch (Exception e) {
        steps = 1;
    }

    System.out.println("Move " + (i+1) + ":" + piece + " " +
direction + " " + steps);

    char[][] newBoardState = copyBoard(boardState);
    applyMoveToBoard(newBoardState, piece, direction, steps);

    solutionSteps.add(new MoveStep(newBoardState, piece, direction,
steps, i + 1));

    boardState = newBoardState;
}

totalMoves = moves.size();
}

private void applyMoveToBoard(char[][] board, char piece, String
direction, int steps) {

```

```
List<int[]> pieceCells = new ArrayList<>();
for (int row = 0; row < board.length; row++) {
    for (int col = 0; col < board[0].length; col++) {
        if (board[row][col] == piece) {
            pieceCells.add(new int[]{row, col});
        }
    }
}

if (pieceCells.isEmpty()) return;

boolean isHorizontal = true;
int firstRow = pieceCells.get(0)[0];
for (int[] cell : pieceCells) {
    if (cell[0] != firstRow) {
        isHorizontal = false;
        break;
    }
}

int minRow = Integer.MAX_VALUE, minCol = Integer.MAX_VALUE;
int maxRow = Integer.MIN_VALUE, maxCol = Integer.MIN_VALUE;

for (int[] cell : pieceCells) {
    minRow = Math.min(minRow, cell[0]);
    minCol = Math.min(minCol, cell[1]);
    maxRow = Math.max(maxRow, cell[0]);
    maxCol = Math.max(maxCol, cell[1]);
}

for (int[] cell : pieceCells) {
    board[cell[0]][cell[1]] = '.';
}

int newMinRow = minRow;
int newMinCol = minCol;
```

```

if (isHorizontal) {
    if (direction.equals("left")) {
        newMinCol = minCol - steps;
    } else if (direction.equals("right")) {
        newMinCol = minCol + steps;
    }
} else {
    if (direction.equals("up")) {
        newMinRow = minRow - steps;
    } else if (direction.equals("down")) {
        newMinRow = minRow + steps;
    }
}

int pieceHeight = maxRow - minRow + 1;
int pieceWidth = maxCol - minCol + 1;

for (int i = 0; i < pieceHeight; i++) {
    for (int j = 0; j < pieceWidth; j++) {
        if (newMinRow + i >= 0 && newMinRow + i < board.length &&
            newMinCol + j >= 0 && newMinCol + j < board[0].length) {
            board[newMinRow + i][newMinCol + j] = piece;
        }
    }
}
}

private void displayMove(MoveStep step) {
    boardPane.updateBoard(step.board);
}

private String convertDirectionToGui(String algorithmDirection) {
    switch (algorithmDirection.toLowerCase()) {
        case "atas": return "up";
        case "bawah": return "down";
        case "kiri": return "left";
    }
}

```

```
        case "kanan": return "right";
    default: return algorithmDirection;
}
}

public void showPreviousMove() {
    if (currentStepIndex > 0) {
        currentStepIndex--;
        displayMove(solutionSteps.get(currentStepIndex));
    }
}

public void showNextMove() {
    if (currentStepIndex < solutionSteps.size() - 1) {
        currentStepIndex++;
        displayMove(solutionSteps.get(currentStepIndex));
    } else {
        animation.stop();
    }
}

public void playAnimation() {
    currentStepIndex = 0;
    displayMove(solutionSteps.get(0));

    animation.play();
}

public void stopAnimation() {
    animation.stop();
}

public void jumpToStep(int stepIndex) {
    if (stepIndex >= 0 && stepIndex < solutionSteps.size()) {
        currentStepIndex = stepIndex;
    }
}
```

```
        displayMove(solutionSteps.get(currentStepIndex));

    }

}

public class MoveStep {
    char[][] board;
    Character piece;
    String direction;
    int steps = 1;
    int moveNumber;

    MoveStep(char[][] board, Character piece, String direction, int moveNumber) {
        this(board, piece, direction, 1, moveNumber);
    }

    MoveStep(char[][] board, Character piece, String direction, int steps,
            int moveNumber) {
        this.board = board;
        this.piece = piece;
        this.direction = direction;
        this.steps = steps;
        this.moveNumber = moveNumber;
    }

    private char[][] copyBoard(char[][] original) {
        if (original == null) return null;
        char[][] copy = new char[original.length][];
        for (int i = 0; i < original.length; i++) {
            copy[i] = original[i].clone();
        }
        return copy;
    }

    public void saveSolutionToFile(File file) {
```

```
if (solutionSteps.isEmpty() || solutionSteps.size() <= 1) {
    showAlertDialog("Save Error", "No solution available to save.");
    return;
}

try (FileWriter writer = new FileWriter(file)) {
    // Write header with fancy formatting
    writer.write("[" + "Rush Hour Puzzle Solution" + "]\n");
    writer.write("[" + "Rush Hour Puzzle Solution" + "]\n");
    writer.write("[" + "Rush Hour Puzzle Solution" + "]\n\n");

    // Write solution information
    writer.write("Solution Details:\n");
    writer.write("-----\n");
    writer.write(String.format("Algorithm: %s\n",
formatAlgorithmName(lastUsedAlgorithm)));
    writer.write(String.format("Heuristic: %s\n",
formatHeuristicName(lastUsedHeuristic)));
    writer.write(String.format("Total Moves: %d\n", totalMoves));
    writer.write(String.format("Nodes Visited: %d\n", nodesVisited));
    writer.write(String.format("Execution Time: %d ms\n\n",
executionTime));

    // Write initial state
    writer.write("Initial State:\n");
    char[][] initialBoard = solutionSteps.get(0).board;
    writeBoardWithBorder(writer, initialBoard);
    writer.write("\n");

    // Write each move
    for (int i = 1; i < solutionSteps.size(); i++) {
        MoveStep step = solutionSteps.get(i);
        String directionName = formatDirectionName(step.direction);

        writer.write(String.format("Move %d: %s-%s",
step.moveNumber,
step.piece,
step.direction));
    }
}
```

```

        directionName);

    if (step.steps > 1) {
        writer.write(String.format(" %d step(s)", step.steps));
    }
    writer.write("\n");

    writeBoardWithBorder(writer, step.board);
    writer.write("\n");
}

// Write footer
writer.write("End of solution\n");
writer.write("Generated on: " + java.time.LocalDateTime.now() +
"\n");

System.out.println("Solution saved to: " +
file.getAbsolutePath());

// Show success dialog using the custom dialog instead of Alert
javafx.application.Platform.runLater(() -> {
    showSuccessDialog("Success", "Solution successfully saved
to:\n" + file.getAbsolutePath());
});

} catch (IOException e) {
    System.err.println("Error saving solution: " + e.getMessage());
    showErrorDialog("Save Error", "Failed to save solution: " +
e.getMessage());
}
}

private void writeBoardWithBorder(FileWriter writer, char[][] board)
throws IOException {
    int cols = board[0].length;

    // Top border

```

```

writer.write("┌");
for (int j = 0; j < cols; j++) {
    writer.write("─");
}
writer.write("┐\n");

// Board with left and right borders
for (int i = 0; i < board.length; i++) {
    writer.write("│ ");
    writer.write(new String(board[i]));
    writer.write("│\n");
}

// Bottom border
writer.write("└");
for (int j = 0; j < cols; j++) {
    writer.write("─");
}
writer.write("┘\n");
}

/**
 * Formats the algorithm name for display
 */
private String formatAlgorithmName(String algorithm) {
    if (algorithm == null) return "Unknown";

    switch (algorithm.toLowerCase()) {
        case "astar": return "A* (A-Star)";
        case "dijkstra": return "Dijkstra's Algorithm";
        case "gbfs": return "Greedy Best-First Search";
        case "ucs": return "Uniform Cost Search";
        default: return algorithm;
    }
}

/**

```

```
* Formats the heuristic name for display
*/
private String formatHeuristicName(String heuristic) {
    if (heuristic == null) return "None";

    switch (heuristic.toLowerCase()) {
        case "manhattan": return "Manhattan Distance";
        case "blocking": return "Blocking Heuristic";
        case "combined": return "Combined Heuristic";
        default: return heuristic;
    }
}

/**
 * Formats the direction name for display
*/
private String formatDirectionName(String direction) {
    if (direction == null) return "Unknown";

    switch (direction.toLowerCase()) {
        case "up": return "Up";
        case "down": return "Down";
        case "left": return "Left";
        case "right": return "Right";
        default: return direction;
    }
}

public int getTotalMoves() {
    return totalMoves;
}

public int getNodesVisited() {
    return nodesVisited;
}

public long getExecutionTime() {
```

```

        return executionTime;
    }

    public void loadPuzzleFromFile(String filePath) {
        loadPuzzleFromFile(new File(filePath));
    }

    public int getCurrentStepIndex() {
        return currentStepIndex;
    }

    public int getTotalSteps() {
        return solutionSteps.size();
    }
}

```

Attribute	Tipe Data	Keterangan
boardPane	private BoardPane	Panel tampilan papan permainan
currentBoard	private char[][]	Matriks papan saat ini
aboutButton	private Button	Atribut tombol (<i>button</i>) about
solutionSteps	private List<MoveStep>	Langkah-langkah solusi dari algoritma
currentStepIndex	private int	Indeks langkah solusi yang sedang ditampilkan
animation	private Timeline	Animasi untuk memutar langkah satu per satu
lastUsedAlgorithm	private String	Algoritma terakhir yang digunakan
lastUsedHeuristic	private String	Heuristic terakhir yang digunakan
totalMoves	private int	Jumlah langkah dalam solusi

nodesVisited	private int	Total node yang dikunjungi selama pencarian
executionTime	private long	Lama waktu eksekusi algoritma (ms)

Method	Returned Data Type	Keterangan
Renderer(BoardPane)	Constructor	Konstruktor, inisialisasi board dan animasi
loadPuzzleFromFile(File)	void	Memuat dan memvalidasi puzzle dari file
loadPuzzleFromFile(String)	void	Versi overload dengan path string
showErrorDialog(String, String)	void	Menampilkan dialog kesalahan
showSuccessDialog(String, String)	void	Menampilkan dialog sukses simpan solusi
solvePuzzle(String, String)	boolean	Menjalankan algoritma dan memproses hasil solusi
processAlgorithmSolution(GameState)	void	Memasukkan solusi ke dalam langkah-langkah GUI
applyMoveToBoard(char[][], char, String, int)	void	Menerapkan langkah ke papan
displayMove(MoveStep)	void	Menampilkan langkah tertentu di GUI
showNextMove()	void	Pindah ke langkah selanjutnya
showPreviousMove()	void	Kembali ke langkah sebelumnya
jumpToStep(int)	void	Melompat ke langkah tertentu
playAnimation()	void	Memulai animasi langkah solusi
stopAnimation()	void	Menghentikan animasi solusi

saveSolutionToFile(File)	void	Menyimpan langkah solusi ke file .txt
countNonPrimaryPieces()	int	Menghitung jumlah unik pieces selain P dan K
copyBoard(char[][])	char[][]	Membuat salinan papan
debugBoard(char[][])	void	Menampilkan info debug papan
debugReadFile(File)	void	Menampilkan isi file untuk debug
convertDirectionToGui(String)	String	Mengubah arah dari algoritma ke GUI
formatAlgorithmName(String)	String	Format nama algoritma untuk tampil di file
formatHeuristicName(String)	String	Format nama heuristic untuk tampil di file
formatDirectionName(String)	String	Format nama arah langkah untuk tampil di file
getTotalMoves()	int	Getter total langkah solusi
getNodesVisited()	int	Getter jumlah simpul dikunjungi
getExecutionTime()	long	Getter lama eksekusi algoritma
getCurrentStepIndex()	int	Getter indeks langkah saat ini
getTotalSteps()	int	Getter jumlah langkah total dalam solusi

3.15 WelcomePage.java

Berikut ini merupakan attribute dan method yang terdapat dalam class WelcomePage.java

```
package gui;

import java.io.File;
```

```
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;

public class WelcomePage extends VBox {

    private Button startButton;
    private Button creatorButton;
    private Button aboutButton;

    public WelcomePage() {
        this.setSpacing(20);
        this.setAlignment(Pos.CENTER);
        this.setPadding(new Insets(30));
        this.getStyleClass().add("welcome-page");

        ImageView logoView = createLogoImageView();

        startButton = createStyledButton("Start the Engine", "start-button");
        creatorButton = createStyledButton("Creators", "nav-button");
        aboutButton = createStyledButton("About", "nav-button");

        this.getChildren().addAll(
            logoView,
            startButton,
            creatorButton,
            aboutButton
        );
    }

    private ImageView createLogoImageView() {
        try {
            File imageFile = new File("resources/images/car.png");
            if (!imageFile.exists()) {
```

```
        System.err.println("File tidak ditemukan: " +
imageFile.getAbsolutePath());
            throw new Exception("Logo file not found");
        }

        Image logoImage = new Image(imageFile.toURI().toString());
        ImageView logoView = new ImageView(logoImage);
        logoView.setFitWidth(280);
        logoView.setPreserveRatio(true);
        return logoView;
    } catch (Exception e) {
        System.err.println("Error loading image: " + e.getMessage());
        return null;
    }
}

private Button createStyledButton(String text, String styleClass) {
    Button button = new Button(text);
    button.getStyleClass().add(styleClass);
    button.setPrefWidth(200);
    button.setPrefHeight(40);
    return button;
}

// Getters
public Button getStartButton() {
    return startButton;
}

public Button getCreatorButton() {
    return creatorButton;
}

public Button getAboutButton() {
    return aboutButton;
}

}
```

Attribute	Tipe Data	Keterangan
startButton	private Button	Atribut tombol (<i>button</i>) start
creatorButton	private Button	Atribut tombol (<i>button</i>) creator
aboutButton	private Button	Atribut tombol (<i>button</i>) about

Method	Returned Data Type	Keterangan
WelcomePage	-	Konstruktor
createLogoImageView	ImageView	Metode untuk memuat dan menampilkan logo gambar utama sebagai ImageView
createStyledButton	Button	Metode untuk membuat tombol dengan teks dan gaya yang telah disesuaikan
getStartButton	Button	Metode untuk mengembalikan <i>start button</i>
getCreatorButton	Button	Metode untuk mengembalikan <i>creator button</i>
getAboutButton	Button	Metode untuk mengembalikan <i>about button</i>

3.16 App.java

Berikut ini merupakan attribute dan method yang terdapat dalam class App.java

```
package gui;

import javafx.animation.FadeTransition;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.geometry.Insets;
import javafx.scene.Node;
import javafx.scene.Scene;
```

```
import javafx.scene.control.Menu;
import javafx.scene.controlMenuBar;
import javafx.scene.controlMenuItem;
import javafx.scene.controlSeparatorMenuItem;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.util.Duration;

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

public class App extends Application {

    private enum PageType {
        WELCOME, GAME, CREATOR, ABOUT
    }

    private StackPane rootContainer;

    private BoardPane boardPane;
    private ControlPanel controlPanel;
    private Renderer renderer;
    private BorderPane gameRoot;

    private Map<PageType, Node> pages = new HashMap<>();
    private PageType currentPage = null;

    @Override
    public void start(Stage primaryStage) {
        rootContainer = new StackPane();

        setupGameComponents();

        setupPages();

        navigateTo(PageType.WELCOME);
    }
}
```

```
Scene scene = new Scene(rootContainer, 1200, 800);

scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());
}

primaryStage.setTitle("Rush Hour");
primaryStage.setScene(scene);
primaryStage.setResizable(true);
primaryStage.show();
}

private void setupGameComponents() {
    boardPane = new BoardPane();
    renderer = new Renderer(boardPane);
    controlPanel = new ControlPanel(renderer);

    BorderPane boardContainer = new BorderPane();
    boardContainer.setCenter(boardPane);
    boardContainer.setPrefWidth(Double.MAX_VALUE);
    boardContainer.setPrefHeight(Double.MAX_VALUE);

    boardPane.setStyle("-fx-fit-to-width: true;");

    gameRoot = new BorderPane();

    StackPane controlContainer = new StackPane();
    controlContainer.getChildren().add(controlPanel);

    controlContainer.setPadding(new Insets(0, 15, 11, 0));

    controlPanel.setMinHeight(controlPanel.getPrefHeight());
    controlPanel.setMaxHeight(controlPanel.getPrefHeight());
    controlPanel.setMinWidth(controlPanel.getPrefWidth());
    controlPanel.setMaxWidth(controlPanel.getPrefWidth());
}
```

```
controlPanel.setStyle("-fx-fit-to-height: false; -fx-fit-to-width: false;");

BorderPane centerContent = new BorderPane();
centerContent.setCenter(boardContainer);
centerContent.setRight(controlContainer);

gameRoot.setCenter(centerContent);
gameRoot.getStyleClass().add("game-root");

setupMenuBar();
}

private void setupMenuBar() {
    MenuBar menuBar = new MenuBar();

    menuBar.setStyle(null);
    menuBar.getStyleClass().clear();
    menuBar.getStyleClass().add("menu-bar");

    Menu fileMenu = new Menu("File");
    fileMenu.getStyleClass().add("custom-menu");

    MenuItem welcomeItem = new MenuItem("Main Menu");
    welcomeItem.getStyleClass().add("custom-menu-item");

    MenuItem exitItem = new MenuItem("Exit");
    exitItem.getStyleClass().add("custom-menu-item");

    welcomeItem.setOnAction(e -> navigateTo(PageType.WELCOME));
    exitItem.setOnAction(e -> Platform.exit());

    fileMenu.getItems().addAll(welcomeItem, new SeparatorMenuItem(),
        exitItem);

    Menu viewMenu = new Menu("View");
    viewMenu.getStyleClass().add("custom-menu");
```

```
MenuItem creatorItem = new MenuItem("Creators");
creatorItem.getStyleClass().add("custom-menu-item");

MenuItem aboutItem = new MenuItem("About");
aboutItem.getStyleClass().add("custom-menu-item");

creatorItem.setOnAction(e -> navigateTo(PageType.CREATOR));
aboutItem.setOnAction(e -> navigateTo(PageType.ABOUT));

viewMenu.getItems().addAll(creatorItem, aboutItem);

menuBar.getMenus().addAll(fileMenu, viewMenu);

BorderPane menuContainer = new BorderPane();
menuContainer.setTop(menuBar);
menuContainer.setStyle("-fx-background-color: #725861;");

gameRoot.setTop(menuContainer);

BorderPane.setMargin(menuContainer, new Insets(0, 0, 15, 0));
}

private void setupPages() {
    WelcomePage welcomePage = new WelcomePage();
    welcomePage.getStartButton().setOnAction(e ->
navigateTo(PageType.GAME));
    welcomePage.getCreatorButton().setOnAction(e ->
navigateTo(PageType.CREATOR));
    welcomePage.getAboutButton().setOnAction(e ->
navigateTo(PageType.ABOUT));
}

CreatorPage creatorPage = new CreatorPage();
creatorPage.getBackButton().setOnAction(e ->
navigateTo(PageType.WELCOME));

AboutPage aboutPage = new AboutPage();
```

```
        aboutPage.getBackButton().setOnAction(e ->
navigateTo(PageType.WELCOME)) ;

        pages.put(PageType.WELCOME, welcomePage);
        pages.put(PageType.GAME, gameRoot);
        pages.put(PageType.CREATOR, creatorPage);
        pages.put(PageType.ABOUT, aboutPage);

    }

private void navigateTo(PageType pageType) {
    if (currentPage == pageType) return;

    Node oldContent = currentPage != null ? pages.get(currentPage) : null;
    Node newContent = pages.get(pageType);

    if (oldContent != null) {
        switchContent(oldContent, newContent);
    } else {
        rootContainer.getChildren().clear();
        newContent.setOpacity(1.0);
        rootContainer.getChildren().add(newContent);
    }

    currentPage = pageType;
}

private void switchContent(Node oldContent, Node newContent) {
    Duration duration = Duration.millis(300);

    FadeTransition fadeOut = new FadeTransition(duration, oldContent);
    fadeOut.setFromValue(1.0);
    fadeOut.setToValue(0.0);

    newContent.setOpacity(0.0);

    if (!rootContainer.getChildren().contains(newContent)) {
        rootContainer.getChildren().add(newContent);
    }
}
```

```

    }

    FadeTransition fadeIn = new FadeTransition(duration, newContent);
    fadeIn.setFromValue(0.0);
    fadeIn.setToValue(1.0);

    fadeOut.setOnFinished(e -> {
        rootContainer.getChildren().remove(oldContent);
        fadeIn.play();
    });

    fadeOut.play();
}

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

```

Attribute	Tipe Data	Keterangan
rootContainer	private StackPane	Kontainer utama yang menampung semua halaman
boardPane	private BoardPane	Komponen visual untuk menampilkan papan permainan
controlPanel	private ControlPanel	Panel kontrol berisi tombol dan input user
renderer	private Renderer	Komponen logika untuk memproses solusi dan board
gameRoot	private BorderPane	Root layout khusus untuk halaman permainan (game screen)
pages	private Map<PageType, Node>	Menyimpan semua halaman berdasarkan enum PageType
currentPage	private PageType	Halaman saat ini yang sedang

		aktif
--	--	-------

Method	Returned Data Type	Keterangan
start(Stage primaryStage)	void	Entry point JavaFX; inisialisasi scene dan tampilan awal
setupGameComponents()	void	Menyusun komponen board, panel kontrol, dan layout game utama
setupMenuBar()	void	Membuat dan mengatur menu navigasi (File & View)
setupPages()	void	Mengatur dan menyimpan instance dari semua halaman
navigateTo(PageType)	void	Berpindah ke halaman tertentu (dengan transisi jika ada halaman lama)
switchContent(Node, Node)	void	Efek transisi fade antar halaman lama dan baru
main(String[] args)	static void	Metode main untuk menjalankan aplikasi JavaFX

3.17 SaveSolutionHandler.java

Berikut ini merupakan attribute dan method yang terdapat dalam class SaveSolutionHandler.java

```
package gui;

import javafx.scene.control.Button;
import javafx.stage.FileChooser;

import java.io.File;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class SaveSolutionHandler {
```

```
private final Renderer renderer;
private final Button saveSolutionButton;
private boolean puzzleSolved = false;
private boolean fileLoaded = false;

public SaveSolutionHandler(Renderer renderer) {
    this.renderer = renderer;
    this.saveSolutionButton = createSaveSolutionButton();
    updateButtonState();
}

private Button createSaveSolutionButton() {
    Button button = new Button("Save Solution");
    button.getStyleClass().add("start-button");
    button.setId("saveSolutionButton");
    button.setMaxWidth(Double.MAX_VALUE);
    button.setOnAction(e -> handleSaveSolution());
    button.setDisable(true);
    return button;
}

public Button getSaveSolutionButton() {
    return saveSolutionButton;
}

private void handleSaveSolution() {
    if (!puzzleSolved || renderer.getTotalSteps() <= 1) {
        renderer.showErrorDialog("Save Error", "No solution available to
save.");
        return;
    }

    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Save Solution");
    fileChooser.getExtensionFilters().add(
        new FileChooser.ExtensionFilter("Text Files", "*.txt")
    );
}
```

```

String dateStr =
LocalDate.now().format(DateTimeFormatter.ISO_LOCAL_DATE);

String initialFileName = "rushhour_solution_" + dateStr + ".txt";
fileChooser.setInitialFileName(initialFileName);

File file =
fileChooser.showSaveDialog(saveSolutionButton.getScene().getWindow());

if (file != null) {
    renderer.saveSolutionToFile(file);
}
}

public void setFileLoaded(boolean loaded) {
    this.fileLoaded = loaded;
    if (!loaded) {
        this.puzzleSolved = false;
    }
    updateButtonState();
}

public void setPuzzleSolved(boolean solved) {
    this.puzzleSolved = solved;
    updateButtonState();
}

private void updateButtonState() {
    boolean shouldEnable = fileLoaded && puzzleSolved;
    saveSolutionButton.setDisable(!shouldEnable);
}
}

```

Attribute	Tipe Data	Keterangan
renderer	private final Renderer	Atribut handler tampilan dan logika game
saveSolutionButton	private final Button	Atribut tombol untuk

		menyimpan solusi
puzzleSolved	private boolean	Atribut status apakah puzzle sudah terselesaikan
fileLoaded	private boolean	Atribut status apakah file puzzle sudah dimuat

Method	Returned Data Type	Keterangan
SaveSolutionHandler(Renderer)	-	Konstruktor untuk inisialisasi renderer dan tombol save
createSaveSolutionButton()	Button	Metode untuk membuat tombol save solution dengan styling dan aksi klik
getSaveSolutionButton()	Button	Metode untuk mengembalikan tombol save solution agar dapat ditambahkan ke GUI
handleSaveSolution()	void	Metode untuk menyimpan solusi .txt jika puzzle sudah diselesaikan
setFileLoaded(boolean)	void	Metode untuk menandai apakah file konfigurasi sudah dimuat
setPuzzleSolved(boolean)	void	Metode untuk menandai apakah solusi puzzle sudah ditemukan
updateButtonState()	void	Metode untuk mengaktifkan/menonaktifkan tombol save solution berdasarkan status file dan solusi

3.18 Main.java

Berikut ini merupakan attribute dan method yang terdapat dalam class Main.java

```
import java.io.BufferedReader;
import java.io.FileReader;
```

```
import java.io.IOException;
import core.Board;
import core.GameState;
import algorithm.AStar;
import algorithm.Dijkstra;
import algorithm.GBFS;
import algorithm.UCS;

public class Main {
    private static final String[] VALID_ALGORITHMS = {"astar", "dijkstra",
"gbfs", "ucs"};
    private static final String[] VALID_HEURISTICS = {"manhattan", "blocking",
"combined"};

    public static void main(String[] args) {
        try {
            if (args.length > 0) {
                runPuzzleFromFile(args[0]);
            } else {
                runPuzzleFromFile("test/test1.txt");
            }
        } catch (IOException e) {
            System.out.println("Error loading board: " + e.getMessage());
            e.printStackTrace();
        }
    }

    private static void runPuzzleFromFile(String filepath) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filepath));

        StringBuilder boardConfigBuilder = new StringBuilder();

        String line;
        String algorithm = null;
        String heuristic = null;
        int lineCount = 0;
        int numRows = 0;
```

```
while ((line = reader.readLine()) != null) {
    lineCount++;

    if (lineCount == 1) {
        String[] parts = line.trim().split("\\s+");
        if (parts.length >= 2) {
            numRows = Integer.parseInt(parts[0]);
        }
        boardConfigBuilder.append(line).append("\n");
    } else if (lineCount == 2) {
        boardConfigBuilder.append(line).append("\n");
    } else if (lineCount <= numRows + 2) {
        boardConfigBuilder.append(line).append("\n");
    } else if (algorithm == null) {
        algorithm = line.trim().toLowerCase();
    } else if (heuristic == null) {
        heuristic = line.trim().toLowerCase();
    }
}
reader.close();

boolean validAlgorithm = false;
for (String validAlg : VALID_ALGORITHMS) {
    if (validAlg.equals(algorithm)) {
        validAlgorithm = true;
        break;
    }
}

if (!validAlgorithm) {
    algorithm = "astar";
    System.out.println("Invalid or no algorithm specified, using A* as default.");
}

boolean validHeuristic = false;
for (String validHeur : VALID_HEURISTICS) {
    if (validHeur.equals(heuristic)) {
```

```
        validHeuristic = true;
        break;
    }
}

if (!validHeuristic) {
    heuristic = "manhattan";
    System.out.println("Invalid or no heuristic specified, using
Manhattan distance as default.");
}

if (algorithm.equals("ucs")) {
    System.out.println("Note: UCS does not use heuristic functions,
ignoring heuristic setting.");
}

java.io.File tempFile = java.io.File.createTempFile("rushHourBoard",
".txt");
tempFile.deleteOnExit();

java.io.FileWriter writer = new java.io.FileWriter(tempFile);
writer.write(boardConfigBuilder.toString());
writer.close();

Board board = new Board(tempFile.getAbsolutePath());

System.out.println("==> Puzzle Configuration ==>");
System.out.println("Algorithm: " + algorithm);
if (!algorithm.equals("ucs")) {
    System.out.println("Heuristic: " + heuristic);
}
System.out.println("\nInitial Board:");
board.printBoard(null);

runAlgorithm(algorithm, heuristic, board);
}
```

```
    private static void runAlgorithm(String algorithm, String heuristic, Board
board) {
        if (!algorithm.equals("ucs")) {
            System.out.println("\n==== Running " + algorithm.toUpperCase() + " "
with " + heuristic + " heuristic ===");
        } else {
            System.out.println("\n==== Running " + algorithm.toUpperCase() + " "
===");
        }
    }

    GameState solution = null;

    switch (algorithm) {
        case "astar":
            AStar astar = new AStar(heuristic);
            solution = astar.solve(board);

            if (solution != null) {
                System.out.println("Solution found with " +
solution.getMoves().size() + " steps");
                System.out.println("Nodes visited: " +
astar.getNodesVisited());
                System.out.println("Execution time: " +
astar.getExecutionTime());
                System.out.println("\nSolution path:");
                astar.printSolution(solution);
            } else {
                System.out.println("No solution found!");
                System.out.println("Nodes visited: " +
astar.getNodesVisited());
                System.out.println("Execution time: " +
astar.getExecutionTime());
            }
            break;

        case "dijkstra":
            Dijkstra dijkstra = new Dijkstra();
            solution = dijkstra.solve(board);
    }
}
```

```
        if (solution != null) {
            System.out.println("Solution found with " +
solution.getMoves().size() + " steps");
            System.out.println("Nodes visited: " +
dijkstra.getNodesVisited());
            System.out.println("Execution time: " +
dijkstra.getExecutionTime());
            System.out.println("\nSolution path:");
            dijkstra.printSolution(solution);
        } else {
            System.out.println("No solution found!");
            System.out.println("Nodes visited: " +
dijkstra.getNodesVisited());
            System.out.println("Execution time: " +
dijkstra.getExecutionTime());
        }
        break;

    case "gbfs":
        GBFS gbfs = new GBFS(heuristic);
        solution = gbfs.solve(board);

        if (solution != null) {
            System.out.println("Solution found with " +
solution.getMoves().size() + " steps");
            System.out.println("Nodes visited: " +
gbfs.getNodesVisited());
            System.out.println("Execution time: " +
gbfs.getExecutionTime());
            System.out.println("\nSolution path:");
            gbfs.printSolution(solution);
        } else {
            System.out.println("No solution found!");
            System.out.println("Nodes visited: " +
gbfs.getNodesVisited());
            System.out.println("Execution time: " +
gbfs.getExecutionTime());
        }
    }
}
```

```

    }

    break;

case "ucs":
    UCS ucs = new UCS();
    solution = ucs.solve(board);

    if (solution != null) {
        System.out.println("Solution found with " +
solution.getMoves().size() + " steps");
        System.out.println("Nodes visited: " +
ucss.getNodesVisited());
        System.out.println("Execution time: " +
ucss.getExecutionTime());
        System.out.println("\nSolution path:");
        ucs.printSolution(solution);
    } else {
        System.out.println("No solution found!");
        System.out.println("Nodes visited: " +
ucss.getNodesVisited());
        System.out.println("Execution time: " +
ucss.getExecutionTime());
    }
    break;

default:
    System.out.println("Unknown algorithm: " + algorithm);
    System.out.println("Available algorithms: astar, dijkstra,
gbfs, ucs");
    break;
}
}
}

```

Attribute	Tipe Data	Keterangan
VALID_ALGORITHMS	private static final String[]	Atribut nama algoritma yang valid

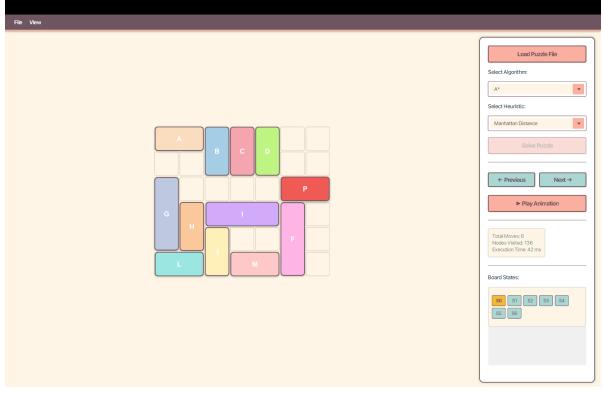
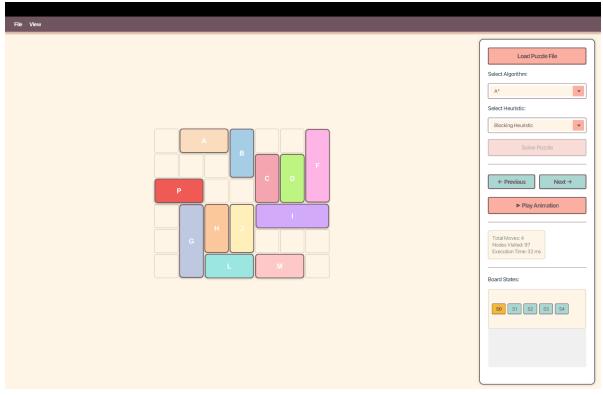
VALID_HEURISTICS	private static final String[]	Atribut nama heuristics yang valid
------------------	-------------------------------	------------------------------------

Method	Returned Data Type	Keterangan
main	-	Metode program utama
runPuzzleFromFile	-	Metode untuk menjalankan puzzle dari file
runAlgorithm	-	Metode untuk menjalankan algoritma

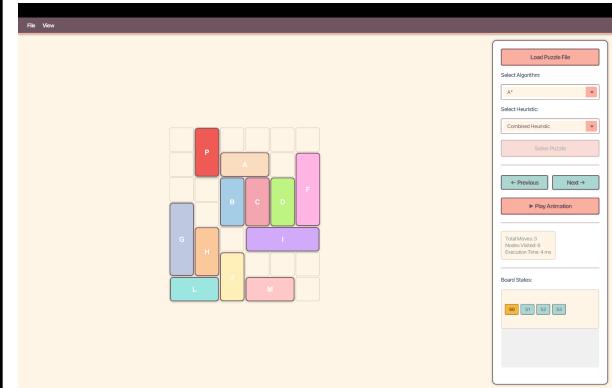
BAB 4

Pengujian

3.1. Algoritma A*

Input	Output
Pintu keluar di sebelah kanan <pre>test > ≡ test1.txt 1 6 6 2 11 3 AAB..F 4 ..BCDF 5 GPPCDFK 6 GH.III 7 GHJ... 8 LLJMM.</pre>	Heuristic: Manhattan Distance 
Pintu keluar di sebelah kiri <pre>test > ≡ test2.txt 1 6 6 2 11 3 AAB..F 4 ..BCDF 5 KGPPCDF 6 GH.III 7 GHJ... 8 LLJMM.</pre>	Heuristic: Blocking 
Pintu keluar di atas	Heuristic: Combined

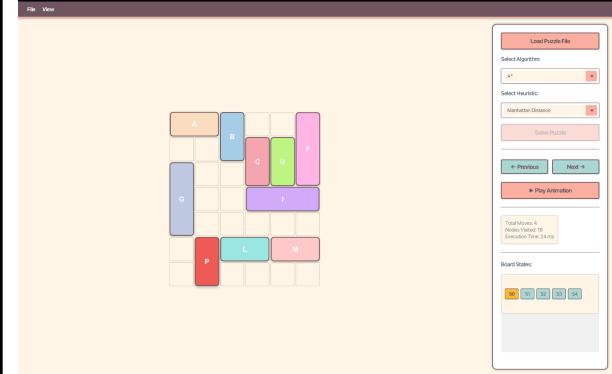
```
test > test3.txt
1 6 6
2 11
3 | K
4 AAB..F
5 .PBCDF
6 GP.CDF
7 GH.III
8 GHJ...
9 LLJMM.
```



Pintu keluar di bawah

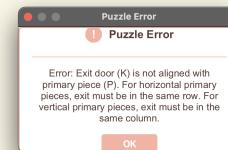
```
test > test4.txt
1 6 6
2 11
3 AAB..F
4 .PBCDF
5 GP.CDF
6 GH.III
7 GHJ...
8 LLJMM.
9 K
```

Heuristic: Manhattan Distance



Pintu keluar tidak searah dengan bentuk primary piece

```
test > test5.txt
1 6 6
2 11
3 AAB..F
4 .PBCDF
5 GP.CDFK
6 GH.III
7 GHJ...
8 LLJMM.
```



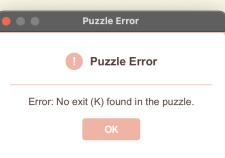
Pintu keluar ada 2

```
test > ≡ test6.txt
1   6 6
2   11
3   K
4   AAB..F
5   .PBCDF
6   GP.CDF
7   GH.III
8   GHJ...
9   LLJMM.
10  K
```



Tidak ada pintu keluar

```
test > ≡ test7.txt
1   6 6
2   11
3   AAB..F
4   .PBCDF
5   GP.CDF
6   GH.III
7   GHJ...
8   LLJMM.
```

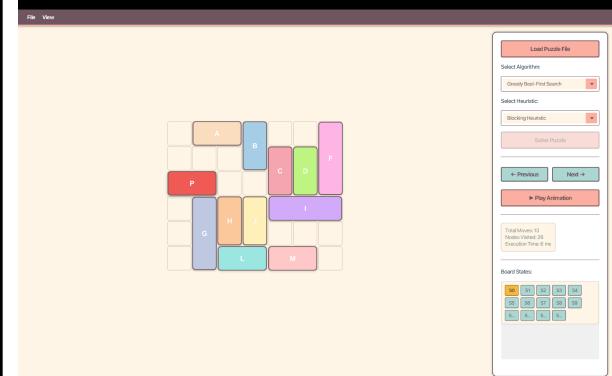


3.2. Algoritma Greedy Best First Search (GBFS)

Input	Output
Pintu keluar di sebelah kanan	Heuristic: Manhattan Distance
Pintu keluar di sebelah kiri	Heuristic: Blocking

```
test > ≡ test2.txt
```

```
1   6 6
2   11
3   AAB..F
4   ..BCDF
5   KGPPCDF
6   GH.III
7   GHJ...
8   LLJMM.
```

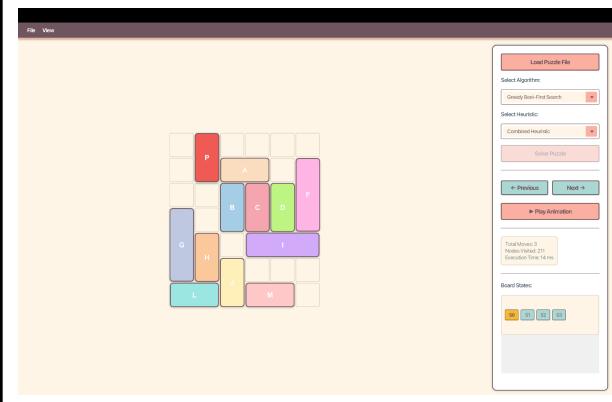


Pintu keluar di atas

```
test > ≡ test3.txt
```

```
1   6 6
2   11
3   K
4   AAB..F
5   .PBCDF
6   GP.CDF
7   GH.III
8   GHJ...
9   LLJMM.
```

Heuristic: Combined

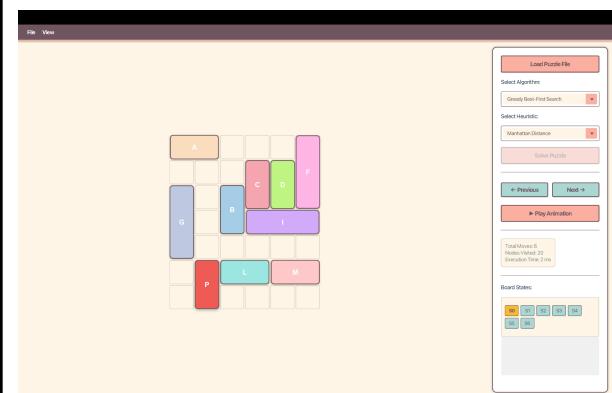


Pintu keluar di bawah

```
test > ≡ test4.txt
```

```
1   6 6
2   11
3   AAB..F
4   .PBCDF
5   GP.CDF
6   GH.III
7   GHJ...
8   LLJMM.
9   K
```

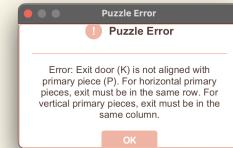
Heuristic: Manhattan Distance



Pintu keluar tidak searah dengan bentuk primary piece

```
test > ≡ test5.txt
```

```
1 6 6
2 11
3 AAB..F
4 .PBCDF
5 GP.CDFK
6 GH.III
7 GHJ...
8 LLJMM.
```



Pintu keluar ada 2

```
test > ≡ test6.txt
```

```
1 6 6
2 11
3 K
4 AAB..F
5 .PBCDF
6 GP.CDF
7 GH.III
8 GHJ...
9 LLJMM.
10 K
```



Tidak ada pintu keluar

```
test > ≡ test7.txt
```

```
1 6 6
2 11
3 AAB..F
4 .PBCDF
5 GP.CDF
6 GH.III
7 GHJ...
8 LLJMM.
```

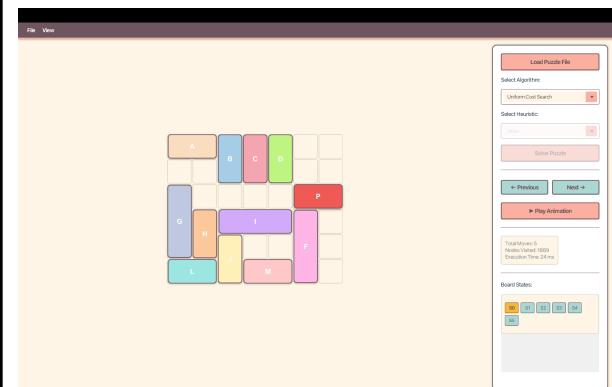


3.3. Algoritma Uniform Cost Search (UCS)

Input	Output
-------	--------

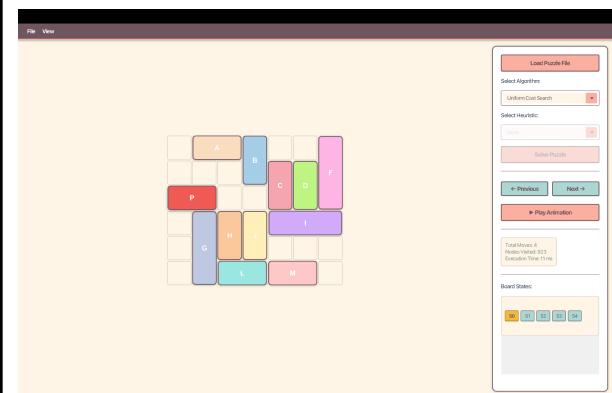
Pintu keluar di sebelah kanan

```
test > ≡ test1.txt
1   6 6
2   11
3   AAB..F
4   ..BCDF
5   GPPCDFK
6   GH.III
7   GHJ...
8   LLJMM.
```



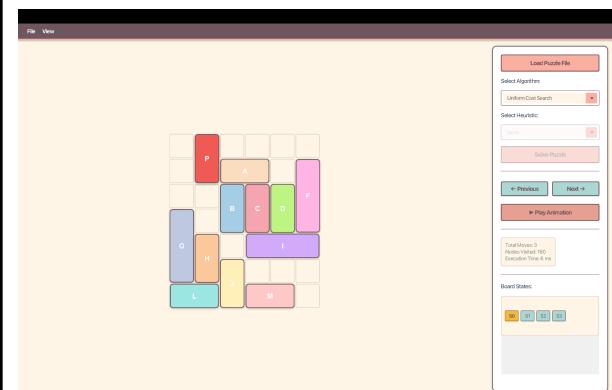
Pintu keluar di sebelah kiri

```
test > ≡ test2.txt
1   6 6
2   11
3   AAB..F
4   ..BCDF
5   KGPPCDF
6   GH.III
7   GHJ...
8   LLJMM.
```



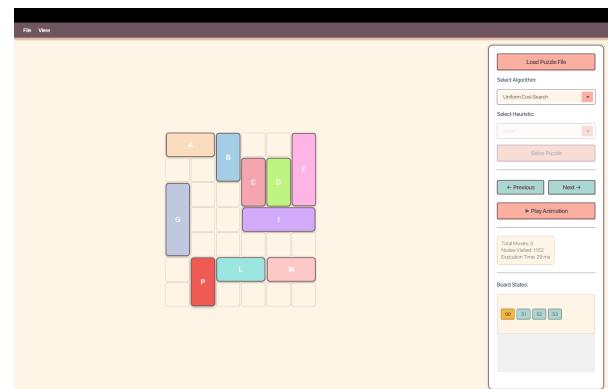
Pintu keluar di atas

```
test > ≡ test3.txt
1   6 6
2   11
3   K
4   AAB..F
5   .PBCDF
6   GP.CDF
7   GH.III
8   GHJ...
9   LLJMM.
```



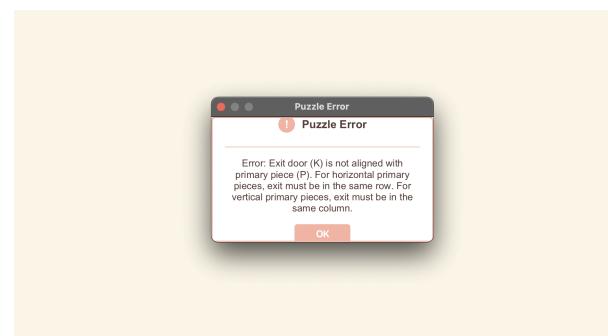
Pintu keluar di bawah

```
test > ≡ test4.txt
1   6 6
2   11
3 AAB..F
4 .PBCDF
5 GP.CDF
6 GH.III
7 GHJ...
8 LLJMM.
9   K
```



Pintu keluar tidak searah dengan bentuk primary piece

```
test > ≡ test5.txt
1   6 6
2   11
3 AAB..F
4 .PBCDF
5 GP.CDFK
6 GH.III
7 GHJ...
8 LLJMM.
```



Pintu keluar ada 2

```
test > ≡ test6.txt
1   6 6
2   11
3   K
4 AAB..F
5 .PBCDF
6 GP.CDF
7 GH.III
8 GHJ...
9 LLJMM.
10  K
```



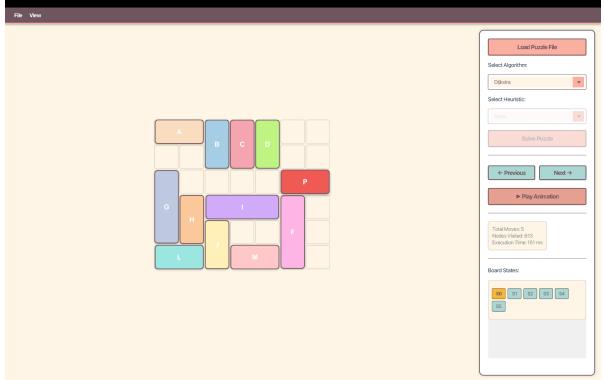
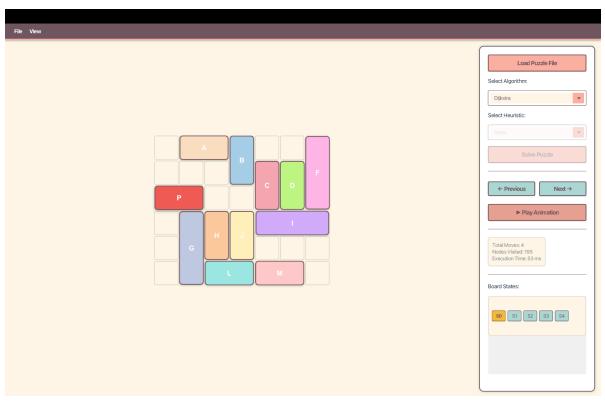
Tidak ada pintu keluar

```
test > ≡ test7.txt
```

```
1   6 6
2   11
3 AAB..F
4 ..BCDF
5 GP.CDF
6 GH.III
7 GHJ...
8 LLJMM.
```

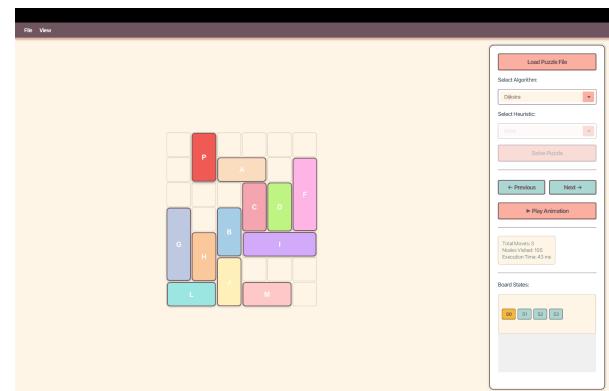


3.4. Algoritma Dijkstra

Input	Output
Pintu keluar di sebelah kanan	
Pintu keluar di sebelah kiri	

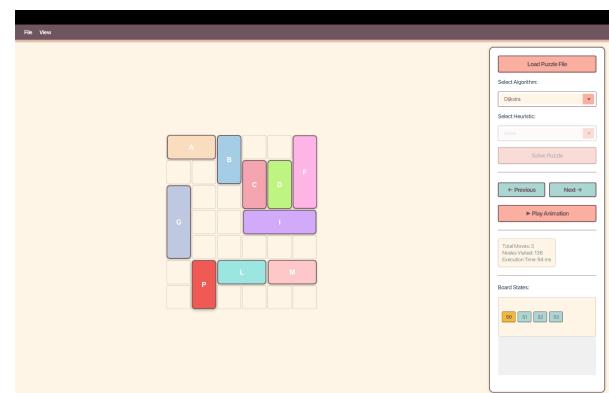
Pintu keluar di atas

```
test > ≡ test3.txt
1   6 6
2   11
3   K
4   AAB..F
5   .PBCDF
6   GP.CDF
7   GH.III
8   GHJ...
9   LLJMM.
```



Pintu keluar di bawah

```
test > ≡ test4.txt
1   6 6
2   11
3   AAB..F
4   .PBCDF
5   GP.CDF
6   GH.III
7   GHJ...
8   LLJMM.
9   K
```



Pintu keluar tidak searah dengan bentuk primary piece

```
test > ≡ test5.txt
1   6 6
2   11
3   AAB..F
4   .PBCDF
5   GP.CDFK
6   GH.III
7   GHJ...
8   LLJMM.
```



Pintu keluar ada 2

```
test > ≡ test6.txt
1   6 6
2   11
3   K
4   AAB..F
5   .PBCDF
6   GP.CDF
7   GH.III
8   GHJ...
9   LLJMM.
10  K
```



Tidak ada pintu keluar

```
test > ≡ test7.txt
1   6 6
2   11
3   AAB..F
4   .PBCDF
5   GP.CDF
6   GH.III
7   GHJ...
8   LLJMM.
```



BAB 5

Hasil Analisis

Dalam pengujian performa terhadap empat algoritma *pathfinding*, yaitu A*, Greedy Best First Search (GBFS), Uniform Cost Search (UCS), dan Dijkstra, kami melakukan analisis berdasarkan tiga metrik utama, yaitu jumlah node yang dikunjungi, waktu eksekusi, dan jumlah langkah solusi (total moves). Pengujian ini dilakukan terhadap beberapa konfigurasi papan permainan *Rush Hour Puzzle* untuk melihat sejauh mana efisiensi dan efektivitas masing-masing algoritma dalam menyelesaikan permasalahan.

Ditinjau dari jumlah node yang dikunjungi, algoritma GBFS menunjukkan kinerja yang paling efisien. GBFS hanya memprioritaskan node yang secara heuristic tampak paling dekat ke goal state, sehingga tidak menghabiskan banyak waktu untuk mengeksplorasi jalur-jalur alternatif. Sementara itu, A* mempertimbangkan cost aktual dari jalur meskipun menggunakan heuristic, sehingga eksplorasinya lebih luas dibanding GBFS, tetapi lebih terbatas dibanding algoritma pencarian UCS dan Dijkstra. UCS dan Dijkstra cenderung mengeksplorasi hampir seluruh ruang kemungkinan karena tidak menggunakan heuristic, sehingga jumlah node yang dikunjungi menjadi sangat besar.

Berdasarkan waktu eksekusi, hasil yang diperoleh berbanding lurus dengan jumlah node yang dieksplorasi. GBFS menghasilkan waktu eksekusi tercepat secara konsisten karena proses eksplorasinya yang selektif. A* berada di posisi tengah, menunjukkan waktu eksekusi yang lebih cepat dibanding UCS dan Dijkstra, tetapi sedikit lebih lambat dari GBFS. Sementara itu, UCS dan Dijkstra membutuhkan waktu paling lama untuk menyelesaikan puzzle..

Dalam hal jumlah langkah solusi (total moves), algoritma A*, UCS, dan Dijkstra secara umum mampu memberikan hasil yang optimal. Ketiga algoritma ini mempertimbangkan cost aktual dari setiap pergerakan, sehingga mampu menghasilkan solusi dengan jumlah langkah minimum. Berbeda halnya dengan GBFS, yang sering kali memberikan solusi lebih cepat namun dengan jumlah langkah yang tidak optimal. Hal ini terjadi karena GBFS hanya mengandalkan estimasi jarak ke goal (heuristic) tanpa memperhitungkan cost perjalanan aktual.

Berdasarkan hasil analisis tersebut, dapat disimpulkan bahwa masing-masing algoritma memiliki keunggulan tersendiri tergantung pada konteks penggunaannya. Apabila tujuan utama adalah memperoleh solusi secepat mungkin dan tidak terlalu mementingkan optimalitas, maka GBFS menjadi pilihan yang tepat. Jika yang diutamakan adalah keseimbangan antara kecepatan dan kualitas solusi, maka A* dapat dipilih karena memanfaatkan kombinasi antara heuristic dan cost aktual. Terakhir, UCS dan Dijkstra lebih cocok digunakan ketika diperlukan jaminan solusi optimal dalam konteks di mana heuristic tidak dapat digunakan atau tidak tersedia. Kombinasi pengujian ini memberikan gambaran menyeluruh terhadap kekuatan dan keterbatasan masing-masing algoritma dalam menyelesaikan puzzle *Rush Hour* secara efisien dan efektif.

BAB 6

Implementasi Bonus

5.1. Bonus 1: Implementasi Algoritma Alternatif

Dalam tugas ini, selain mengimplementasikan algoritma yang diwajibkan, yaitu Greedy Best First Search (GBFS), Uniform Cost Search (UCS), dan A*, kami juga menambahkan algoritma alternatif yaitu Dijkstra. Algoritma Dijkstra dikenal sebagai salah satu metode klasik dalam pencarian jalur terpendek (*shortest path*) pada struktur graf, dan sangat sesuai untuk kasus seperti puzzle *Rush Hour* yang dapat dimodelkan sebagai graf pencarian *state*.

Proses pencarian dimulai dengan menginisialisasi *state* awal, yaitu konfigurasi papan saat permainan dimulai. *State* ini diberikan *cost* (*g*) sebesar nol dan dimasukkan ke dalam *priority queue* yang memprioritaskan eksplorasi berdasarkan *cost* terkecil. Dijkstra akan terus mengekstrak *state* dengan *cost* terendah dari antrian dan mengevaluasi apakah *state* tersebut adalah *goal* (*primary piece* berhasil mencapai pintu keluar). Jika ya, maka solusi dikembalikan dan algoritma dihentikan.

Untuk menghindari eksplorasi *state* yang sama berulang kali (siklus), Dijkstra menggunakan *closed set*, yakni sebuah struktur penyimpanan untuk mencatat *state* yang sudah pernah dieksplorasi. Hal ini mencegah pemborosan waktu dan memori karena tidak perlu memproses ulang konfigurasi papan yang sama. Selain itu, untuk setiap *state* yang sedang dievaluasi, algoritma akan menghasilkan semua kemungkinan *successor state*, yaitu konfigurasi baru yang bisa dicapai dengan satu pergeseran *piece*.

Cost dari setiap *successor* akan dihitung sebagai *cost parent* ditambah 1, karena setiap langkah dipandang sebagai satu unit biaya. Jika terdapat jalur yang lebih pendek menuju suatu *state* yang sudah pernah ditemukan sebelumnya, maka *cost*-nya akan diperbarui dengan nilai yang lebih kecil. Untuk keperluan ini, Dijkstra menggunakan struktur data tambahan seperti *map* yang menyimpan *cost* minimum untuk setiap *state* yang ditemukan sejauh ini.

Berbeda dengan Breadth-First Search (BFS) yang mengeksplorasi semua *node* di tingkat kedalaman yang sama secara berurutan, Dijkstra selalu memprioritaskan eksplorasi terhadap *state* dengan total *cost* terkecil. Semua langkah dalam *Rush Hour* memiliki *cost* yang sama, perilaku Dijkstra secara praktis akan mirip dengan BFS, tetapi dengan kontrol yang lebih jelas terhadap pemilihan jalur optimal.

Algoritma ini menjamin solusi yang optimal karena selalu memilih jalur dengan *cost* terkecil terlebih dahulu. Selain itu, kami juga melacak metrik performa seperti jumlah *node* yang dikunjungi dan waktu eksekusi untuk memungkinkan analisis perbandingan efisiensi antara Dijkstra dan algoritma pencarian lainnya yang digunakan dalam tugas ini.

5.2. Bonus 2: Implementasi Heuristic Alternatif

Dalam rangka memenuhi bonus implementasi *heuristic* alternatif, kami mengembangkan tiga fungsi *heuristic* berbeda yang digunakan oleh algoritma pencarian berbasis estimasi, yaitu Greedy Best First Search (GBFS) dan A*. Tujuan utama dari pendekatan ini adalah untuk

menguji pengaruh berbagai bentuk estimasi terhadap performa algoritma dalam menyelesaikan puzzle Rush Hour. Ketiga heuristic tersebut mencakup estimasi berbasis jarak (Manhattan Distance), hambatan fisik (Blocking Vehicles), serta kombinasi dari keduanya.

5.2.1. Manhattan Distance

Heuristic pertama adalah *Manhattan Distance*, yang menghitung jarak garis lurus antara primary piece dan posisi pintu keluar, baik secara horizontal maupun vertikal tergantung orientasi piece. Untuk primary piece horizontal, jarak diukur dari ujung piece ke kolom tempat pintu keluar berada, sedangkan untuk orientasi vertikal jarak dihitung berdasarkan baris. Estimasi ini tidak mempertimbangkan hambatan lain, namun cukup efektif sebagai pendekatan ringan dan cepat dalam menggambarkan kedekatan terhadap goal state. Heuristic ini sangat cocok untuk konfigurasi puzzle yang relatif sederhana.

5.2.2. Blocking Vehicles

Heuristic kedua, *Blocking Vehicles*, lebih fokus pada kompleksitas sebenarnya dari puzzle. Ia menghitung berapa banyak kendaraan yang menghalangi jalur langsung primary piece menuju pintu keluar. Untuk piece horizontal, yang dihitung adalah jumlah kendaraan di antara ujung piece dan posisi K pada baris yang sama; sedangkan untuk piece vertikal, kendaraan yang menghalangi di kolom yang sama akan diperhitungkan. Heuristic ini memberikan representasi realistik terhadap jumlah intervensi yang dibutuhkan, karena memindahkan kendaraan penghalang biasanya melibatkan lebih dari satu langkah.

5.2.3. Combined Heuristic

Sebagai penyempurnaan, kami juga mengimplementasikan *Combined Heuristic* yang menggabungkan kedua pendekatan sebelumnya. Formula yang digunakan adalah $\text{manhattanDistance} + 2 * \text{blockingVehicles}$, di mana blocking vehicles diberikan bobot lebih tinggi karena dianggap mewakili effort lebih besar. Kombinasi ini memberikan estimasi yang lebih menyeluruh, mempertimbangkan baik jarak ke goal maupun hambatan aktual. Hasilnya adalah heuristic yang umumnya lebih akurat dalam memandu algoritma pencarian menuju solusi optimal dengan efisiensi waktu yang baik.

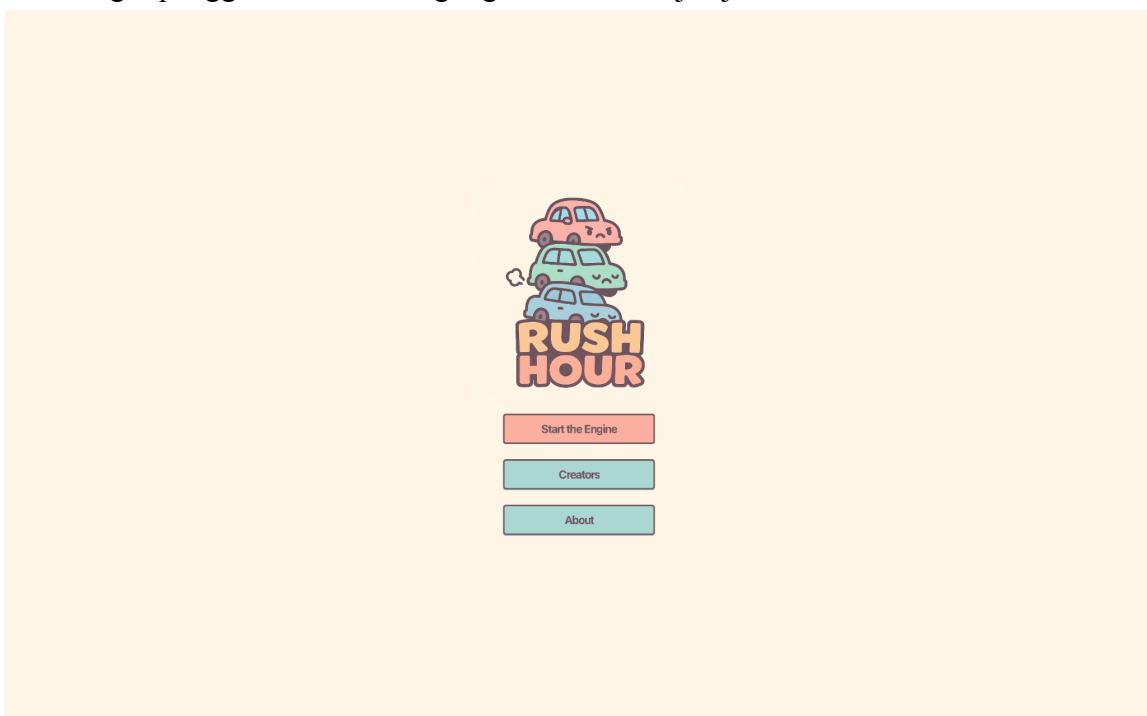
Dengan menyediakan tiga pilihan heuristic ini, pengguna dapat menyesuaikan strategi pencarian berdasarkan karakteristik puzzle yang dihadapi. Selain itu, pendekatan ini juga memungkinkan dilakukan analisis komparatif terhadap performa masing-masing heuristic, baik dari segi jumlah langkah, waktu eksekusi, maupun jumlah node yang dikunjungi. Keberagaman fungsi heuristic ini memperkaya fleksibilitas sistem pencarian dan meningkatkan kemampuan adaptasinya terhadap berbagai konfigurasi tantangan pada permainan Rush Hour.

5.3. Bonus 3: Graphical User Interface

Sebagai bagian dari fitur bonus, kami mengembangkan antarmuka grafis (*Graphical User Interface / GUI*) yang memungkinkan pengguna untuk berinteraksi secara visual dengan permainan Rush Hour. Antarmuka ini dirancang menggunakan JavaFX dan bertujuan untuk meningkatkan kenyamanan pengguna saat melakukan input file, menavigasi solusi, dan memahami jalannya algoritma. GUI tidak hanya memperindah tampilan, tetapi juga memperjelas alur pemecahan puzzle secara interaktif. Berikut adalah halaman-halaman utama yang tersedia dalam aplikasi GUI ini:

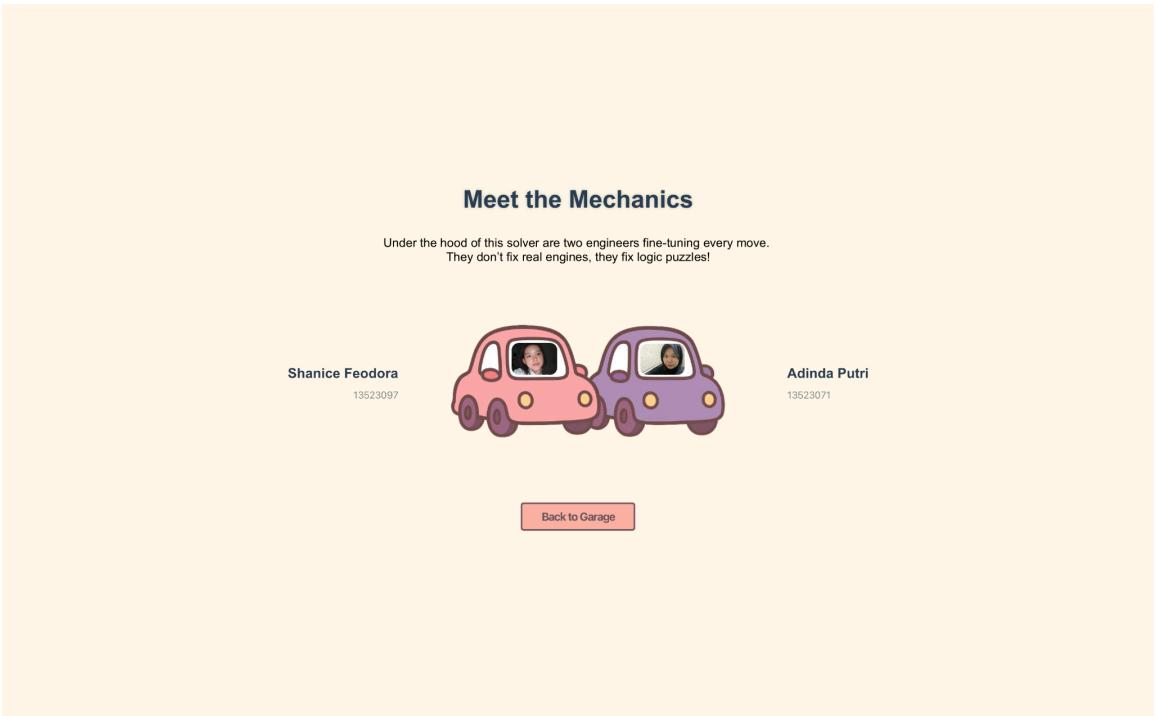
5.3.1. Landing Page

Halaman utama yang menyambut pengguna saat pertama kali membuka aplikasi. Di sini, pengguna dapat memilih untuk memulai permainan (*Start Game*), melihat informasi kreator, atau membaca penjelasan tentang aplikasi. Desain dibuat simpel dan intuitif agar pengguna tidak kebingungan dalam menjelajahi menu awal.



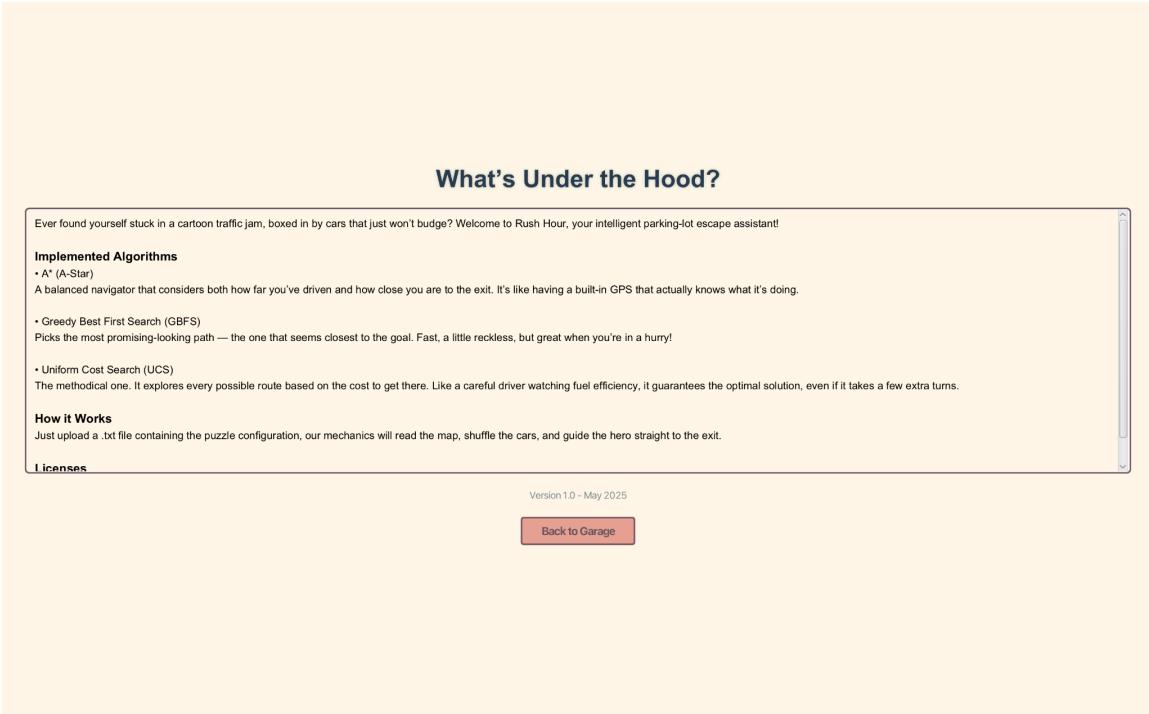
5.3.2. Creator

Halaman ini menampilkan daftar anggota tim pengembang aplikasi berupa nama dan NIM masing-masing anggota. Meskipun informasinya singkat, setiap identitas ditampilkan dengan hiasan ilustrasi kartun mobil yang lucu dan unik sehingga memberikan nuansa yang menyenangkan. Pendekatan visual ini bertujuan untuk memperkuat tema permainan mobil dalam aplikasi Rush Hour ini.



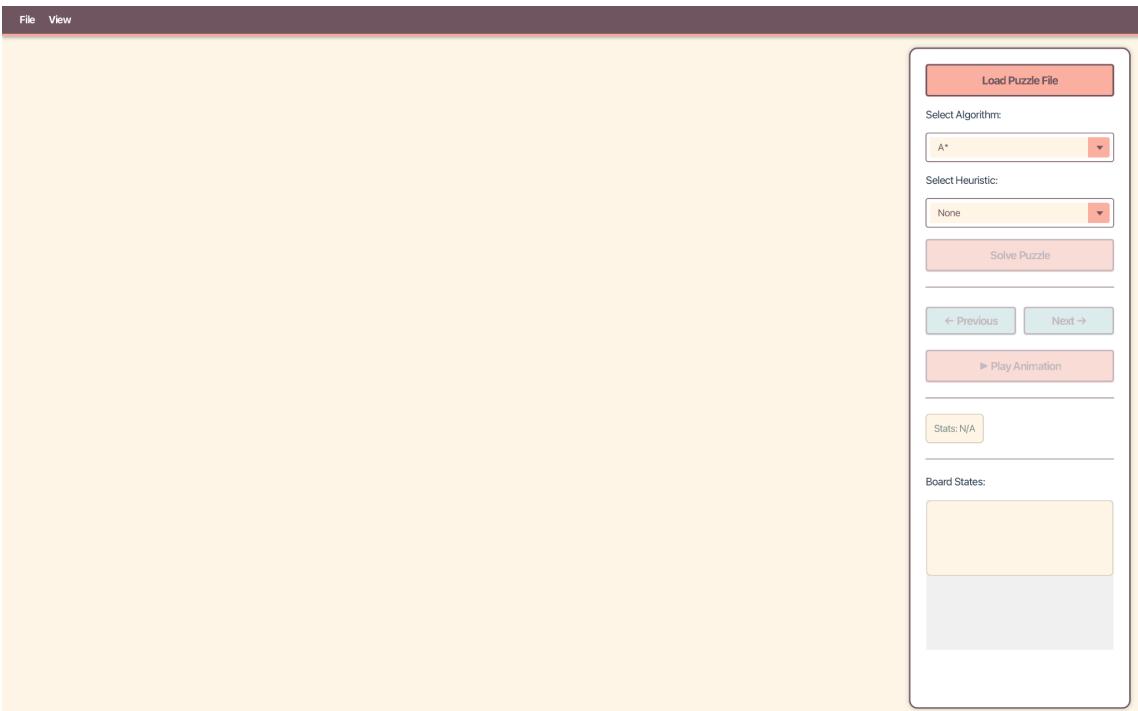
5.3.3. About

Berisi deskripsi singkat mengenai aplikasi, tujuannya, dan cara kerjanya. Dijelaskan bahwa aplikasi ini bertujuan untuk membantu pengguna menyelesaikan puzzle Rush Hour menggunakan algoritma pencarian cerdas (A*, UCS, GBFS, Dijkstra). Pengguna cukup memasukkan konfigurasi puzzle dalam bentuk file .txt, dan aplikasi akan secara otomatis menampilkan solusi langkah demi langkah.



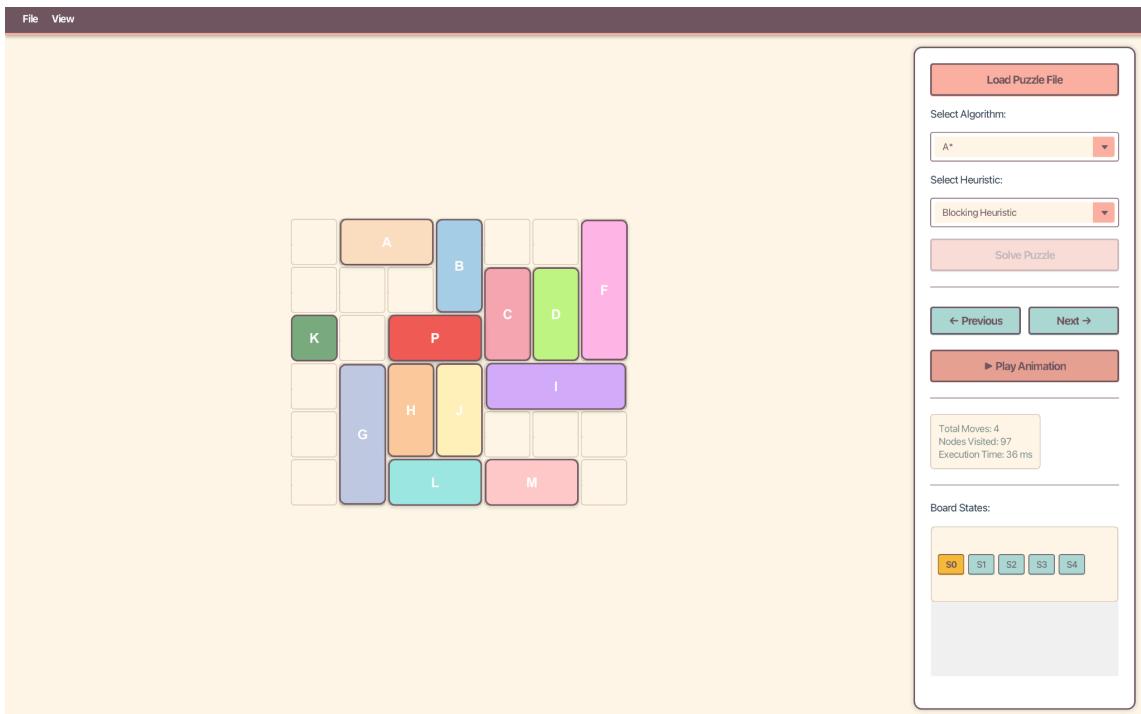
5.3.4. Solver

Halaman utama tempat pengguna dapat mengunggah file konfigurasi puzzle dan memilih algoritma serta heuristic (jika relevan). Setelah pengguna menekan tombol solve, sistem akan menjalankan proses pencarian solusi dan menampilkannya secara visual.



5.3.5. Result

Halaman ini menampilkan hasil dari proses pencarian solusi. Pengguna dapat melihat pergerakan kendaraan secara animasi langkah demi langkah, serta informasi penting seperti total langkah solusi, jumlah node yang dikunjungi, dan waktu eksekusi algoritma. Halaman ini juga menyediakan kontrol navigasi seperti *next*, *previous*, dan *play animation*. Selain itu, pengguna juga dapat menyimpan hasil pencarian dalam format .txt ke dalam direktori tertentu.



GUI ini diimplementasikan untuk meningkatkan pengalaman pengguna serta mendemonstrasikan bagaimana algoritma pencarian bekerja dalam konteks yang nyata dan dapat divisualisasikan. Melalui tampilan yang interaktif dan informatif, pengguna tidak hanya memperoleh solusi dari puzzle, tetapi juga pemahaman yang lebih dalam mengenai proses pencarian tersebut.

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	✓	

Pranala Repository

https://github.com/adndax/Tucil3_13523071_13523097

Daftar Pustaka

[1] R. Munir, *Penentuan Rute (Route/Path Planning) - Bagian 1*, 19 Mei 2025. [Daring].

Tersedia:

[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)

[2] R. Munir, *Penentuan Rute (Route/Path Planning) - Bagian 2*, 19 Mei 2025. [Daring].

Tersedia:

[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)