

Penerapan Algoritma Pathfinding dalam Penyelesaian Puzzle Rush Hour

Maheswara Bayu Kaindra (13523015) & Peter Wongsoredjo (13523039)

13523015@std.stei.itb.ac.id, 13523039@std.stei.itb.ac.id

1 Latar Belakang

Pada dunia komputasi, kemampuan merancang dan mengimplementasikan algoritma pencarian jalur (*pathfinding*) menjadi sangat penting, contohnya pada sistem navigasi, kecerdasan buatan, dan pemodelan (simulasi). Pencarian jalur menguji logika, efisiensi kerja, serta ketelitian dalam mengevaluasi kemungkinan solusi dari keadaan awal menuju tujuan, dengan memperhatikan batasan yang berlaku. Dengan demikian, penguasaan terhadap berbagai algoritma *pathfinding*, seperti *Uninformed* dan *Informed Search*, merupakan kunci dalam penyelesaian masalah kompleks yang krusial.

Tugas Kecil Strategi Algoritma ini dirancang untuk melatih pemahaman dan keterampilan mahasiswa dalam mengimplementasikan algoritma *pathfinding* melalui permainan Puzzle Rush Hour. Melalui tugas ini, mahasiswa tidak hanya dituntut untuk memahami prinsip kerja algoritma seperti *Uniform Cost Search*, *Greedy Best First Search*, dan *A**, tetapi juga menerapkannya dalam permasalahan nyata. Melalui tugas ini, mahasiswa diharapkan dapat memahami perbedaan mendasar antara *Uninformed* dan *Informed Search* dengan penggunaan *heuristic*. Mahasiswa juga ditantang untuk mengembangkan kemampuan eksplorasi melalui spesifikasi bonus.

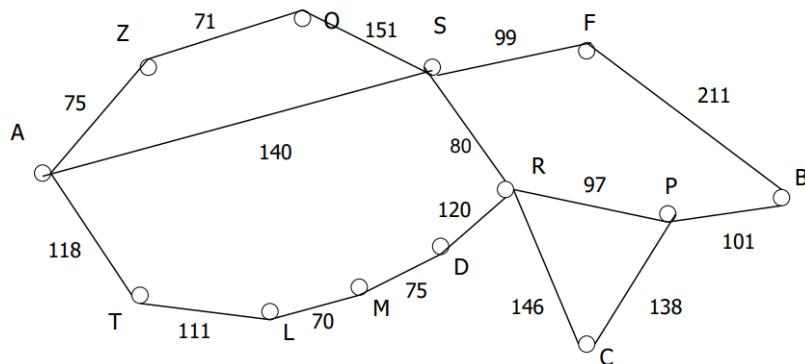
2 Landasan Teori

2.1 Algoritma Uniform Cost Search

Uniform Cost Search (UCS) merupakan salah satu algoritma pencarian *uninformed search*, yang berorientasi pada biaya, dengan tujuan utama untuk menemukan lintasan dari titik awal menuju solusi dengan total biaya minimum. Berbeda dengan algoritma *Breadth-First Search* yang melakukan penelusuran seluruh langkah tanpa mempedulikan bobot antar langkah, UCS mempertimbangkan bobot atau *cost* dari setiap aksi yang dilakukan untuk menentukan urutan eksplorasi setiap simpul. Dalam UCS, setiap simpul memiliki nilai biaya $g(n)$, dengan fungsi evaluasi $f(n)$ didefinisikan sebagai:

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

Dengan $g(n)$ merepresentasikan total biaya akumulasi gerakan mulai dari simpul awal hingga simpul n.



Gambar 2.1.1 : Contoh kasus graf
Sumber : PPT (tercantum)

Pada persoalan di atas, setiap sisi antar simpul memiliki bobot atau biaya yang merepresentasikan jarak atau beban untuk berpindah dari satu simpul ke simpul lainnya. UCS akan memprioritaskan penelusuran simpul berdasarkan nilai $g(n)$ terkecil, yaitu total biaya dari simpul akar A ke simpul saat ini. UCS mengeksplorasi simpul-simpul tetangga dari A, dan memilih simpul dengan $g(n)$ paling kecil untuk terus memperluas simpul-simpul baru dan memperbarui antrian jika ditemukan lintasan ke simpul yang sama dengan biaya yang lebih rendah.

UCS selalu menghasilkan solusi optimal yang meminimalkan total biaya perjalanan dari simpul awal ke simpul tujuan. Namun, UCS membutuhkan banyak memori karena bersifat eksploratif terhadap semua jalur “murah”. Hal tersebut berpotensi menghasilkan antrian yang cukup besar dengan ruang pencarian yang luas. Berikut merupakan gambaran Algoritma UCS.

```
function uniformCostSearch (input Node m) → result : List of Node
KAMUS
queue : PriorityQueue of Node {membandingkan cost atau g(n)}
currentNode : Node
result : List of Node
visited : Set of Node

ALGORITMA
queue ← new PriorityQueue()
queue.Enqueue(m, m.cost)
visited ← ∅

{ Mencari tetangga dengan cost terkecil }
while (!queue.isEmpty()) do
    currentNode ← queue.Dequeue() {ambil node terdepan (minimal cost)}

    if (currentNode.element = goal) then { Solusi ditemukan }
        result ← new List
        while (currentNode ≠ null) do { Membentuk Rantai Solusi }
            result.Append(currentNode.element)
        
```

```

        currentNode ← currentNode.parent
endwhile
→ result
endif

if (currentNode.element ∈ visited) then
    visited ← visited ∪ {currentNode.element}
    neighborList ← generatePath(currentNode) {cari tetangga}
    while (!neighborList.isEmpty()) do
        if (neighbor.board ∈ visited) then
            queue.Enqueue(neighbor, neighbor.cost)
        endif
    endwhile
endif
endwhile

→ ∅ { solusi tidak ditemukan }

```

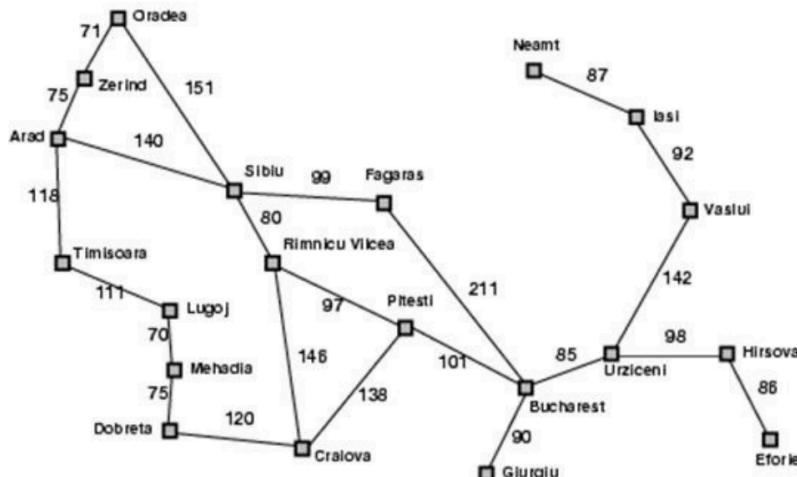
Pada algoritma tersebut, node menggunakan konsep **reversed tree**, di mana ia memiliki element, cost, dan parent.

2.2 Algoritma Greedy Best First Search

Greedy Best First Search (GBFS) merupakan salah satu algoritma pencarian dengan *heuristik* yang termasuk *informed search*. GBFS menggunakan *heuristik* berupa pemilihan simpul yang paling dekat dengan tujuan (memiliki *cost* terkecil) berdasarkan fungsi evaluasi $f(n)$. Di sini, fungsi evaluasi didefinisikan sebagai:

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

Di mana $h(n)$ adalah **estimasi biaya** dari simpul n ke simpul tujuan.



Gambar 2.2.1 : Contoh kasus graf (Jarak Antar Kota)

Sumber : PPT (tercantum)

Pada kasus di atas, $h(n)$ dapat berupa **jarak garis lurus dari kota saat ini ke kota tujuan** (*straight-line distance*). Dengan menggunakan informasi tersebut, GBFS akan mengevaluasi semua simpul terbuka dan memilih simpul dengan $h(n)$ terkecil (Maulidevi, 2025). GBFS termasuk cepat karena dengan jelas memilih simpul yang terlihat “menjanjikan”, namun memiliki beberapa kelemahan:

1. **Tidak menjamin solusi optimal**, seringkali solusi yang dihasilkan berada di maksimum atau minimum lokal, layaknya algoritma Greedy.
2. Memiliki penglihatan yang sempit, sehingga **tidak dapat membatalkan keputusan sebelumnya** (*irrevocable*).

Berikut merupakan Algoritma pencarian menggunakan GBFS (sesuai konteks Tugas Kecil ini dan interpretasi developer).

```
function GreedyBestFirstSearch(input inputNode : Node) → List of Node
KAMUS
queue : PriorityQueue of Node {membandingkan cost atau g(n)}
currentNode : Node
result : List of Node
visited : Set of Node
currentNode : Node

ALGORITMA
queue.enqueue(inputNode)

while (not queue.isEmpty()) do
    currentNode ← queue.dequeue()
    if (visited.contains(currentNode)) then
        continue
    endif
    visited.add(currentNode)

    if (node.isSolved()) then
        while (currentNode ≠ NIL) do
            result.add(0, currentNode)
            currentNode ← currentNode.getParent()
        endwhile
        → result
    endif

    for (semua neighbor ← tetangga node) do
        if (not visited.contains(neighbor)) then
            queue.enqueue(neighbor)
        endif
    endfor
```

```
endwhile
```

```
→ NIL
```

Pada algoritma tersebut, node menggunakan konsep **reversed tree**, di mana ia memiliki board, cost, dan parent.

2.3 Algoritma A* (A Star)

Algoritma A* (*dibaca: A-Star*) merupakan algoritma *pathfinding* yang menggabungkan algoritma *Uniform Cost Search* (UCS) dan *Greedy Best First Search* (GBFS) (yang keduanya sudah dibahas di atas). A* menggunakan fungsi evaluasi $f(n)$ yang merupakan total *cost* ke simpul tujuan ditambah dengan sejauh mana perjalanan sudah ditempuh (dengan *cost* per langkah konstan seperti UCS).

$$f(n) = g(n) + h(n), \text{ di mana}$$

$g(n)$ merupakan *cost* uniform atau jauhnya perjalanan yang sudah ditempuh dan $h(n)$ merupakan *cost* dari simpul saat ini ke tujuan. Dengan menggabungkan *uniform cost* dengan *cost*, A* mampu menghasilkan solusi yang optimal dan *complete*. Berikut merupakan *pseudocode* untuk algoritma A*.

```
function AStar(input inputNode : Node) → List of Node
    KAMUS
        queue : PriorityQueue of Node {membandingkan cost atau g(n)}
        currentNode : Node
        result : List of Node
        visited : Set of Node
        currentNode : Node

    ALGORITMA
        queue.enqueue(inputNode)

        while (not queue.isEmpty()) do
            currentNode ← queue.dequeue()
            if (visited.contains(currentNode)) then
                continue
            endif
            visited.add(currentNode)

            if (node.isSolved()) then
                while (currentNode ≠ NIL) do
                    result.add(0, currentNode)
                    currentNode ← currentNode.getParent()
                endwhile
                → result
            endif
```

```

for (semua neighbor ← tetangga node) do
    if (not visited.contains(neighbor)) then
        queue.enqueue(neighbor)
    endif
endfor
endwhile
→ NIL

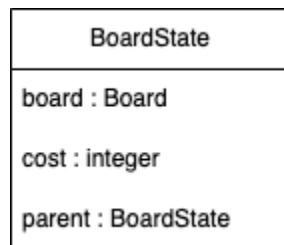
```

Dengan ini, algoritma A* mampu meminimalkan jumlah simpul yang dikunjungi dibandingkan UCS dan GBFS karena sifatnya lebih *informed*. Pada algoritma tersebut, node menggunakan konsep **reversed tree**, di mana ia memiliki board, cost, dan parent.

3 Strategi Penerapan Algoritma

3.1 Struktur Data

Untuk merepresentasikan node, digunakan struktur data (class) BoardState, yang secara umum bentuknya sebagai berikut.



*Gambar 3.1.1 Node direpresentasikan dengan BoardState.
Dibuat dengan [draw.io](#)*

Pada struktur tersebut, BoardState tidak hanya menyimpan kondisi papan, melainkan juga *cost* menuju target dan parent (dari mana ia diiterasi sebelumnya), sehingga struktur ini membentuk *reversed tree*. Node inilah yang akan diiterasi dalam algoritma UCS, GBFS, dan A*.

3.2 Algoritma Uniform Cost Search

Dalam konteks permainan Rush Hour, UCS digunakan karena setiap gerakan kendaraan memiliki bobot ataupun biaya yang konstan, sebesar satu satuan langkah. Dengan demikian, $g(n)$ akan pada algoritma UCS untuk permainan ini dapat didefinisikan sebagai jumlah langkah kendaraan yang telah dilakukan, mulai dari konfigurasi awal papan, hingga mencapai konfigurasi papan tertentu. Karena setiap langkah kendaraan pada permainan memiliki bobot yang sama, UCS dalam kasus ini akan berperilaku serupa dengan BFS, dan akan menghasilkan solusi optimum dengan jumlah langkah minimum untuk mengeluarkan kendaraan utama dari papan.

A	A	A	
P	P		B
			B



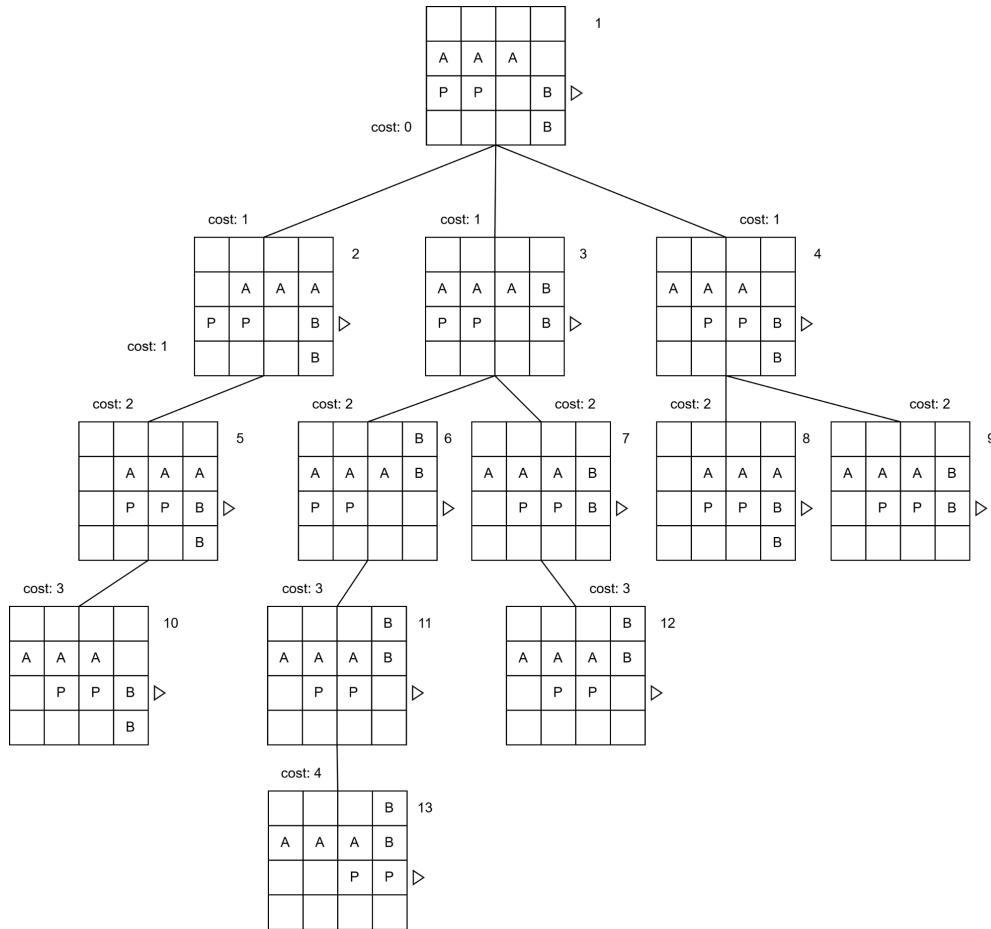
Gambar 3.2.1 : Contoh kasus Rush Hour

Sumber : Penulis

Sebagai ilustrasi, pada kasus konfigurasi papan seperti berikut, algoritma UCS akan menambahkan kondisi papan awal ke dalam antrian. Dilakukan *deque* untuk elemen pertama yang berupa kondisi awal papan, dan karena kondisi papan belum merupakan kondisi *goal*, dan belum pernah dikunjungi sebelumnya, algoritma akan menelusuri semua node (semua gerakan yang bisa dilakukan *piece* di papan), dan memasukkannya ke dalam antrian dengan *cost* + 1.

Dengan demikian, terdapat 3 *node* di dalam *queue*, dan algoritma mengulang kembali proses untuk kondisi kedua, dan karena kondisi papan juga belum merupakan kondisi *goal*, maka algoritma kembali menelusuri semua kondisi lain yang mungkin tercipta dari gerakan *piece*, yang sebelumnya belum pernah dikunjungi, dan menambahkannya ke dalam *queue* dengan *cost* + 1. Seluruh proses terulang, hingga dicapai suatu kondisi *goal*, atau tidak ada lagi kombinasi yang bisa dikunjungi, yang berarti tidak ditemukan solusi.

Algoritma Uniform Cost Search untuk kondisi awal papan seperti berikut, akan memunculkan suatu tree penelusuran berupa:



Gambar 3.2.2 : Pohon penelusuran Uniform Cost Search
Sumber : Penulis

3.3 Algoritma Greedy Best First Search

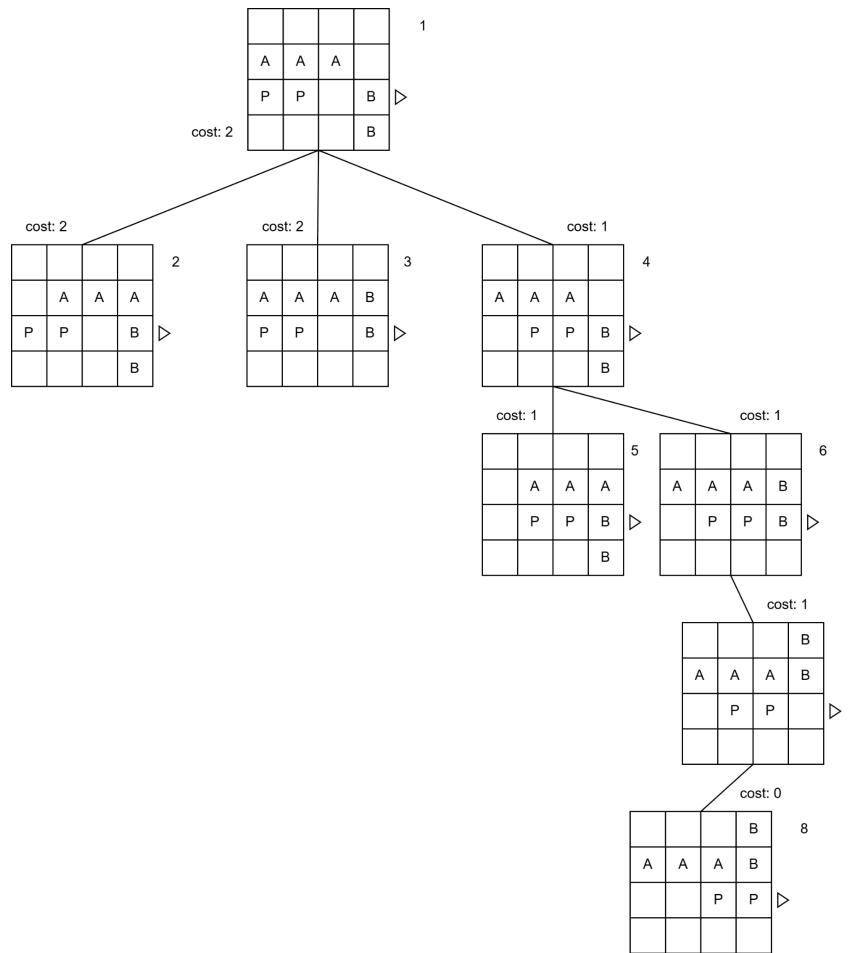
Dalam konteks permainan Rush Hour, setiap konfigurasi papan (BoardState) dianggap sebagai satu simpul dalam ruang pencarian. GBFS bekerja dengan memilih konfigurasi papan yang terlihat paling menjanjikan untuk mendekati kondisi goal. Konfigurasi papan yang terlihat menjanjikan ditentukan dengan memilih fungsi heuristic $h((n))$ yang diinginkan. Dalam implementasinya, terdapat tiga heuristic yang digunakan pada tugas ini yaitu: jarak kendaraan utama (Primary Piece) ke pintu keluar, banyak kendaraan (Piece) lain yang menghalangi kendaraan utama menuju ke pintu keluar, ataupun gabungan dari kedua heuristic tersebut. Proses pencarian dilakukan dengan menggunakan struktur data PriorityQueue, di mana setiap node dimasukkan dengan prioritas berdasarkan nilai $h((n))$ terkecil.

Sebagai ilustrasi, digunakan pada kasus konfigurasi papan pada gambar 3.2.1, algoritma GBFS dengan menggunakan heuristic jarak kendaraan utama menuju ke pintu keluar akan mulai dengan meletakkan BoardState awal ke dalam Queue dengan cost awal Board. Dilakukan deque untuk elemen pertama, dan karena kondisi papan belum merupakan kondisi goal, dan belum pernah dikunjungi

sebelumnya, algoritma akan menelusuri semua node (semua gerakan yang bisa dilakukan piece di papan), dan memasukkannya ke dalam antrian dengan *cost* (jarak kendaraan menuju exit).

Algoritma kemudian akan memilih elemen pertama dalam priority queue yang berupa kondisi papan dengan cost yang paling rendah. Algoritma akan diulang kembali untuk kondisi papan saat itu, dan apabila papan kembali belum berupa kondisi goal, proses akan terulang seterusnya. Apabila dalam suatu ranting Greedy Best First Search, tidak ditemukan kondisi goal hingga akhir, algoritma akan meneruskan proses untuk simpul hidup lainnya hingga pada akhirnya ditemukan suatu solusi.

Algoritma Greedy Best First Search untuk kondisi awal papan seperti berikut, akan memunculkan suatu tree penelusuran berupa:



Gambar 3.3.1 : Pohon penelusuran Greedy Best First Search

Sumber : Penulis

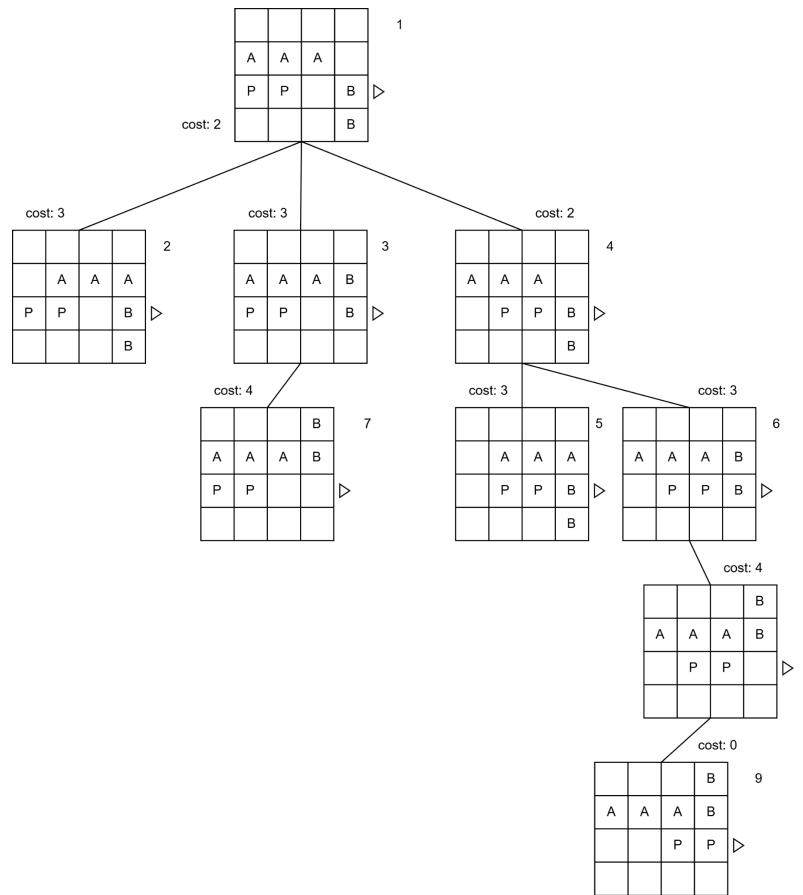
3.4 Algoritma A* (A Star)

Dalam konteks penyelesaian permainan Rush Hour, algoritma A* digunakan untuk mencari solusi apabila diketahui informasi mengenai jumlah bagian mobil lain yang menghalangi (*cost*), jarak piece saat ini ke target (*distanceCost*), dan jumlah langkah yang h dilakukan (*pathCost*). Sesuai dasar teori, $f(n)$

pada algoritma A* merupakan penjumlahan dari *pathCost* dengan salah satu antara *cost*, *distanceCost*, atau keduanya (sesuai heuristik yang dipilih).

Algoritma akan memilih BoardState input sebagai currentBoard pertama kali. Untuk setiap iterasi, currentBoard diambil dari *priority queue* (berdasarkan $f(n)$ terkecil) dan akan ditambahkan ke *boardChain* (lihat implementasi setiap algoritma) apabila memenuhi syarat (cost nya kurang dari cost saat ini). List of nodes yang sudah ada di *boardChain* (dengan urutan yang berbalik) akan dimasukkan ke result dari belakang ke depan (dengan melihat parent dari masing-masing node) hingga ke kondisi awal, sehingga urutan node akan sesuai.

Sebagai ilustrasi, lihat gambar 3.4.1. Algoritma ini menggunakan heuristik Jumlah Mobil yang menghalangi, sehingga $f(n) = \text{cost} + \text{pathCost}$, yaitu jumlah mobil yang menghalangi *primary piece* ditambah jumlah langkah yang sudah dilakukan. Dalam setiap iterasi, Prioqueue memilih satu node dengan $f(n)$ terkecil, membuangnya dari list, dan memeriksa apakah gerakan tersebut valid (baik secara aturan permainan, maupun secara historik, bahwa node yang sudah dikunjungi tidak boleh dikunjungi lagi). Seperti pada gambar, algoritma selalu melanjutkan pencarian dari node terkecil yang masih *open*.



Gambar 3.4.1 : Pohon penelusuran A*

Sumber : Penulis

4 Implementasi, Pengujian, dan Analisis

4.1 Implementasi

4.1.1 Components

Class Board

Atribut		
Nama	Tipe	Deskripsi
rows, cols	private int	Menyimpan panjang dan lebar Board.
cell	private Cell[][]	Menyimpan matriks yang berisi cell.
pieces	private List<Piece>	Menyimpan list <i>piece</i> yang akan digunakan.
exit	private Exit	Menyimpan informasi pintu keluar <i>puzzle</i> .
Metode		
Nama		Deskripsi
public Board(int rows, int cols)		Konstruktor dengan input parameter panjang dan lebar.
public Board(Board other)		Copy constructor, konstruksi board baru dengan referensi board lama
public void setCell(int row, int col, Cell cell)		Mengubah suatu Cell dalam matriks di posisi row dan col
public Cell getCell(int row, int col)		Mengembalikan suatu Cell dalam matriks di posisi row dan col
public int getRows()		Mengembalikan rows.
public int getCols()		Mengembalikan cols.
public void addPiece(Piece piece)		Menambah sebuah <i>piece</i> pada

	pieces.
<code>public void addPrimaryPiece(PrimaryPiece piece)</code>	Menambah <i>primary piece</i> pada primaryPiece.
<code>public List<Piece> getPieces()</code>	Mengembalikan pieces.
<code>public PrimaryPiece getPrimaryPiece()</code>	Mengembalikan primaryPiece.
<code>public void setExit(Exit exit)</code>	Menentukan posisi exit dalam row dan col
<code>public Exit getExit()</code>	Mengembalikan exit.
<code>public boolean isInsideBoard(int row, int col)</code>	Mengembalikan <i>true</i> jika koordinat berada di dalam <i>board</i> .
<code>public boolean isCellEmpty(int row, int col)</code>	Mengembalikan <i>true</i> jika <i>cell</i> kosong.
<code>public boolean isSolved()</code>	Mengembalikan <i>true</i> jika kondisi board sudah mencapai target.
<code>public boolean equalTo(Board other)</code>	Pengganti <i>equals</i> untuk <i>board</i> .
<code>public int calculateCost()</code>	Menghitung <i>cost</i> (mobil yang menghalangi).
<code>public int calculateDistanceCost()</code>	Menghitung jarak ke target.
<code>public int calculateCombinedCost()</code>	Menggabungkan <i>cost</i> dengan <i>distance cost</i> .
Implementasi Metode	
<pre>public class Board { private int rows, cols; private Cell[][] cell; private List<Piece> pieces; private PrimaryPiece primaryPiece; private Exit exit; public Board(int rows, int cols){ this.rows = rows; this.cols = cols; } }</pre>	

```

        this.cell = new Cell[rows][cols];
        this.pieces = new ArrayList<>();
    }

    public Board(Board other){
        this.rows = other.rows;
        this.cols = other.cols;
        this.cell = new Cell[rows][cols];
        this.pieces = new ArrayList<>();

        for(int r = 0; r < rows; r++){
            for(int c = 0; c < cols; c++){
                this.cell[r][c] = new Cell(other.cell[r][c]);
            }
        }

        for(Piece piece : other.pieces){
            Piece newPiece;
            if(piece instanceof PrimaryPiece){
                newPiece = new PrimaryPiece(piece.getRow(), piece.getCol(),
piece.getSymbol(), piece.getLength(), piece.isHorizontal());
                this.addPrimaryPiece((PrimaryPiece) newPiece);
            } else {
                newPiece = new Piece(piece.getRow(), piece.getCol(),
piece.getSymbol(), piece.getLength(), piece.isHorizontal());
                this.addPiece(newPiece);
            }

            for(int[] position : newPiece.getFullPosition()){
                this.cell[position[0]][position[1]].occupy(newPiece.getSymbol());
            }
        }

        Exit newExit = other.getExit();
        this.setExit(new Exit(newExit.getRow(), newExit.getCol()));
    }

    public void setCell(int row, int col, Cell cell){
        this.cell[row][col] = cell;
    }

    public Cell getCell(int row, int col){
        return this.cell[row][col];
    }

    public int getRows(){

```

```
        return this.rows;
    }

    public int getCols(){
        return this.cols;
    }

    public void addPiece(Piece piece){
        this.pieces.add(piece);
    }

    public void addPrimaryPiece(PrimaryPiece piece){
        this.primaryPiece = piece;
        this.pieces.add(piece);
    }

    public List<Piece> getPieces(){
        return this.pieces;
    }

    public PrimaryPiece getPrimaryPiece(){
        return this.primaryPiece;
    }

    public void setExit(Exit exit){
        this.exit = exit;
    }

    public Exit getExit(){
        return this.exit;
    }

    public boolean isInsideBoard(int row, int col){
        return row >= 0 && row < rows && col >= 0 && col < cols;
    }

    public boolean isCellEmpty(int row, int col){
        return isInsideBoard(row, col) && cell[row][col] != null &&
cell[row][col].isEmpty();
    }

    public boolean isSolved(){
        return exit.isHit();
    }

    public void displayBoard() {
```

```

        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                Cell cell = getCell(r, c);
                System.out.print(cell != null ? cell.getSymbol() : '.');
            }
            System.out.println();
        }

        System.out.println("Exit: " + exit.getRow() + ", " + exit.getCol());
    }

    public boolean equalTo(Board other){
        if(this.rows != other.getRows() || this.cols != other.getCols()){
            return false;
        } // jujur bakal sama sih, jadi ngga pake ya yauda

        for(int r = 0; r < this.rows; r++){
            for(int c = 0; c < this.cols; c++){
                char a = this.getCell(r, c).getSymbol();
                char b = other.getCell(r, c).getSymbol();
                if(a != b){
                    return false;
                }
            }
        }

        return true;
    }

    public int calculateCost() {
        int returnValue = 0;
        Piece tempPrimaryPiece = this.getPrimaryPiece();
        Exit tempExit = this.getExit();
        boolean tempIsHorizontal = tempPrimaryPiece.isHorizontal();
        if (tempIsHorizontal) {
            if (tempExit.getCol() == -1) {
                // Cek ke kiri.
                for (int i = tempPrimaryPiece.getCol()-1; i >= 0 ; i--) {
                    if (!this.getCell(tempPrimaryPiece.getRow(), i).isEmpty()) {
                        returnValue++;
                    }
                }
            } else {
                // Cek ke kanan.
                for (int i = tempPrimaryPiece.getCol() + tempPrimaryPiece.getLength()

```

```

- 1; i < this.getCols() - 1; i++) {
            if (!this.getCell(tempPrimaryPiece.getRow(), i).isEmpty()) {
                returnValue++;
            }
        }
    } else {
        if (tempExit.getRow() == -1) {
            // Cek ke atas.
            for (int i = tempPrimaryPiece.getRow()-1; i >= 0; i--) {
                if (!this.getCell(i, tempPrimaryPiece.getCol()).isEmpty()) {
                    returnValue++;
                }
            }
        } else {
            // Cek ke bawah.
            for (int i = tempPrimaryPiece.getRow() + tempPrimaryPiece.getLength()
- 1; i < this.getRows() - 1; i++) {
                if (!this.getCell(i, tempPrimaryPiece.getCol()).isEmpty()) {
                    returnValue++;
                }
            }
        }
    }
    return returnValue;
}

public int calculateDistanceCost() {
    if (this.getPrimaryPiece().isHorizontal()) {
        return Math.abs(this.getPrimaryPiece().getCol() -
this.getExit().getCol());
    }
    return Math.abs(this.getPrimaryPiece().getRow() - this.getExit().getRow());
}

public int calculateCombinedCost() {
    return this.calculateCost() + this.calculateDistanceCost();
}
}

```

Class BoardState

Atribut		
Nama	Tipe	Deskripsi

<code>board</code>	<code>private Board</code>	Menyimpan papan milik <i>node</i> .
<code>cost</code>	<code>private int</code>	Menyimpan banyak halangan ke target.
<code>pathCost</code>	<code>private int</code>	Menyimpan panjang path yang sudah dilalui.
<code>distanceCost</code>	<code>private int</code>	Menyimpan jarak <i>primary piece</i> saat ini ke <i>target</i> .
<code>parent</code>	<code>private BoardState</code>	Menyimpan asal-muasal iterasi <i>node</i> .
Metode		
Nama	Deskripsi	
<code>public BoardState(Board board, int cost, int pathCost, BoardState parent)</code>	Konstruktor BoardState dengan tiga jenis parameter input berbeda.	
<code>public BoardState(Board board, int pathCost, BoardState parent)</code>		
<code>public BoardState(Board board, BoardState parent)</code>		
<code>public Board getBoard()</code>	Mengembalikan board.	
<code>public int getCost()</code>	Mengembalikan cost.	
<code>public int getPathCost()</code>	Mengembalikan pathCost.	
<code>public int getDistanceCost()</code>	Mengembalikan distanceCost.	
<code>public int getAStarCost()</code>	Menghitung <i>cost</i> untuk algoritma A* tergantung heuristik yang digunakan.	
<code>public int getAStarDistanceCost()</code>		
<code>public int calculateAStarCombinedCost()</code>		
<code>public BoardState getParent()</code>	Mengembalikan parent.	
<code>private Piece findPiece(Board board, char symbol)</code>	Mengembalikan suatu Piece dalam suatu board berdasarkan input simbol	
<code>public List<BoardState> generatePath()</code>	Menghasilkan list tetangga yang mungkin dikunjungi.	

Implementasi Metode

```
public class BoardState {
    private Board board;
    private int cost;
    private int pathCost;
    private int distanceCost;
    private BoardState parent;

    public BoardState(Board board, int cost, int pathCost, BoardState parent){
        this.board = board;
        this.cost = cost;
        this.pathCost = pathCost;
        this.distanceCost = board.calculateDistanceCost();
        this.parent = parent;
    }

    public BoardState(Board board, int pathCost, BoardState parent){
        this.board = board;
        this.cost = board.calculateCost();
        this.pathCost = pathCost;
        this.distanceCost = board.calculateDistanceCost();
        this.parent = parent;
    }

    public BoardState(Board board, BoardState parent){
        this.board = board;
        this.cost = board.calculateCost();
        this.pathCost = 0;
        this.distanceCost = board.calculateDistanceCost();
        this.parent = parent;
    }

    public Board getBoard() {
        return this.board;
    }

    public int getCost() {
        return this.cost;
    }

    public int getPathCost() {
        return this.pathCost;
    }

    public int getDistanceCost() {
```

```

        return this.distanceCost;
    }

    public int getAStarCost() {
        return this.pathCost + this.cost;
    }

    public int getAStarDistanceCost() {
        return this.pathCost + this.distanceCost;
    }

    public int calculateAStarCombinedCost() {
        return this.cost + this.distanceCost + this.pathCost;
    }

    public BoardState getParent() {
        return this.parent;
    }

    private Piece findPiece(Board board, char symbol) {
        for (Piece piece : board.getPieces()) {
            if (piece.getSymbol() == symbol) {
                return piece;
            }
        }
        return null;
    }

    public List<BoardState> generatePath(){
        List <BoardState> path = new ArrayList<>();
        Board currentBoard = this.board;

        for(Piece piece : currentBoard.getPieces()){
            for(boolean forward : new boolean[]{true, false}){
                if(piece.isValidMove(currentBoard, forward)){
                    Board newBoard = new Board(currentBoard);
                    Piece copyPiece = findPiece(newBoard, piece.getSymbol());

                    newBoard = copyPiece.move(newBoard, forward);

                    BoardState newBoardState = new BoardState(newBoard, pathCost + 1,
this);

                    if(!newBoardState.getBoard().equalTo(currentBoard)){
                        path.add(newBoardState);
                    }
                }
            }
        }
    }
}

```

```

        }
    }
    return path;
}
}

```

Class Cell

Atribut		
Nama	Tipe	Deskripsi
row, col	private int	Menyimpan koordinat letak sel.
symbol	private char	Menyimpan simbol sel sesuai pembacaan .txt.
Metode		
Nama	Deskripsi	
public Cell(int row, int col, char symbol)	Konstruktor cell, mengatur row, col, dan symbol.	
public Cell(Cell other)	Copy constructor.	
public int getRow()	Mengembalikan row.	
public int getCol()	Mengembalikan col	
public char getSymbol()	Mengembalikan simbol	
public void setSymbol(char symbol)	Menentukan simbol yang dimiliki suatu sel	
public boolean isEmpty()	Mengembalikan true jika sel kosong (.), dan false jika sel terisi suatu piece	
public void occupy(char pieceId)	Mengisi suatu sel dengan suatu char simbol	
public void clear()	Mengosongkan suatu sel dengan mengisinya dengan ‘.’	
Implementasi Metode		

```
public class Cell {
    private int row;
    private int col;
    private char symbol;

    public Cell(int row, int col, char symbol) {
        this.row = row;
        this.col = col;
        this.symbol = symbol;
    }

    public Cell(Cell other) {
        this.row = other.row;
        this.col = other.col;
        this.symbol = other.symbol;
    }

    public int getRow() {
        return row;
    }

    public int getCol() {
        return col;
    }

    public char getSymbol() {
        return symbol;
    }

    public void setSymbol(char symbol) {
        this.symbol = symbol;
    }

    public boolean isEmpty(){
        return this.symbol == '.';
    }

    public void occupy(char pieceId){
        this.symbol = pieceId;
    }

    public void clear(){
        this.symbol = '.';
    }
}
```

Class Exit

Atribut		
Nama	Tipe	Deskripsi
row, col	private int	Menyimpan koordinat pintu keluar.
isHit	private boolean	Menyimpan data apakah target sudah tercapai.
Metode		
Nama		Deskripsi
public Exit(int row, int col)		Konstruktor exit.
public int getRow()		Mengembalikan row.
public int getCol()		Mengembalikan col.
public boolean isHit()		Mengembalikan <i>true</i> jika target telah dicapai (pintu exit tidak null).
public void setHit(boolean hit)		Mengubah status hit (apabila sudah sampai target).
public int[] getPoint()		Mengembalikan koordinat.
Implementasi Metode		
<pre>public class Exit { private int row; private int col; private boolean isHit; public Exit(int row, int col) { this.row = row; this.col = col; this.isHit = false; } public int getRow() { return row; } }</pre>		

```

public int getCol() {
    return col;
}

public boolean isHit() {
    return isHit;
}

public void setHit(boolean hit) {
    isHit = hit;
}

public int[] getPoint(){
    return new int[]{row, col};
}
}

```

Class Piece

Atribut		
Nama	Tipe	Deskripsi
row, col	protected int	Menyimpan koordinat posisi awal piece.
symbol	protected char	Menyimpan simbol (huruf yang dipegang oleh piece).
length	protected int	Menyimpan panjang piece.
isHorizontal	protected boolean	Menyimpan orientasi arah piece (vertikal atau horizontal).
Metode		
Nama	Deskripsi	
public Piece(int row, int col, char symbol, int length, boolean isHorizontal)	Konstruktor Piece, menginisialisasi row, col, symbol, length	
public int getRow()	Mengembalikan row.	
public int getCol()	Mengembalikan col.	

<code>public char getSymbol()</code>	Mengembalikan symbol.
<code>public int getLength()</code>	Mengembalikan length.
<code>public boolean isHorizontal()</code>	Mengembalikan <i>true</i> jika orientasi <i>piece</i> horizontal.
<code>public void setStartPosition(int row, int col)</code>	Mengatur posisi start.
<code>public int[][] getFullPosition()</code>	Mengembalikan seluruh posisi piece.
<code>public boolean isValidMove(Board board, boolean forward)</code>	Mengembalikan <i>true</i> apabila gerakan yang dilakukan valid.
<code>public Board move(Board board, boolean forward)</code>	Mengerakan Piece dan mengembalikan kondisi papan.

Implementasi Metode

```

public class Piece {
    protected int row;
    protected int col;
    protected char symbol;
    protected int length;
    protected boolean isHorizontal;

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

    public int getRow() {
        return row;
    }

    public int getCol() {
        return col;
    }

    public char getSymbol() {
        return symbol;
    }
}

```

```
public int getLength() {
    return length;
}

public boolean isHorizontal() {
    return isHorizontal;
}

public void setStartPosition(int row, int col) {
    this.row = row;
    this.col = col;
}

public int[][][] getFullPosition(){
    int[][][] positions = new int[length][2];
    for (int i = 0; i < length; i++) {
        if (isHorizontal) {
            positions[i][0] = row;
            positions[i][1] = col + i;
        } else {
            positions[i][0] = row + i;
            positions[i][1] = col;
        }
    }
    return positions;
}

public boolean isValidMove(Board board, boolean forward) {
    int newRow = row;
    int newCol = col;

    if (isHorizontal) {
        if(forward){
            newCol++;
        } else{
            newCol--;
        }
    } else {
        if(forward){
            newRow++;
        } else{
            newRow--;
        }
    }
}

if (!board.isInsideBoard(newRow, newCol)) {
```

```

        return false;
    }

    for (int i = 0; i < length; i++) {
        int checkRow = isHorizontal ? newRow : newRow + i;
        int checkCol = isHorizontal ? newCol + i : newCol;

        if (!board.isInsideBoard(checkRow, checkCol) || (!board.getCell(checkRow,
checkCol).isEmpty() && board.getCell(checkRow, checkCol).getSymbol() != symbol)) {
            return false;
        }
    }
    return true;
}

public Board move(Board board, boolean forward) {
    for(int[] position : getFullPosition()){
        board.getCell(position[0], position[1]).clear();
    }

    if (isHorizontal) {
        if(forward){
            col++;
        } else {
            col--;
        }
    } else {
        if(forward){
            row++;
        } else {
            row--;
        }
    }

    for(int[] position : getFullPosition()){
        board.getCell(position[0], position[1]).occupy(symbol);
    }
    return board;
}
}

```

Class PrimaryPiece

Atribut		
Nama	Tipe	Deskripsi

Sama seperti Piece (*inheritance*).

Metode	
Nama	Deskripsi
<code>public PrimaryPiece(int row, int col, char symbol, int length, boolean isHorizontal)</code>	Konstruktor PrimaryPiece, menginisialisasi row, col, symbol, length, dan isHorizontal.
<code>public boolean isValidMove(Board board, boolean forward)</code>	Mengembalikan <i>true</i> jika gerakan yang dilakukan valid.
<code>public Board move(Board board, boolean forward)</code>	Mengerakkan primary piece.
Implementasi Metode	
<pre>public class PrimaryPiece extends Piece { public PrimaryPiece(int row, int col, char symbol, int length, boolean isHorizontal) { super(row, col, symbol, length, isHorizontal); } @Override public boolean isValidMove(Board board, boolean forward) { int newRow = getRow(); int newCol = getCol(); if (isHorizontal()) { if(forward){ newCol++; } else { newCol--; } } else { if(forward){ newRow++; } else { newRow--; } } } Exit exit = board.getExit(); int[] exitPosition = exit.getPoint(); for (int i = 0; i < getLength(); i++) {</pre>	

```

        int row = isHorizontal() ? newRow : newRow + i;
        int col = isHorizontal() ? newCol + i : newCol;

        if(!board.isInsideBoard(row, col)) {
            if(row == exitPosition[0] && col == exitPosition[1]) {
                continue;
            } else{
                return false;
            }
        }

        Cell cell = board.getCell(row, col);
        if(!cell.isEmpty() && cell.getSymbol() != getSymbol()) {
            return false;
        }
    }
    return true;
}

@Override
public Board move(Board board, boolean forward) {
    for(int[] position : getFullPosition()){
        if (board.isInsideBoard(position[0], position[1])) {
            board.getCell(position[0], position[1]).clear();
        }
    }

    if (isHorizontal) {
        if(forward){
            col++;
        } else {
            col--;
        }
    } else {
        if(forward){
            row++;
        } else {
            row--;
        }
    }
}

Exit exit = board.getExit();
int[] exitPoint = exit.getPoint();

int[][][] newPosition = getFullPosition();
boolean hitExit = false;

```

```

        for(int[] position : newPositions){
            if (position[0] == exitPoint[0] && position[1] == exitPoint[1]) {
                hitExit = true;
                break;
            }
        }

        if (hitExit) {
            exit.setHit(true);
        }

        for(int[] position : newPositions){
            if (board.isInsideBoard(position[0], position[1])) {
                board.getCell(position[0], position[1]).occupy(symbol);
            }
        }

        return board;
    }
}

```

4.1.2 Algoritma UCS

Atribut		
Nama	Tipe	Deskripsi
boardChain	ArrayList<Board>	Array list board yang merupakan state solusi dari awal permainan hingga mencapai goal state.
totalNodes	private int	Menyimpan banyaknya node yang sudah dikunjungi
Metode		
Nama	Deskripsi	
public UCS()	Konstruktor UCS, menginisialisasi semua atribut kelas.	
public int getTotalNodes()	Mengembalikan total nodes.	

<code>private String boardSignature(Board board)</code>	Menjadi pembeda antara beberapa board state.
<code>private void buildBoardChain(BoardState goal)</code>	Membentuk boardChain.
<code>public ArrayList<Board> ucsSolver(BoardState object)</code>	Algoritma UCS.
Implementasi Metode	
<pre>public class UCS { ArrayList<Board> boardChain; private int totalNodes; public UCS() { this.boardChain = new ArrayList<>(); this.totalNodes = 0; } public int getTotalNodes() { return totalNodes; } private String boardSignature(Board board){ StringBuilder sb = new StringBuilder(); for (int r = 0; r < board.getRows(); r++) { for (int c = 0; c < board.getCols(); c++) { sb.append(board.getCell(r, c).getSymbol()); } } return sb.toString(); } private void buildBoardChain(BoardState goal){ BoardState current = goal; while (current != null){ boardChain.add(current.getBoard()); current = current.getParent(); } Collections.reverse(boardChain); } public ArrayList<Board> ucsSolver(BoardState object) { totalNodes = 0; PriorityQueue<BoardState> queue = new PriorityQueue<>(Comparator.comparingInt(BoardState::getPathCost)); Set<String> visited = new HashSet<>(); } }</pre>	

```

queue.add(object);

while(!queue.isEmpty()){
    BoardState current = queue.poll();
    Board currentBoard = current.getBoard();

    totalNodes++;

    if(currentBoard.isSolved()){
        buildBoardChain(current);
        break;
    }

    String signature = boardSignature(currentBoard);
    if(visited.contains(signature)){
        continue;
    }
    visited.add(signature);

    for(BoardState next : current.generatePath()){
        queue.add(next);
    }
}
return this.boardChain;
}
}

```

4.1.3 Algoritma GBFS

Atribut		
Nama	Tipe	Deskripsi
boardChain	ArrayList<Board>	Array list board yang merupakan state solusi dari awal permainan hingga mencapai goal state.
totalNodes	private int	Menyimpan total node yang sudah dikunjungi (untuk display).
heuristicType	private string	Menyimpan tipe heuristik yang digunakan untuk membentuk <i>priority queue</i> .

Metode	
Nama	Deskripsi
<code>public GreedyBFS()</code>	Konstruktor Greedy BFS, menginisialisasi semua atribut dengan nilai default.
<code>public int getTotalNodes()</code>	Mengembalikan totalNodes.
<code>public void setHeuristic(String heuristicType)</code>	Mengatur heuristik yang digunakan.
<code>private String boardSignature(Board board)</code>	Menjadi pembeda antara beberapa board state.
<code>public ArrayList<Board> greedyBFSSolver(BoardState startState)</code>	Algoritma <i>greedy best first search</i> .
Implementasi Metode	
<pre>public class GreedyBFS { ArrayList<Board> boardChain; private int totalNodes; private String heuristicType = "Distance to Exit"; public GreedyBFS() { this.boardChain = new ArrayList<Board>(); this.totalNodes = 0; } public int getTotalNodes() { return totalNodes; } public void setHeuristic(String heuristicType) { this.heuristicType = heuristicType; } private String boardSignature(Board board){ StringBuilder sb = new StringBuilder(); for (int r = 0; r < board.getRows(); r++) { for (int c = 0; c < board.getCols(); c++) { sb.append(board.getCell(r, c).getSymbol()); } } return sb.toString(); } }</pre>	

```

}

public ArrayList<Board> greedyBFSSolver(BoardState startState) {
    totalNodes = 0;
    PriorityQueue<BoardState> queue;

    if ("Blocking Vehicles".equals(heuristicType)) {
        queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::getCost));
    } else if ("Combined".equals(heuristicType)) {
        queue = new PriorityQueue<>(Comparator.comparingInt(state ->
state.getBoard().calculateCombinedCost()));
    } else {
        queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::getDistanceCost));
    }
    Set<String> visited = new HashSet<>();

    queue.add(startState);

    while (!queue.isEmpty()) {
        BoardState current = queue.poll();
        Board board = current.getBoard();

        String signature = boardSignature(board);
        if(visited.contains(signature)){
            continue;
        }
        visited.add(signature);
        totalNodes++;

        if (board.isSolved()) {
            ArrayList<Board> result = new ArrayList<>();
            while (current != null) {
                result.add(0, current.getBoard());
                current = current.getParent();
            }
            return result;
        }

        for (BoardState neighbor : current.generatePath()) {
            if (!visited.contains(boardSignature(neighbor.getBoard()))){
                queue.add(neighbor);
            }
        }
    }
}

```

```

        return new ArrayList<>();
    }
}

```

4.1.4 Algoritma A*

Atribut				
Nama	Tipe	Deskripsi		
boardChain	ArrayList<Board>	Array list board yang merupakan state solusi dari awal permainan hingga mencapai goal state.		
totalNodes	private int	Menyimpan total node yang sudah dikunjungi (untuk display).		
heuristicType	private string	Menyimpan tipe heuristik yang digunakan untuk membentuk <i>priority queue</i> .		
Metode				
Nama	Deskripsi			
public AStar()	Konstruktor AStar, menginisialisasi semua atribut.			
public int getTotalNodes()	Mengembalikan totalNodes.			
public void setHeuristic(String heuristicType)	Mengatur tipe heuristik.			
private String boardSignature(Board board)	Menjadi pembeda antara beberapa board state.			
public ArrayList<Board> aStarSolver(BoardState startState)	Algoritma A*.			
Implementasi Metode				
<pre> public class AStar { ArrayList<Board> boardChain; private int totalNodes; private String heuristicType = "Distance to Exit"; </pre>				

```

public AStar() {
    this.boardChain = new ArrayList<Board>();
}

public int getTotalNodes() {
    return this.totalNodes;
}

public void setHeuristic(String heuristicType) {
    this.heuristicType = heuristicType;
}

private String boardSignature(Board board){
    StringBuilder sb = new StringBuilder();
    for (int r = 0; r < board.getRows(); r++) {
        for (int c = 0; c < board.getCols(); c++) {
            sb.append(board.getCell(r, c).getSymbol());
        }
    }
    return sb.toString();
}

public ArrayList<Board> aStarSolver(BoardState startState) {
    this.totalNodes = 0;
    PriorityQueue<BoardState> queue;

    if ("Blocking Vehicles".equals(heuristicType)) {
queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::getAStarCost));
    } else if ("Combined".equals(heuristicType)) {
queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::calculateAStarCombinedCost));
    } else {
queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::getAStarDistanceCost));
    }

    Set<String> visited = new HashSet<>();
    queue.add(startState);

    while (!queue.isEmpty()) {
        BoardState currentNode = queue.poll();
        Board board = currentNode.getBoard();

        String signature = boardSignature(board);
        if (visited.contains(signature)) {

```

```

        continue;
    }
    visited.add(signature);
    this.totalNodes++;

    if (board.isSolved()) {
        ArrayList<Board> result = new ArrayList<>();
        while (currentNode != null) {
            result.add(0, currentNode.getBoard());
            currentNode = currentNode.getParent();
        }
        return result;
    }

    for (BoardState neighbor : currentNode.generatePath()) {
        if (!visited.contains(boardSignature(neighbor.getBoard())))
            queue.add(neighbor);
    }
}

return new ArrayList<>();
}
}

```

4.1.5 Implementasi Bonus: Heuristic Alternatif

Sebagai bagian fitur bonus yang diimplementasikan, disediakan opsi pemilihan heuristic untuk algoritma *informed search*, Greedy Best First Search dan A*. Terdapat tiga jenis heuristic yang dapat dipilih oleh pengguna, yaitu: (1) Distance to Exit, yang mengukur jarak posisi ujung kendaraan utama menuju pintu keluar; (2) Blocking Cars, yang menghitung jumlah kendaraan lain yang menghalangi jalur kendaraan utama menuju pintu keluar; serta (3) Combined, yaitu gabungan dari kedua heuristic sebelumnya untuk menghasilkan penilaian yang lebih komprehensif terhadap kedekatan sebuah state terhadap solusi.

```

public ArrayList<Board> greedyBFSSolver(BoardState startState) {
    totalNodes = 0;
    PriorityQueue<BoardState> queue;

    if ("Blocking Vehicles".equals(heuristicType)) {
        queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::getCost));
    } else if ("Combined".equals(heuristicType)) {
        queue = new PriorityQueue<>(Comparator.comparingInt(state ->

```

```

state.getBoard().calculateCombinedCost()));
} else {
    queue = new
PriorityQueue<>(Comparator.comparingInt(BoardState::getDistanceCost));
}

```

4.1.6 Implementasi Bonus: GUI

Atribut		
Nama	Tipe	Deskripsi
pieceColors	private final Map<Character, Color>	Map seluruh karakter yang terdapat dalam papan dan warna yang dimilikin
currentBoard	private Board	Kondisi awal papan setelah parsing yang akan digunakan oleh fungsi lainnya
CELL_SIZE	private int	Ukuran sel pada papan, standar 60 pixel
currentSolution	private ArrayList<Board>	Solusi papan setelah dijalankan melalui algoritma tertentu
executionTime	private long	Waktu yang digunakan untuk eksekusi algoritma
totalNodesExpanded	private int	Banyak node yang ditelusuri algoritma
algorithmUsed	private String	Algoritma yang digunakan untuk menyelesaikan papan
heuristicUsed	private String	Heuristic yang digunakan untuk algoritma GBFS atau A*
Metode		
Nama	Deskripsi	
public void initialize(URL location, ResourceBundle	Menginisialisasi tampilan utama	

<code>resources)</code>	web dengan dropbox algoritma, heuristic, tombol load puzzle, solve, dan warna piece
<code>private void initializePieceColors()</code>	Menetapkan warna untuk setiap piece
<code>private void handleLoadPuzzle()</code>	Menginisialisasi Board menggunakan Parser dan FileChooser kemudian melakukan rendering Board
<code>private void handleSolve()</code>	Melakukan solving dari suatu Board berdasarkan Algoritma dan heuristic yang dipilih pengguna, dan menetapkan stats ke atribut bersangkutan
<code>private void simulateSolution(List<Board> solution)</code>	Melakukan simulasi solusi dengan menunjukkan setiap board dari kondisi awal hingga kondisi solusi
<code>public void renderBoard(Board board)</code>	Melakukan rendering board, dengan setiap simbol yang ditempati piece
<code>private String toRgbString(Color color)</code>	Mengubah suatu warna menjadi string angka RGB
<code>private void showAlert(String message)</code>	Menunjukkan popup window error
<code>private void handleSaveSolution()</code>	Menyimpan suatu solusi ke dalam txt
Implementasi Metode	
<pre>public class RushHourController implements Initializable { // Mapping warna private final Map<Character, Color> pieceColors = new HashMap<>(); private Board currentBoard; private static final int CELL_SIZE = 60; private ArrayList<Board> currentSolution; private long executionTime;</pre>	

```

private int totalNodesExpanded;
private String algorithmUsed;
private String heuristicUsed;

@Override
public void initialize(URL location, ResourceBundle resources) {
    algorithmComboBox.setItems(FXCollections.observableArrayList(
        "UCS", "A*", "Greedy Best-First"
    ));
    algorithmComboBox.getSelectionModel().selectFirst();

    heuristicComboBox.setItems(FXCollections.observableArrayList(
        "Distance to Exit", "Blocking Vehicles", "Combined"
    ));
    heuristicComboBox.getSelectionModel().selectFirst();

    heuristicLabel.setVisible(false);
    heuristicComboBox.setVisible(true);

    algorithmComboBox.getSelectionModel().selectedItemProperty().addListener(
        (observable, oldValue, newValue) -> {
            boolean isAStar = "A*".equals(newValue) || "Greedy
Best-First".equals(newValue);
            heuristicLabel.setVisible(isAStar);
            heuristicComboBox.setVisible(isAStar);
        }
    );
    initializePieceColors();
}

private void initializePieceColors() {
    pieceColors.put('P', Color.RED);

    pieceColors.put('A', Color.BLUE);
    pieceColors.put('B', Color.GREEN);
    pieceColors.put('C', Color.ORANGE);
    pieceColors.put('D', Color.PURPLE);
    pieceColors.put('E', Color.TEAL);
    pieceColors.put('F', Color.PINK);
    pieceColors.put('G', Color.DARKBLUE);
    pieceColors.put('H', Color.DARKGREEN);
    pieceColors.put('I', Color.DARKORANGE);
    pieceColors.put('J', Color.DARKRED);
    pieceColors.put('K', Color.DARKVIOLET);
    pieceColors.put('L', Color.DARKCYAN);
    pieceColors.put('M', Color.MAGENTA);
}

```

```

        pieceColors.put('N', Color.NAVY);
        pieceColors.put('O', Color.OLIVE);

        pieceColors.put('.', Color.LIGHTGRAY);
    }

@FXML
private void handleLoadPuzzle() {
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Open Puzzle File");
    fileChooser.getExtensionFilters().add(
        new FileChooser.ExtensionFilter("Text Files", "*.txt")
    );
}

File selectedFile =
fileChooser.showOpenDialog(boardGridPane.getScene().getWindow());
if (selectedFile != null) {
    try {
        BoardState boardState = Parser.parse(selectedFile);

        currentBoard = boardState.getBoard();
        renderBoard(currentBoard);
        statusLabel.setText("Puzzle loaded: " + selectedFile.getName());

    } catch (IOException e) {
        showAlert("Error loading file: " + e.getMessage());
    } catch (NumberFormatException e) {
        showAlert("Invalid file format. Expected dimensions in first
line.");
    } catch (Exception e) {
        showAlert("Error: " + e.getMessage());
    }
}
}

@FXML
private void handleSolve() {
    if (currentBoard == null) {
        showAlert("Please load a puzzle first.");
        return;
    }

    String algorithm = algorithmComboBox.getValue();
    String heuristic = heuristicComboBox.getValue();

    solveButton.setDisable(true);
    statusLabel.setText("Solving with " + algorithm + "...");
}

```

```

BoardState initialState = new BoardState(currentBoard, 0, null);

ArrayList<Board> solution = null;
totalNodesExpanded = 0;
long startTime = System.currentTimeMillis();

try {
    switch (algorithm) {
        case "UCS":
            UCS ucs = new UCS();
            solution = ucs.ucsSolver(initialState);
            algorithmUsed = "UCS";
            heuristicUsed = "N/A";
            totalNodesExpanded = ucs.getTotalNodes();
            break;
        case "A*":
            AStar aStar = new AStar();
            aStar.setHeuristic(heuristic);
            solution = aStar.aStarSolver(initialState);
            totalNodesExpanded = aStar.getTotalNodes();
            algorithmUsed = "A*";
            heuristicUsed = heuristic;
            break;
        case "Greedy Best-First":
            GreedyBFS greedy = new GreedyBFS();
            greedy.setHeuristic(heuristic);
            solution = greedy.greedyBFSSolver(initialState);
            totalNodesExpanded = greedy.getTotalNodes();
            algorithmUsed = "Greedy Best-First";
            heuristicUsed = heuristic;
            break;
        default:
            showAlert("Unknown algorithm: " + algorithm);
            return;
    }
    long endTime = System.currentTimeMillis();
    executionTime = endTime - startTime;
    if (solution == null || solution.isEmpty()) {
        statusLabel.setText("No solution found!");
        saveSolutionButton.setDisable(true);
        solveButton.setDisable(false);
        statsGridPane.setVisible(false);
    } else {
        statusLabel.setText(String.format("Solution found in %d steps! Time: %d ms, Nodes expanded: %d",

```

```

        (solution.size()-1), executionTime, totalNodesExpanded));
currentSolution = new ArrayList<>(solution);
saveSolutionButton.setDisable(false);

algorithmLabel.setText(algorithmUsed);
heuristicValueLabel.setText(heuristicUsed);
timeLabel.setText(executionTime + " ms");
nodesLabel.setText(String.valueOf(totalNodesExpanded));
statsGridPane.setVisible(true);

simulateSolution(solution);
}
} catch (Exception e) {
showAlert("Error during solving: " + e.getMessage());
e.printStackTrace();
solveButton.setDisable(true);
}
}

private void simulateSolution(List<Board> solution) {
if (solution == null || solution.isEmpty()) {
return;
}

solveButton.setDisable(true);

final int[] currentStepIndex = {0};
Timeline timeline = new Timeline();

for (int i = 0; i < solution.size(); i++) {
final int stepIndex = i;
KeyFrame keyFrame = new KeyFrame(Duration.seconds(i * 0.5), event -> {
Board board = solution.get(stepIndex);
renderBoard(board);
currentStepIndex[0] = stepIndex;

statusLabel.setText("Step " + stepIndex + " of " +
(solution.size()-1));

if (board.isSolved()) {
statusLabel.setText("Puzzle solved in " + (solution.size()-1) +
" steps!");
}
});

timeline.getKeyFrames().add(keyFrame);
}
}

```

```

        timeline.setOnFinished(event -> {
            solveButton.setDisable(false);
        });

        timeline.play();
    }

    public void renderBoard(Board board) {
        if (board == null) return;

        boardGridPane.getChildren().clear();

        int rows = board.getRows();
        int cols = board.getCols();

        boardGridPane.getColumnConstraints().clear();
        boardGridPane.getRowConstraints().clear();

        for (int i = 0; i < cols; i++) {
            ColumnConstraints colConstraints = new ColumnConstraints();
            colConstraints.setMinWidth(CELL_SIZE);
            colConstraints.setPrefWidth(CELL_SIZE);
            boardGridPane.getColumnConstraints().add(colConstraints);
        }

        for (int i = 0; i < rows; i++) {
            RowConstraints rowConstraints = new RowConstraints();
            rowConstraints.setMinHeight(CELL_SIZE);
            rowConstraints.setPrefHeight(CELL_SIZE);
            boardGridPane.getRowConstraints().add(rowConstraints);
        }

        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                app.components.Cell cell = board.getCell(row, col);
                char cellValue = cell.getSymbol();

                StackPane cellPane = new StackPane();
                cellPane.getStyleClass().add("cell");

                // Set warna
                Color pieceColor = pieceColors.getOrDefault(cellValue, Color.GRAY);
                cellPane.setStyle(String.format(
                    "-fx-background-color: %s; -fx-border-color: black;
-fx-border-width: 1px;",

```

```

        toRgbString(pieceColor)
    ));

    if (cellValue != '.') {
        //Label label = new Label(String.valueOf(cellValue));
        //label.setStyle("-fx-font-weight: bold; -fx-text-fill:
white;");
        //cellPane.getChildren().add(label);
    }

    boardGridPane.add(cellPane, col, row);
}
}

Exit exit = board.getExit();
if (exit != null) {
    int exitRow = exit.getRow();
    int exitCol = exit.getCol();

    StackPane exitMarker = new StackPane();
    exitMarker.getStyleClass().add("exit-marker");

    // Exit marker
    if (exitRow >= rows) {
        boardGridPane.add(exitMarker, exitCol, rows - 1);
    } else if (exitRow < 0) {
        boardGridPane.add(exitMarker, exitCol, 0);
    } else if (exitCol >= cols) {
        boardGridPane.add(exitMarker, cols - 1, exitRow);
    } else if (exitCol < 0) {
        boardGridPane.add(exitMarker, 0, exitRow);
    }
}

private String toRgbString(Color color) {
    return String.format(
        "rgb(%d, %d, %d)",
        (int) (color.getRed() * 255),
        (int) (color.getGreen() * 255),
        (int) (color.getBlue() * 255)
    );
}

private void showAlert(String message) {

```

```

        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Error");
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }

    @FXML
    private void handleSaveSolution() {
        if (currentSolution == null || currentSolution.isEmpty()) {
            showAlert("No solution to save. Please solve the puzzle first.");
            return;
        }

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

File selectedFile = null;
fileChooser.showSaveDialog(boardGridPane.getScene().getWindow());
if (selectedFile != null) {
    try (java.io.PrintWriter writer = new java.io.PrintWriter(selectedFile)) {
        writer.println("Algorithm: " + algorithmUsed);
        if (!"N/A".equals(heuristicUsed)) {
            writer.println("Heuristic: " + heuristicUsed);
        }
        writer.println("Execution Time: " + executionTime + " ms");
        writer.println("Nodes Expanded: " + totalNodesExpanded);
        writer.println("Solution Steps: " + (currentSolution.size() - 1));
        writer.println();
        int step = 0;
        for (Board board : currentSolution) {
            writer.println("Step " + step++);

            for (int r = 0; r < board.getRows(); r++) {
                for (int c = 0; c < board.getCols(); c++) {
                    writer.print(board.getCell(r, c).getSymbol());
                }
                writer.println();
            }
            writer.println("Exit: " + board.getExit().getRow() + ", " +
                board.getExit().getCol());
            writer.println();
        }
    }
}

```

```
        }
    }

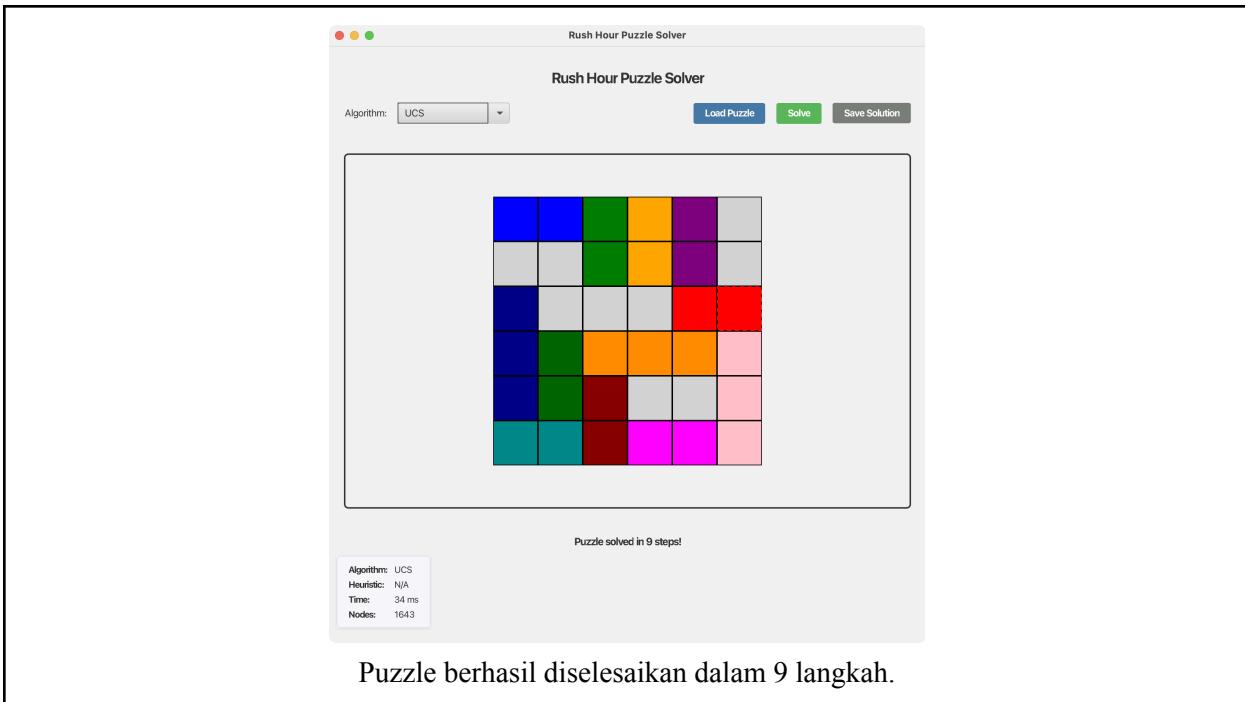
    statusLabel.setText("Solution saved to " + selectedFile.getName());
} catch (IOException e) {
    showAlert("Error saving solution: " + e.getMessage());
}
}
```

4.2 Pengujian

4.2.1 Algoritma UCS

Kasus 1 : Kasus penyelesaian normal, *exit* di kanan.

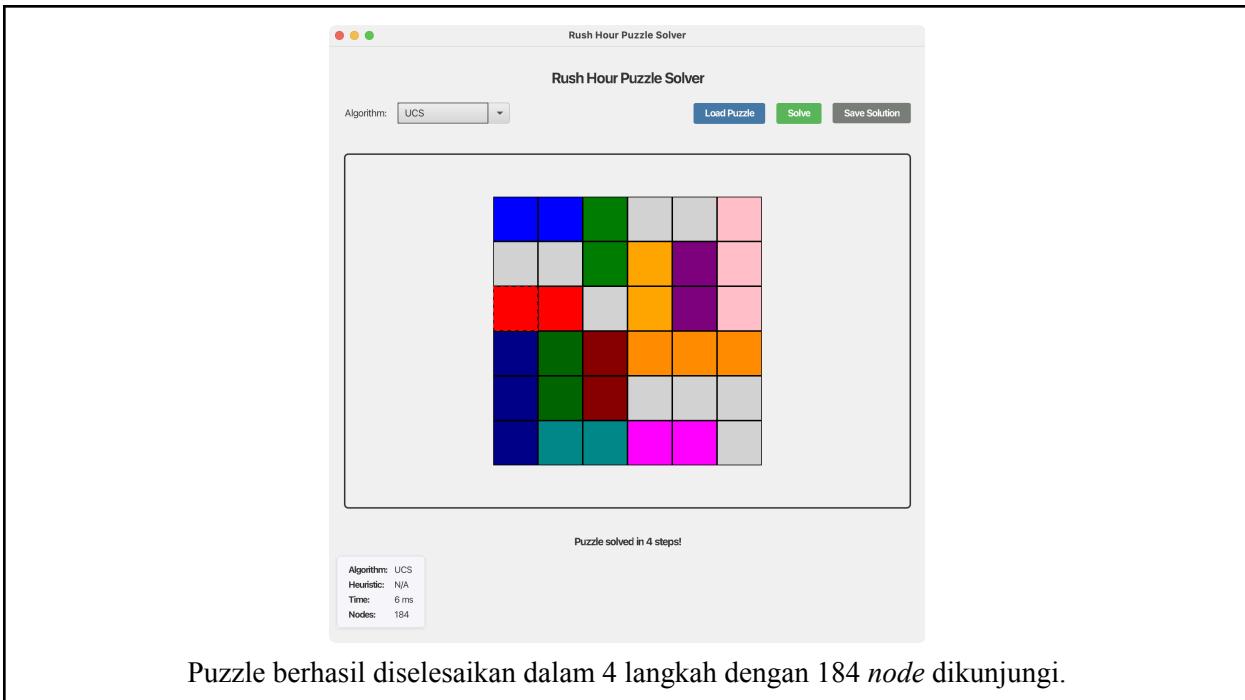
Kasus	
Dalam file .txt	GUI
<pre>6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.</pre>	<p>Rush Hour Puzzle Solver</p> <p>Algorithm: UCS</p> <p>Load Puzzle Solve Save Solution</p> <p>Puzzle loaded: test.txt</p>



Puzzle berhasil diselesaikan dalam 9 langkah.

Kasus 2 : Kasus penyelesaian normal, *exit* di kiri.

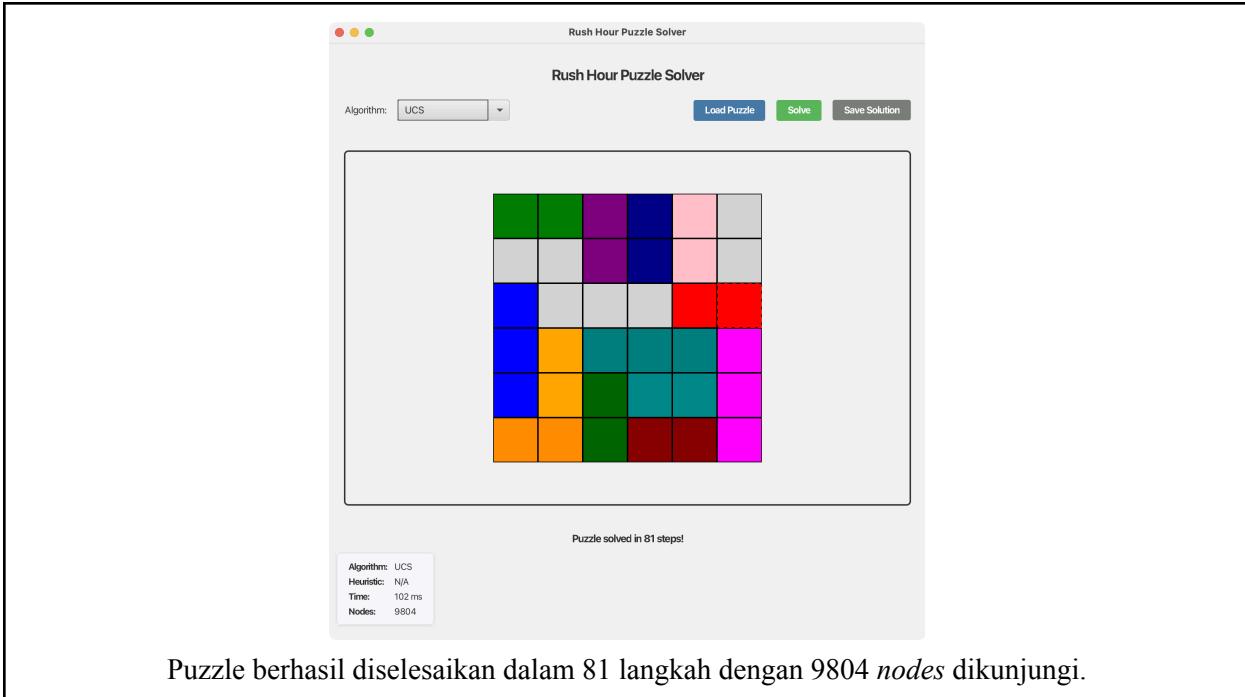
Kasus	
Dalam file .txt	GUI
<pre>6 6 11 AAB..F ..BCDF KGPPCDF GH.III GHJ... LLJMM.</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. At the top, there's a menu bar with 'File', 'Edit', 'Help', and a 'Rush Hour' dropdown. Below the menu is a toolbar with 'Algorithm: UCS' (selected), 'Load Puzzle', 'Solve', and 'Save Solution'. The main area contains a 6x6 grid representing the puzzle board. The board has various colored cars (blue, green, yellow, purple, red, orange, pink, cyan, magenta) and empty slots. A message at the bottom of the grid says 'Puzzle loaded: test.txt'. Below the grid, a status bar displays: Algorithm: UCS, Heuristic: N/A, Time: 6 ms, Nodes: 184.</p>
Solusi	



Puzzle berhasil diselesaikan dalam 4 langkah dengan 184 *node* dikunjungi.

Kasus 3 : Kasus konfigurasi 6×6 tersulit menurut Michael Fogelman.

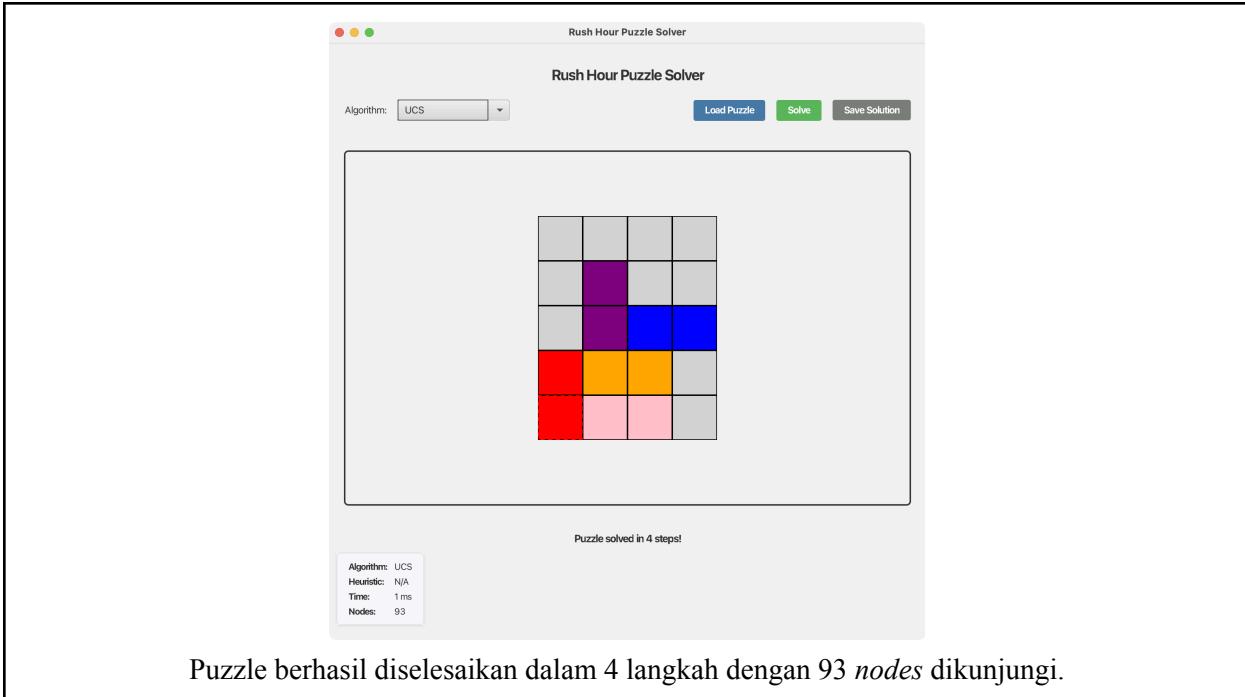
Kasus	
Dalam file .txt	GUI
<pre>6 6 12 ABB.F. ACD.FM ACDPPMK EEEG.M ..HGLL IIHJJ.</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. The title bar says 'Rush Hour Puzzle Solver'. The main area contains a 6x6 grid representing the puzzle board. The grid has various colored cars and empty slots. A message at the bottom of the grid says 'Puzzle loaded: test.txt'. Below the grid, a status bar displays algorithm information: Algorithm: UCS, Heuristic: N/A, Time: 102 ms, Nodes: 9904.</p>
Solusi	



Puzzle berhasil diselesaikan dalam 81 langkah dengan 9804 *nodes* dikunjungi.

Kasus 4 : Kasus *primary piece* tidak horizontal dengan konfigurasi selain 6×6 .

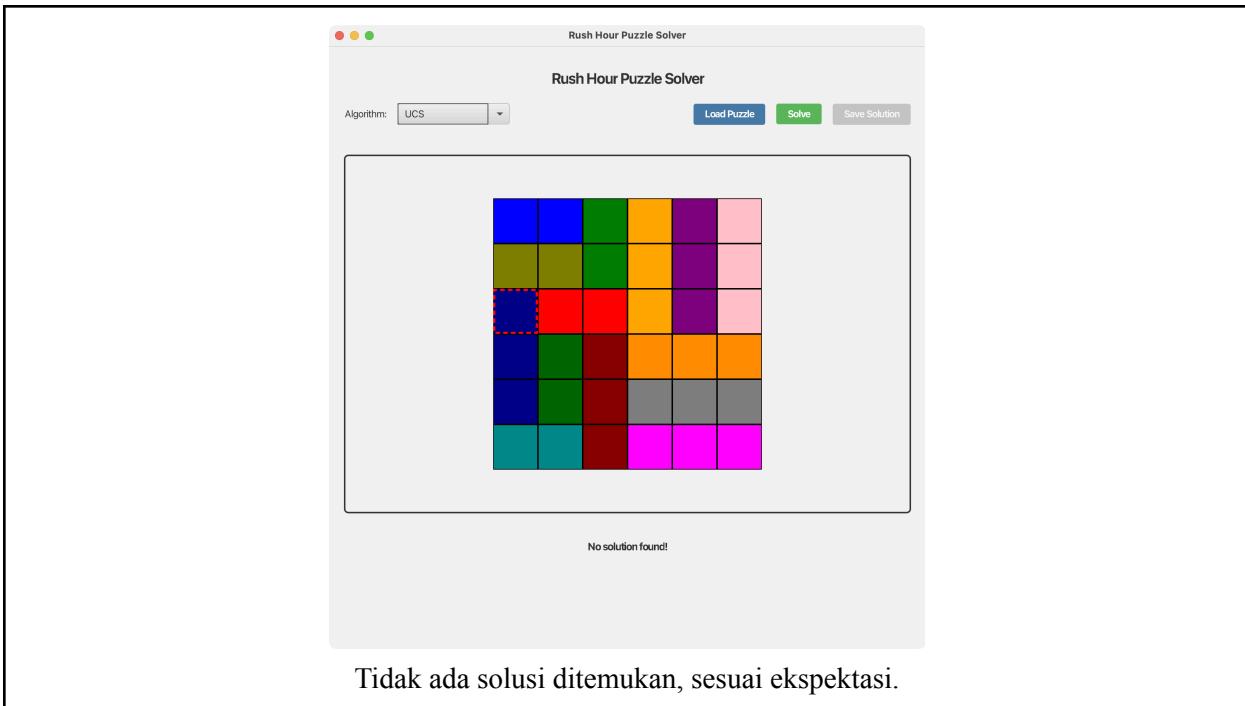
Kasus	
Dalam file .txt	GUI
<pre>5 4 4 PD.. PDAA CC.. FF.. K</pre>	<p>Rush Hour Puzzle Solver</p> <p>Rush Hour Puzzle Solver</p> <p>Algorithm: UCS</p> <p>Heuristic: N/A</p> <p>Time: 102 ms</p> <p>Nodes: 9804</p> <p>Puzzle loaded: test.txt</p> <p>Algorithm: Greedy Best-First</p> <p>Heuristic: Combined</p> <p>Time: 0 ms</p> <p>Nodes: 7</p>
	<p>Solusi</p>



Puzzle berhasil diselesaikan dalam 4 langkah dengan 93 *nodes* dikunjungi.

Kasus 5 : Kasus *unsolvable*.

Kasus	
Dalam file .txt	GUI
<pre>6 6 13 AABCDF 00BCDF KGPPCDF GHJIII GHJRRR LLJMMM</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. The title bar says 'Rush Hour Puzzle Solver'. The toolbar includes 'Algorithm' (set to 'UCS'), 'Load Puzzle', 'Solve', and 'Save Solution'. The main area has a 6x6 grid with various colored cars: blue, green, yellow, purple, red, and pink. A blue car is at the top-left, a green car is above it, a yellow car is to the right, a purple car is further right, and a red car is at the bottom-left. A pink car is at the bottom-right. A blue border surrounds the board. Below the board, a message says 'Puzzle loaded: test.txt'. At the bottom of the window, there's a status bar with algorithm information: Algorithm: UCS, Heuristic: N/A, Time: 1ms, Nodes: 93.</p>
Solusi	



Tidak ada solusi ditemukan, sesuai ekspektasi.

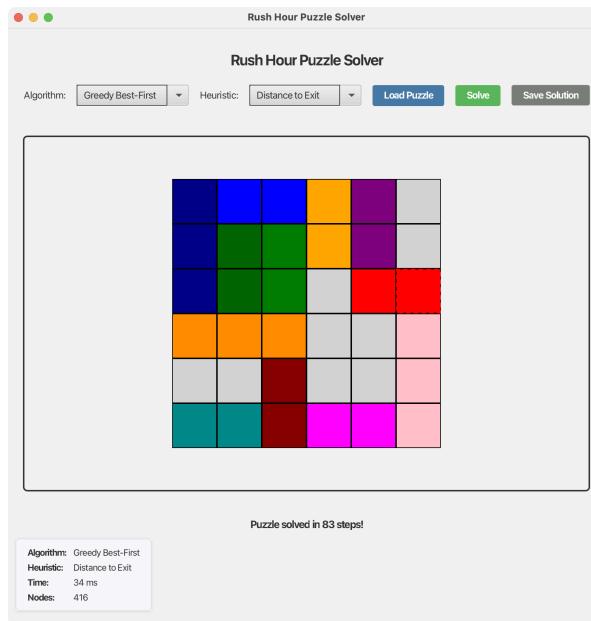
4.2.2 Algoritma GBFS

Kasus 1 : Kasus penyelesaian normal, *exit* di kanan.

Kasus	
Dalam file .txt	GUI
<pre>6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.</pre>	<p>Rush Hour Puzzle Solver</p> <p>Algorithm: Greedy Best-First Heuristic: Distance to Exit Load Puzzle Solve Save Solution</p> <p>Puzzle loaded: test.txt</p> <p>Algorithm: A* Heuristic: Combined Time: 15 ms Nodes: 389</p>

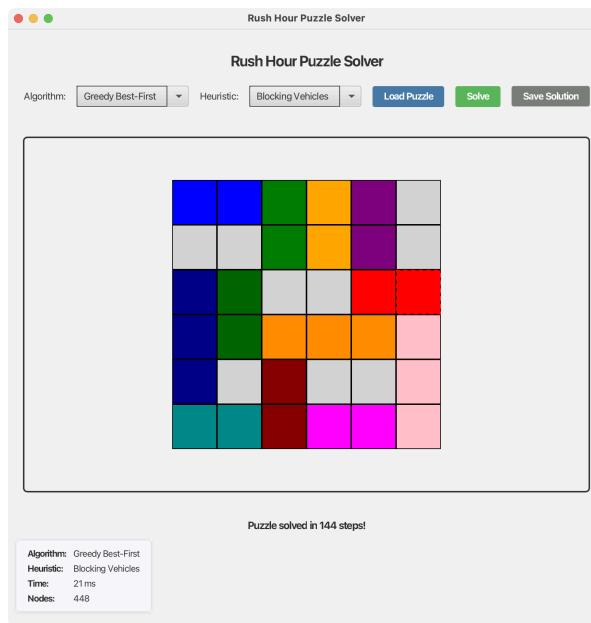
Solusi

Heuristik : *Distance to Exit*



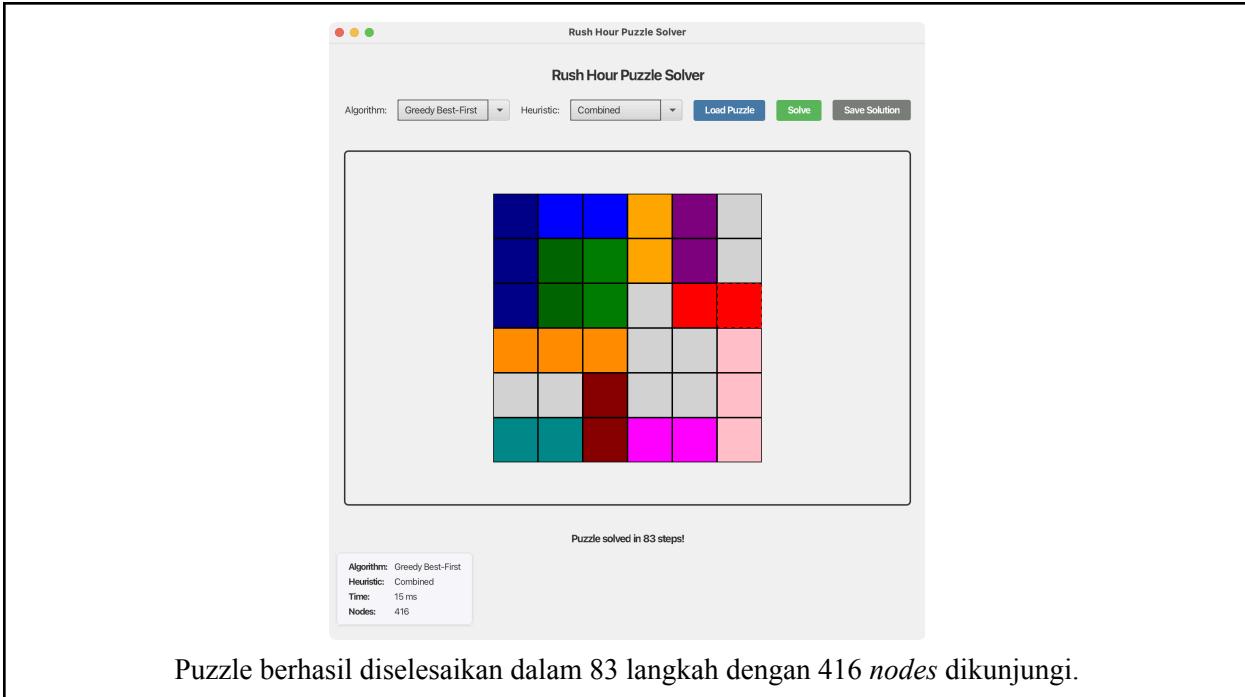
Puzzle berhasil diselesaikan dalam 83 langkah dengan 416 *nodes* dikunjungi.

Heuristik : *Blocking Vehicles*



Puzzle berhasil diselesaikan dalam 144 langkah dengan 448 *nodes* dikunjungi.

Heuristik : *Combined*

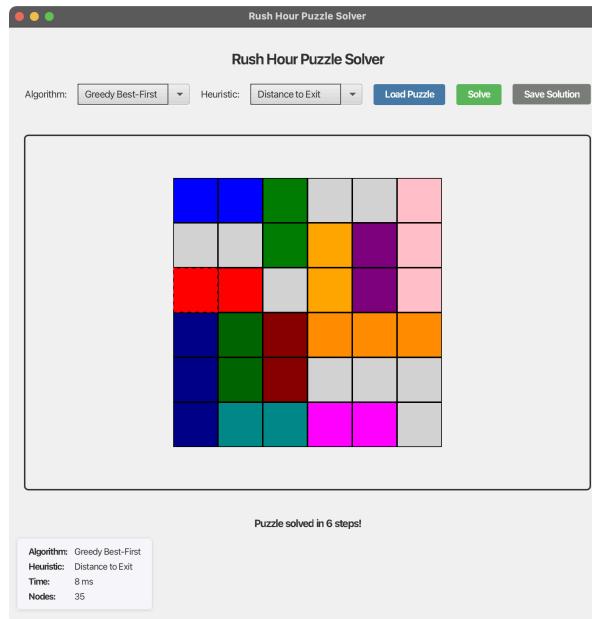


Puzzle berhasil diselesaikan dalam 83 langkah dengan 416 *nodes* dikunjungi.

Kasus 2 : Kasus penyelesaian normal, *exit* di kiri.

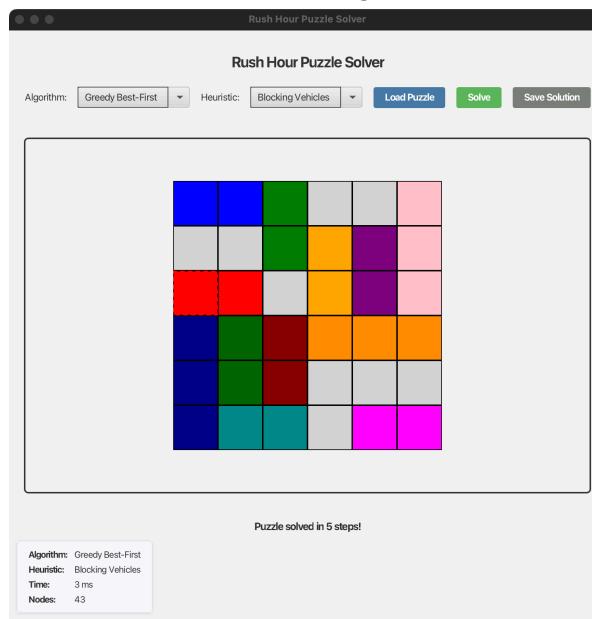
Kasus	
Dalam file .txt	GUI
<pre>6 6 11 AAB..F ..BCDF KGPPCDF GH.III GHJ... LLJMM.</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. The title bar says 'Rush Hour Puzzle Solver'. At the top, there are dropdown menus for 'Algorithm' (set to 'Greedy Best-First') and 'Heuristic' (set to 'Combined'), and buttons for 'Load Puzzle', 'Solve', and 'Save Solution'. Below the controls is a 6x6 grid representing the puzzle board. The grid contains various colored blocks: blue, green, yellow, purple, orange, red, pink, teal, and white. A red car is located at the bottom-left corner. The message 'Puzzle loaded: test.txt' is displayed below the grid. At the bottom left, a status bar shows: Algorithm: UCS, Heuristic: N/A, Time: 6 ms, and Nodes: 184.</p>
Solusi	

Heuristik : Distance to Exit



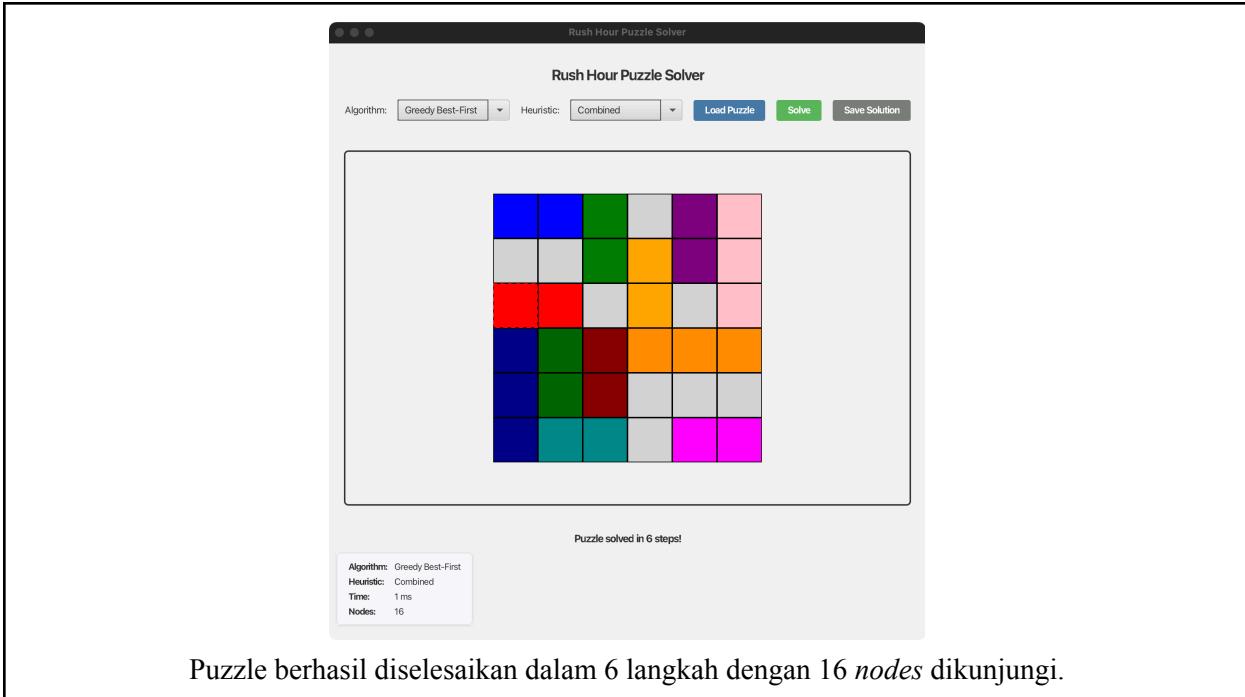
Puzzle berhasil diselesaikan dalam 6 langkah dengan 35 *nodes* dikunjungi.

Heuristik : Blocking Vehicles



Puzzle berhasil diselesaikan dalam 5 langkah dengan 43 *nodes* dikunjungi.

Heuristik : Combined

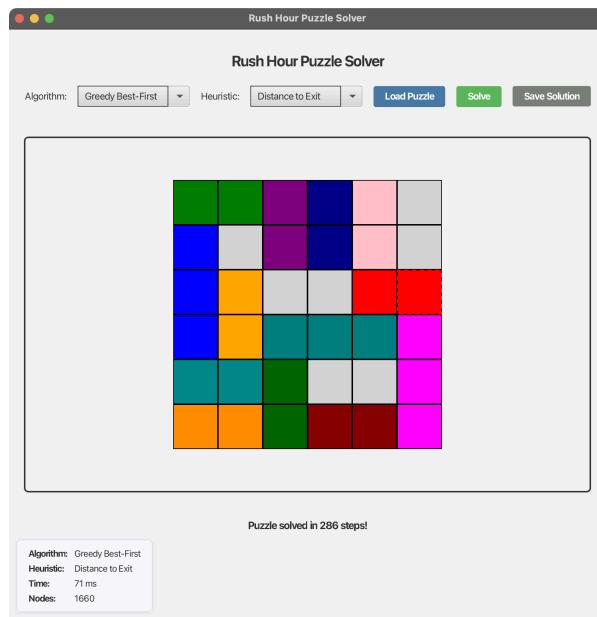


Puzzle berhasil diselesaikan dalam 6 langkah dengan 16 *nodes* dikunjungi.

Kasus 3 : Kasus konfigurasi 6×6 tersulit menurut Michael Fogelman.

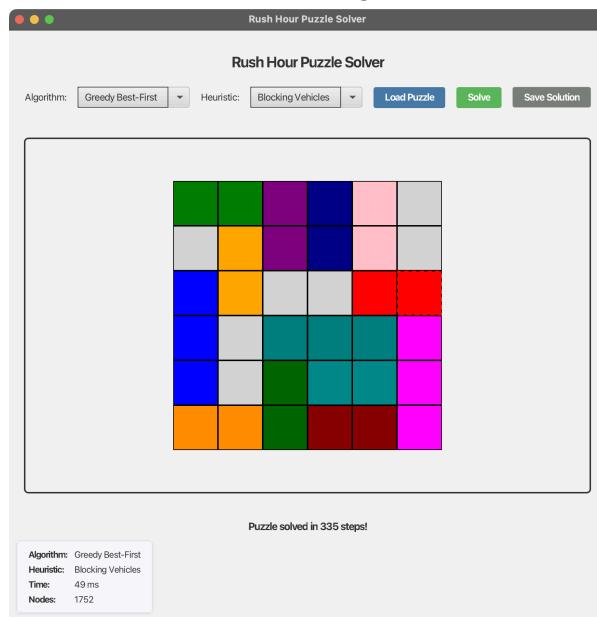
Kasus	
Dalam file .txt	GUI
<pre>6 6 12 ABB.F. ACD.FM ACDPPMK EEEG.M ..HGLL IIHJJ.</pre>	<p>Rush Hour Puzzle Solver</p> <p>Algorithm: Greedy Best-First Heuristic: Combined Load Puzzle Solve Save Solution</p> <p>Puzzle loaded: test.txt</p> <p>Algorithm: UCS Heuristic: N/A Time: 102 ms Nodes: 9904</p>
Solusi	

Heuristik : Distance to Exit



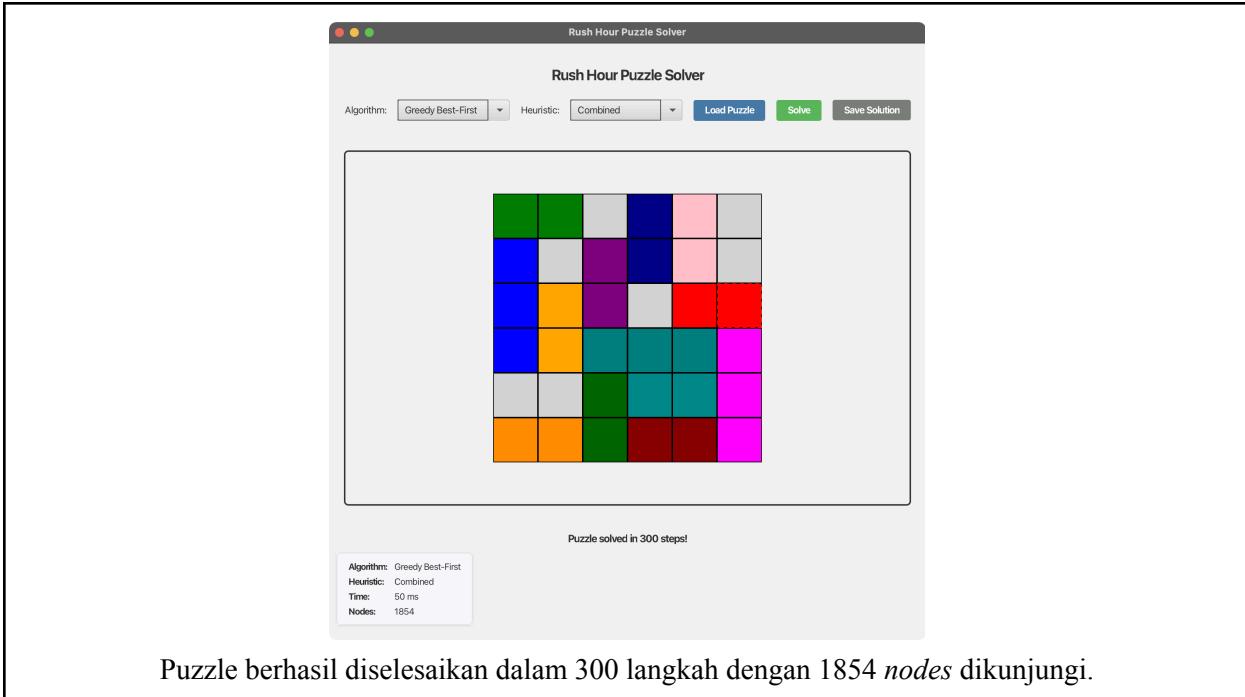
Puzzle berhasil diselesaikan dalam 286 langkah dengan 1660 *nodes* dikunjungi.

Heuristik : Blocking Vehicles



Puzzle berhasil diselesaikan dalam 335 langkah dengan 1752 *nodes* dikunjungi.

Heuristik : Combined

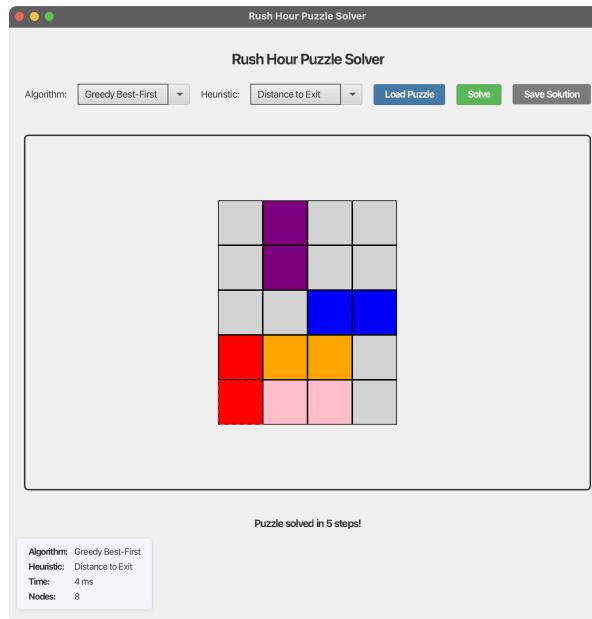


Puzzle berhasil diselesaikan dalam 300 langkah dengan 1854 nodes dikunjungi.

Kasus 4 : Kasus *primary piece* tidak horizontal dengan konfigurasi selain 6×6 .

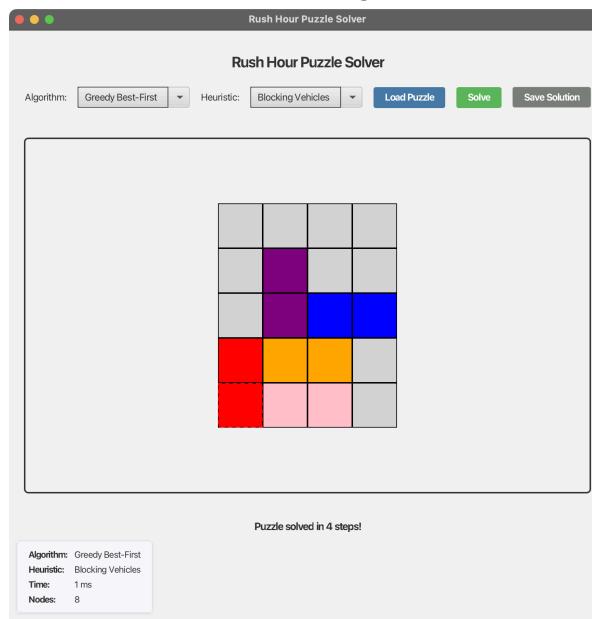
Kasus	
Dalam file .txt	GUI
<pre>5 4 4 PD.. PDAA CC.. FF.. K</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. The title bar says 'Rush Hour Puzzle Solver'. At the top, there are dropdown menus for 'Algorithm' (set to 'Greedy Best-First') and 'Heuristic' (set to 'Distance to Exit'), followed by buttons for 'Load Puzzle', 'Solve', and 'Save Solution'. Below this is a 5x6 grid representing the puzzle board. The grid contains colored blocks: red, purple, blue, orange, yellow, and pink. A message at the bottom of the grid area says 'Puzzle loaded: test.txt'. The status bar at the bottom shows: Algorithm: Greedy Best-First, Heuristic: Distance to Exit, and Time: 50 ms.</p>
Solusi	

Heuristik : Distance to Exit



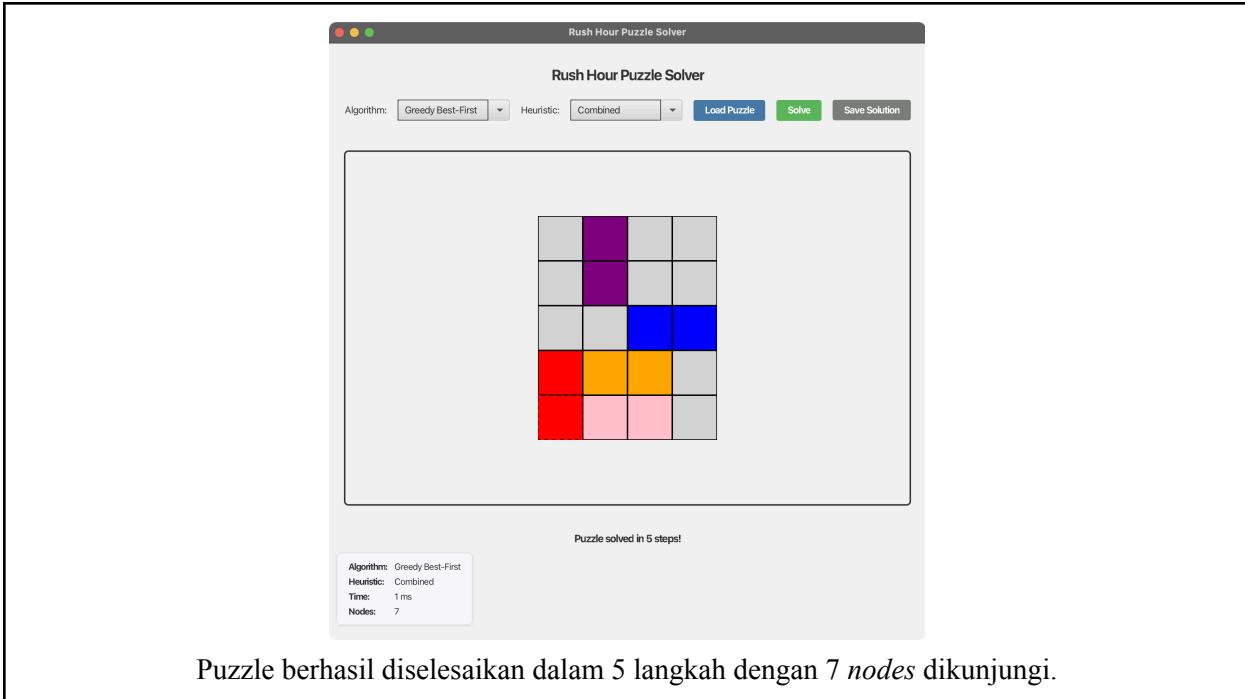
Puzzle berhasil diselesaikan dalam 5 langkah dengan 8 *nodes* dikunjungi.

Heuristik : Blocking Vehicles



Puzzle berhasil diselesaikan dalam 4 langkah dengan 8 *nodes* dikunjungi.

Heuristik : Combined

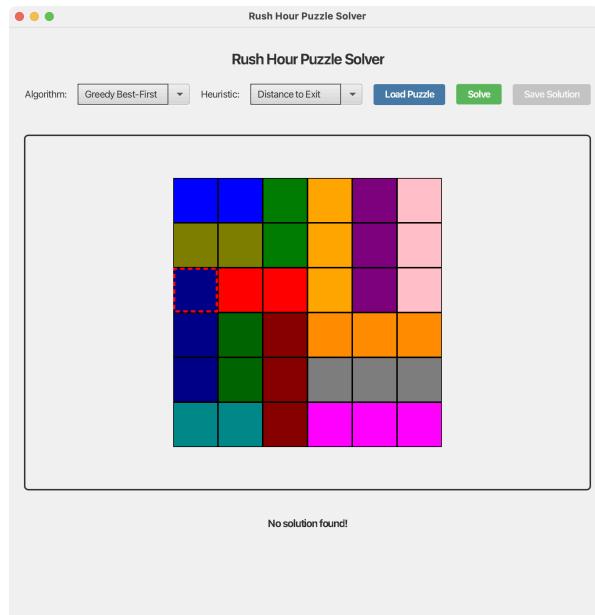


Puzzle berhasil diselesaikan dalam 5 langkah dengan 7 nodes dikunjungi.

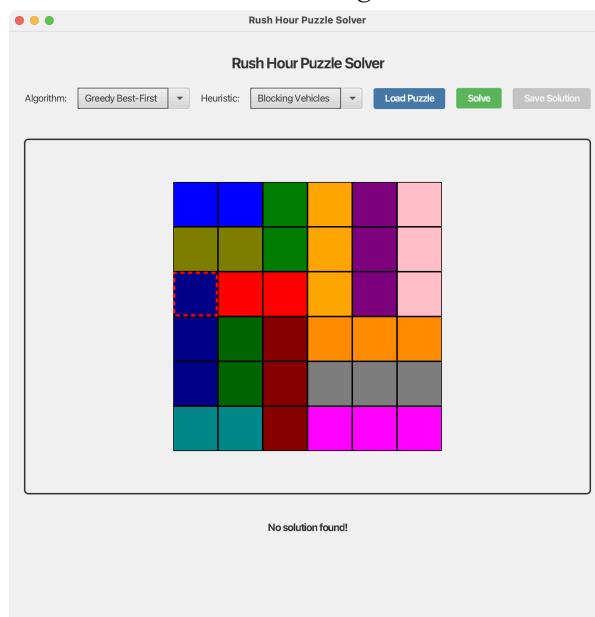
Kasus 5 : Kasus *unsolvable*.

Kasus	
Dalam file .txt	GUI
<pre>6 6 13 AABCDF 00BCDF KGPPCDF GHJIII GHJR LLJMM</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. The title bar says 'Rush Hour Puzzle Solver'. The algorithm dropdown is set to 'UCS'. Below the grid, a message says 'Puzzle loaded: test.txt'. The 6x6 grid contains various colored blocks: blue, green, orange, purple, red, and pink. A red dashed outline highlights a cluster of blue, green, and red blocks in the center-left area. The status bar at the bottom of the window is empty.</p>
Solusi	

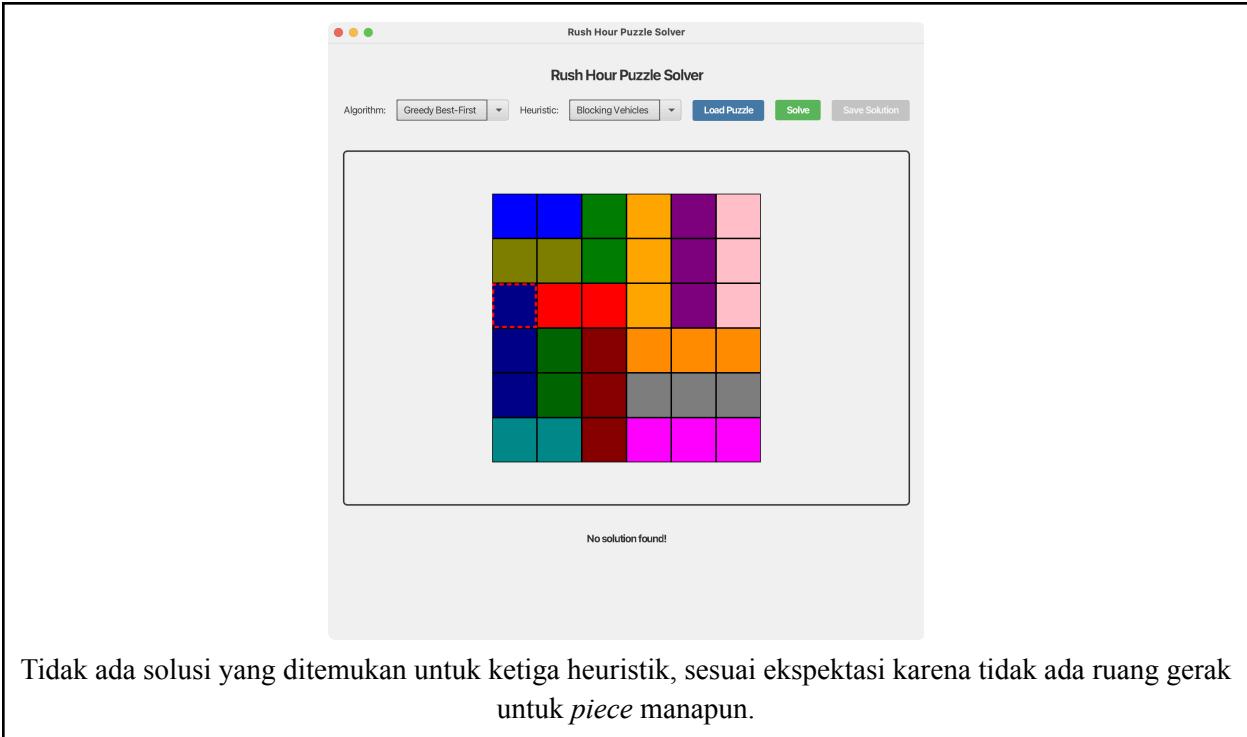
Heuristik : *Distance to Exit*



Heuristik : *Blocking Vehicles*



Heuristik : *Combined*



Tidak ada solusi yang ditemukan untuk ketiga heuristik, sesuai ekspektasi karena tidak ada ruang gerak untuk *piece* manapun.

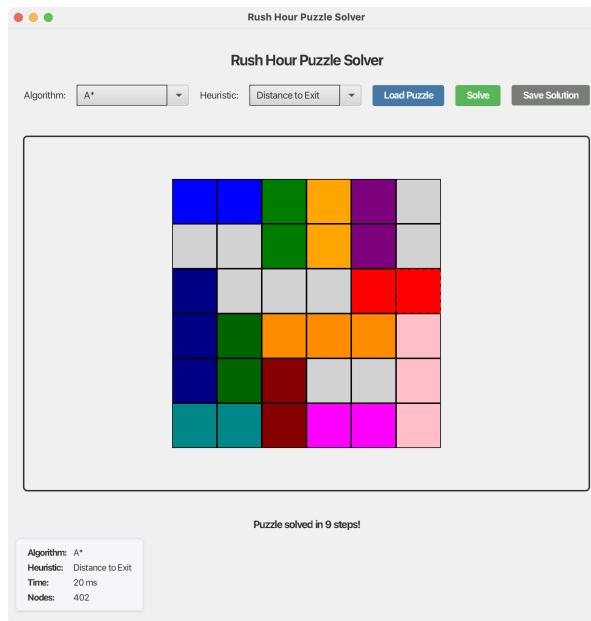
4.2.3 Algoritma A*

Kasus 1 : Kasus penyelesaian normal, *exit* di kanan.

Kasus									
Dalam file .txt	GUI								
<pre>6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.</pre>	<p>Rush Hour Puzzle Solver</p> <p>Algorithm: A* Heuristic: Combined Load Puzzle Solve Save Solution</p> <table border="1"> <tr> <td>Algorithm:</td> <td>UCS</td> </tr> <tr> <td>Heuristic:</td> <td>N/A</td> </tr> <tr> <td>Time:</td> <td>34 ms</td> </tr> <tr> <td>Nodes:</td> <td>1643</td> </tr> </table>	Algorithm:	UCS	Heuristic:	N/A	Time:	34 ms	Nodes:	1643
Algorithm:	UCS								
Heuristic:	N/A								
Time:	34 ms								
Nodes:	1643								

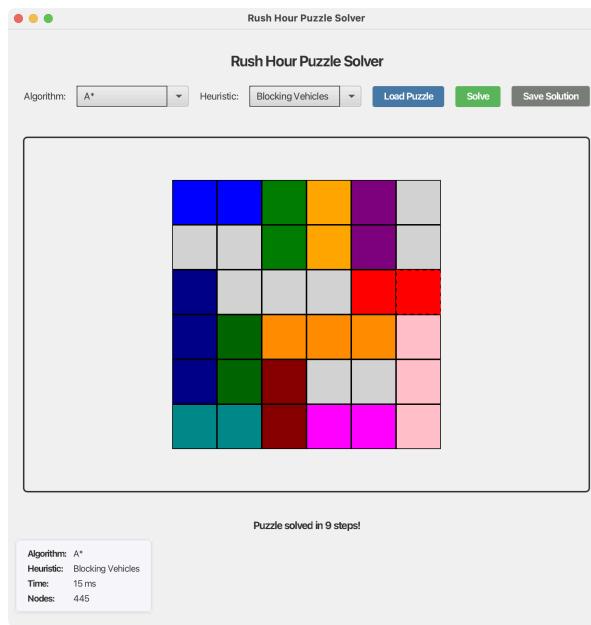
Solusi

Heuristik : *Distance to Exit*



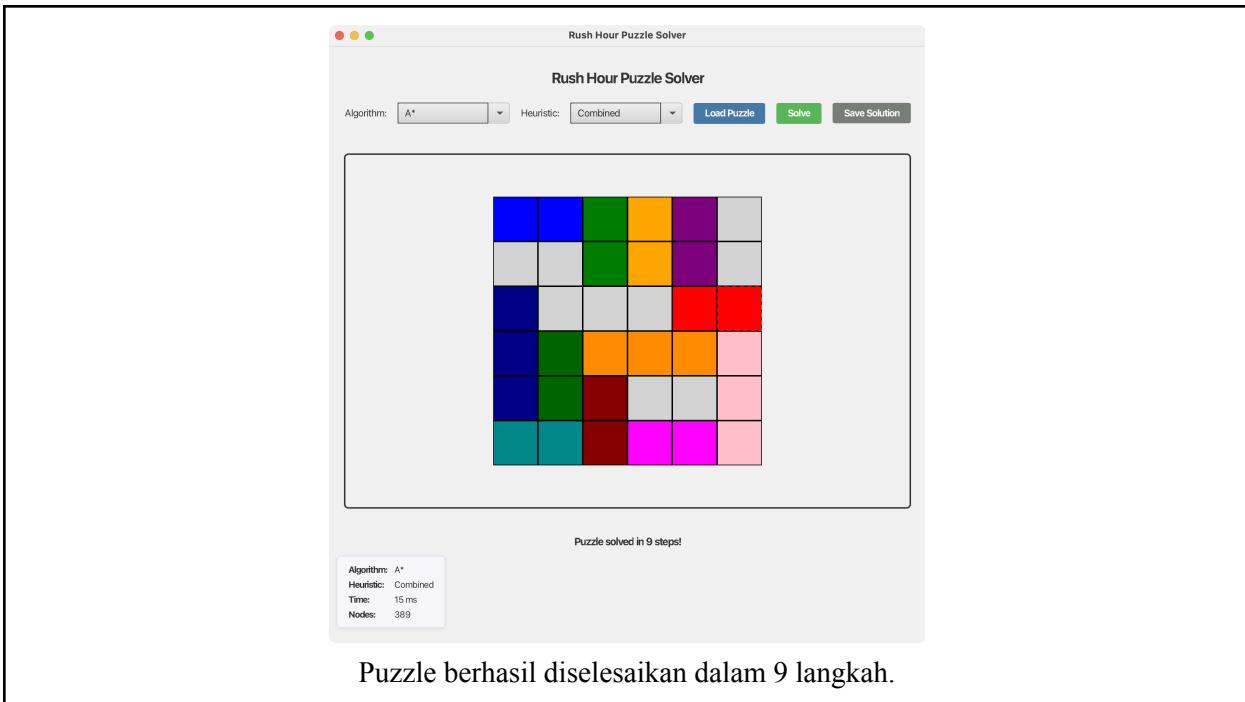
Puzzle berhasil diselesaikan dalam 9 langkah.

Heuristik : *Blocking Vehicles*



Puzzle berhasil diselesaikan dalam 9 langkah.

Heuristik : *Combined*

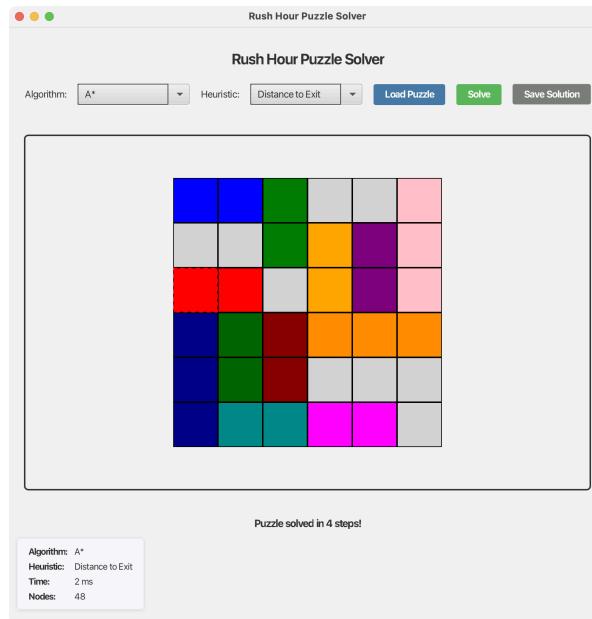


Puzzle berhasil diselesaikan dalam 9 langkah.

Kasus 2 : Kasus penyelesaian normal, *exit* di kiri.

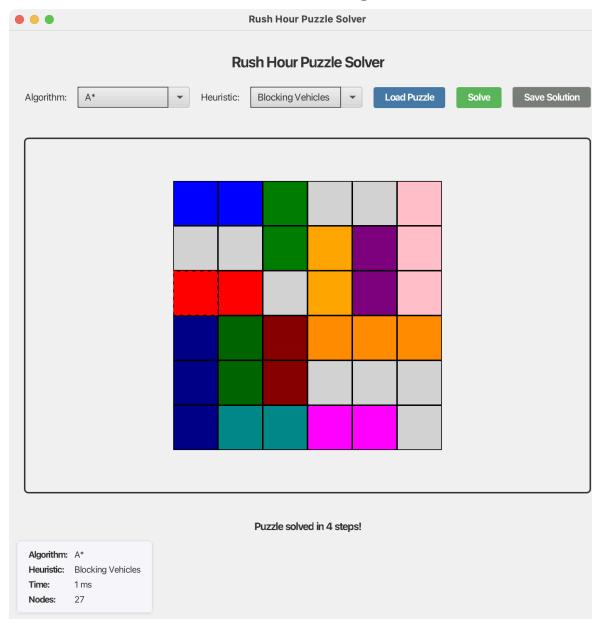
Kasus	
Dalam file .txt	GUI
<pre>6 6 11 AAB..F ..BCDF KGPPCDF GH.III GHJ... LLJMM.</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. The title bar says 'Rush Hour Puzzle Solver'. The algorithm dropdown is set to 'A*' and the heuristic dropdown is set to 'Distance to Exit'. There are buttons for 'Load Puzzle', 'Solve', and 'Save Solution'. Below the controls is a 6x6 grid representing the puzzle board. The grid contains various colored cars (blue, green, yellow, purple, red, pink, cyan, magenta) and empty slots (grey). A message at the bottom states 'Puzzle loaded: test.txt'. Below this message, a statistics box displays: Algorithm: Greedy Best-First, Heuristic: Combined, Time: 2 ms, and Nodes: 23.</p>
Solusi	

Heuristik : Distance to Exit



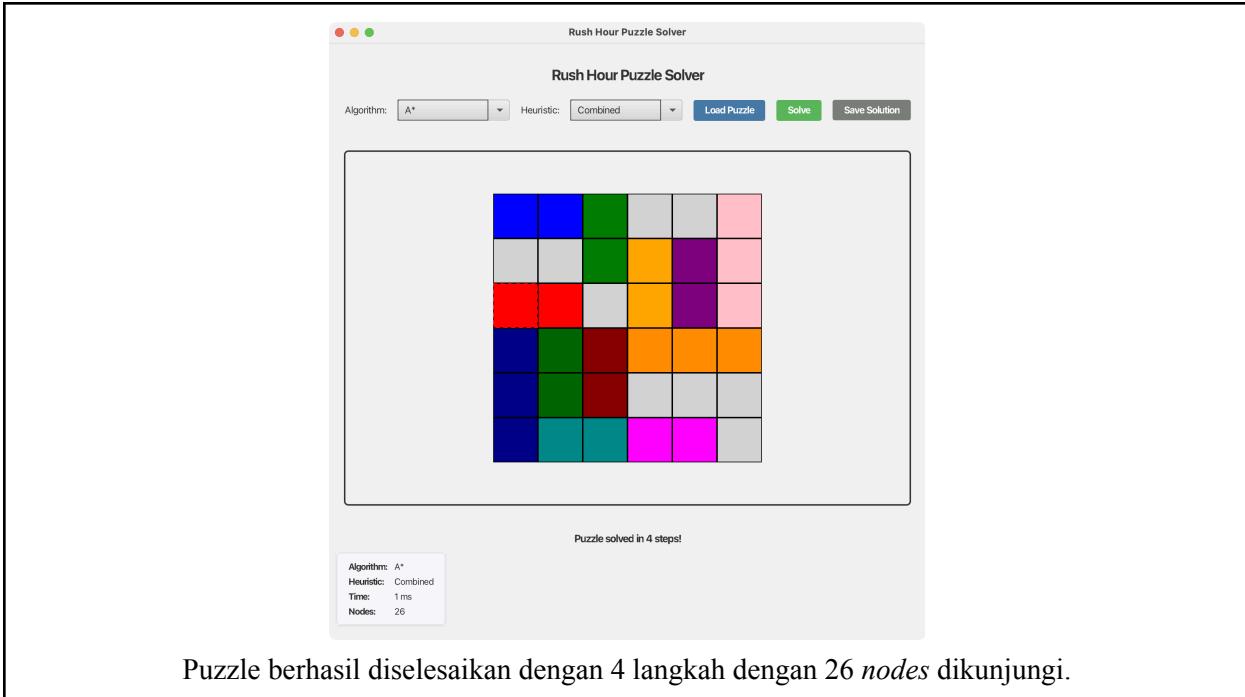
Puzzle berhasil diselesaikan dalam 4 langkah dengan 48 *nodes* dikunjungi.

Heuristik : Blocking Vehicles



Puzzle berhasil diselesaikan dalam 4 langkah dengan 27 *nodes* dikunjungi.

Heuristik : Combined

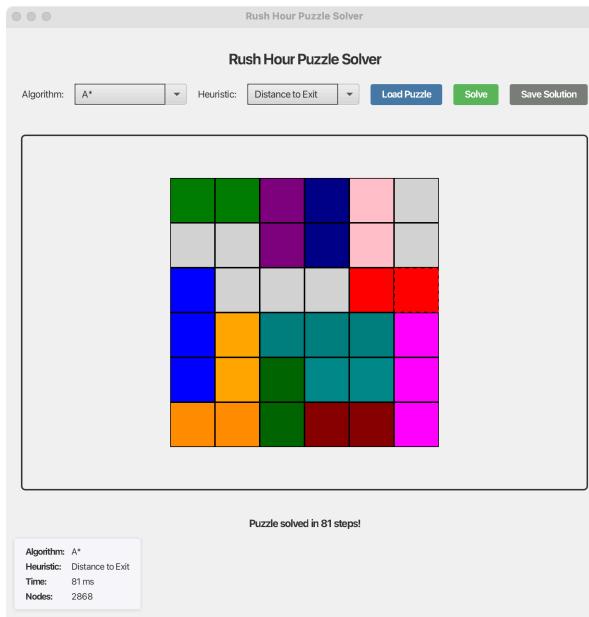


Puzzle berhasil diselesaikan dengan 4 langkah dengan 26 *nodes* dikunjungi.

Kasus 3 : Kasus konfigurasi 6×6 tersulit menurut Michael Fogelman.

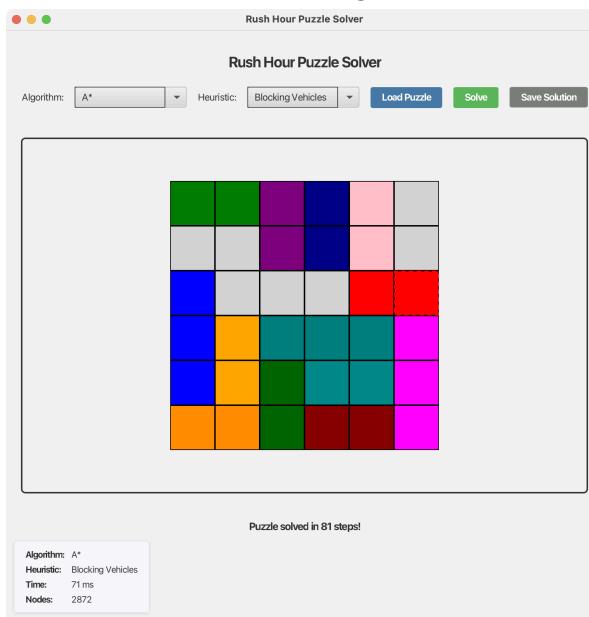
Kasus	
Dalam file .txt	GUI
<pre>6 6 12 ABB.F. ACD.FM ACDPPMK EEEG.M ..HGLL IIHJJ.</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. It is identical to the one above, but it has loaded a puzzle from a file named 'test.txt'. The status bar at the bottom now shows: Algorithm: Greedy Best-First, Heuristic: Combined, Time: 63 ms, Nodes: 2079. The message 'Puzzle loaded: test.txt' is displayed at the bottom of the grid area.</p>
Solusi	

Heuristik : *Distance to Exit*



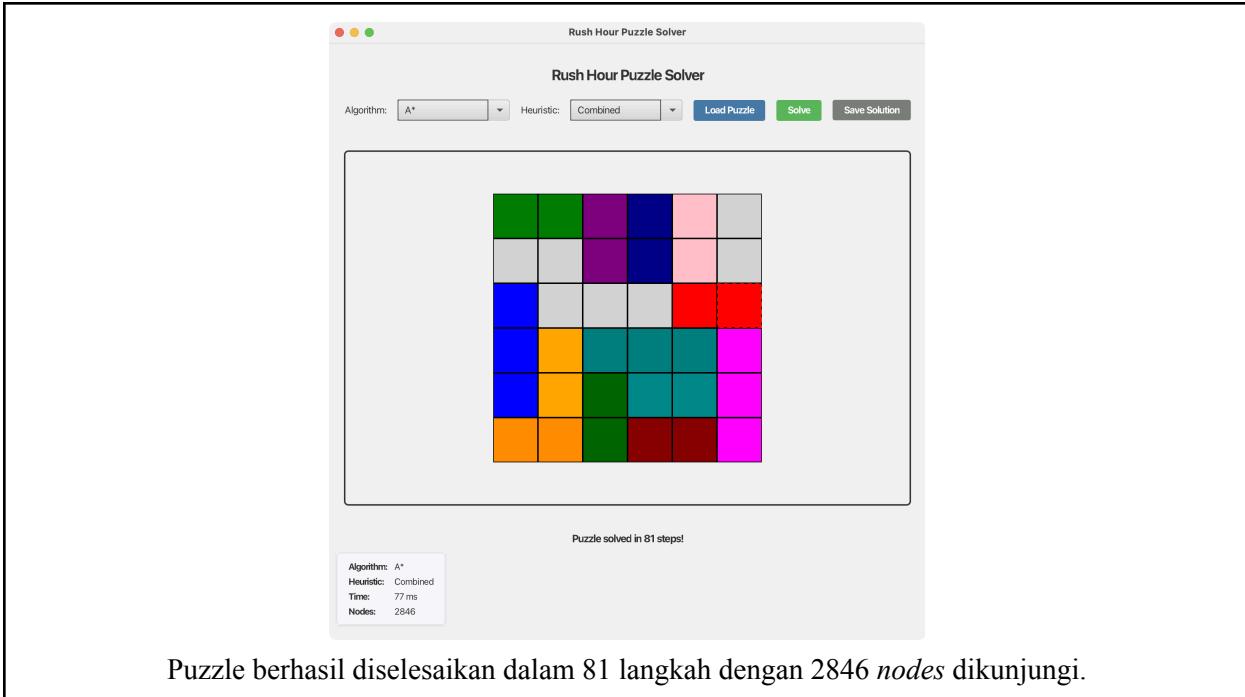
Puzzle berhasil diselesaikan dalam 81 langkah dengan 2868 *nodes* dikunjungi.

Heuristik : *Blocking Vehicles*



Puzzle berhasil diselesaikan dalam 81 langkah dengan 2868 *nodes* dikunjungi.

Heuristik : *Combined*

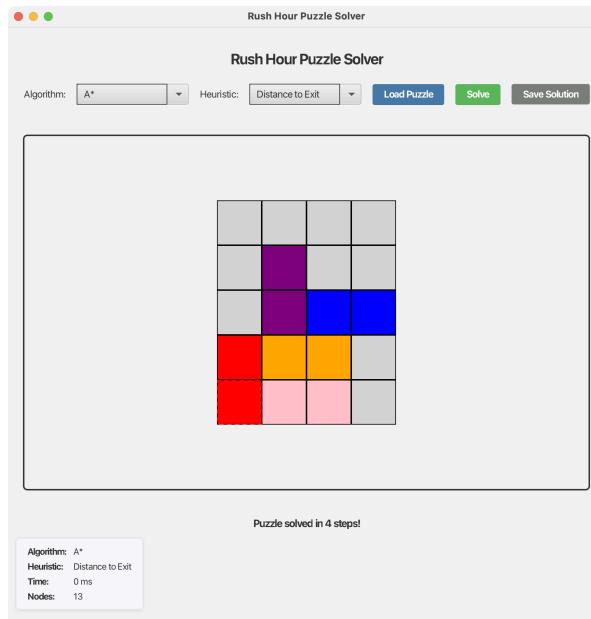


Puzzle berhasil diselesaikan dalam 81 langkah dengan 2846 *nodes* dikunjungi.

Kasus 4 : Kasus *primary piece* tidak horizontal dengan konfigurasi selain 6×6 .

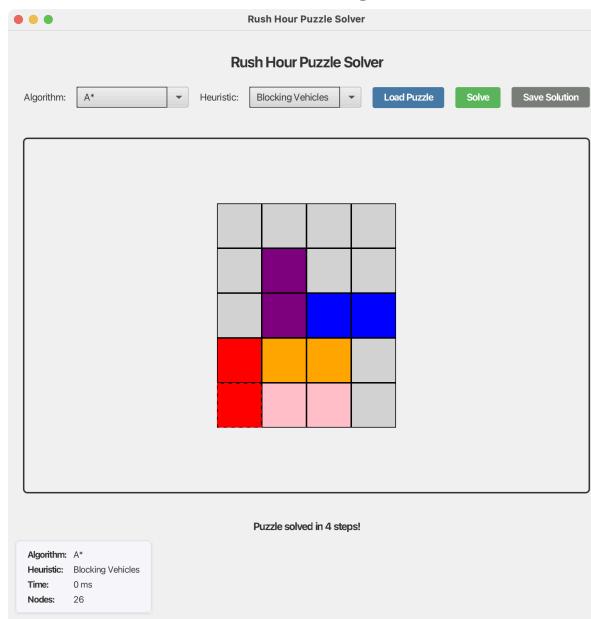
Kasus	
Dalam file .txt	GUI
<pre>5 4 4 PD.. PDAA CC.. FF.. K</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. The title bar says 'Rush Hour Puzzle Solver'. The main area is a 6x6 grid with colored blocks. Unlike the standard 6x6 puzzle, this one has a primary piece (a 2x3 rectangle) oriented vertically in the center. The puzzle is labeled 'test.txt' in the title bar. A message at the bottom says 'Puzzle loaded: test.txt'. Below the grid, a statistics box displays: Algorithm: A*, Heuristic: Distance to Exit, Time: 1 ms, and Nodes: 13.</p>
Solusi	

Heuristik : Distance to Exit



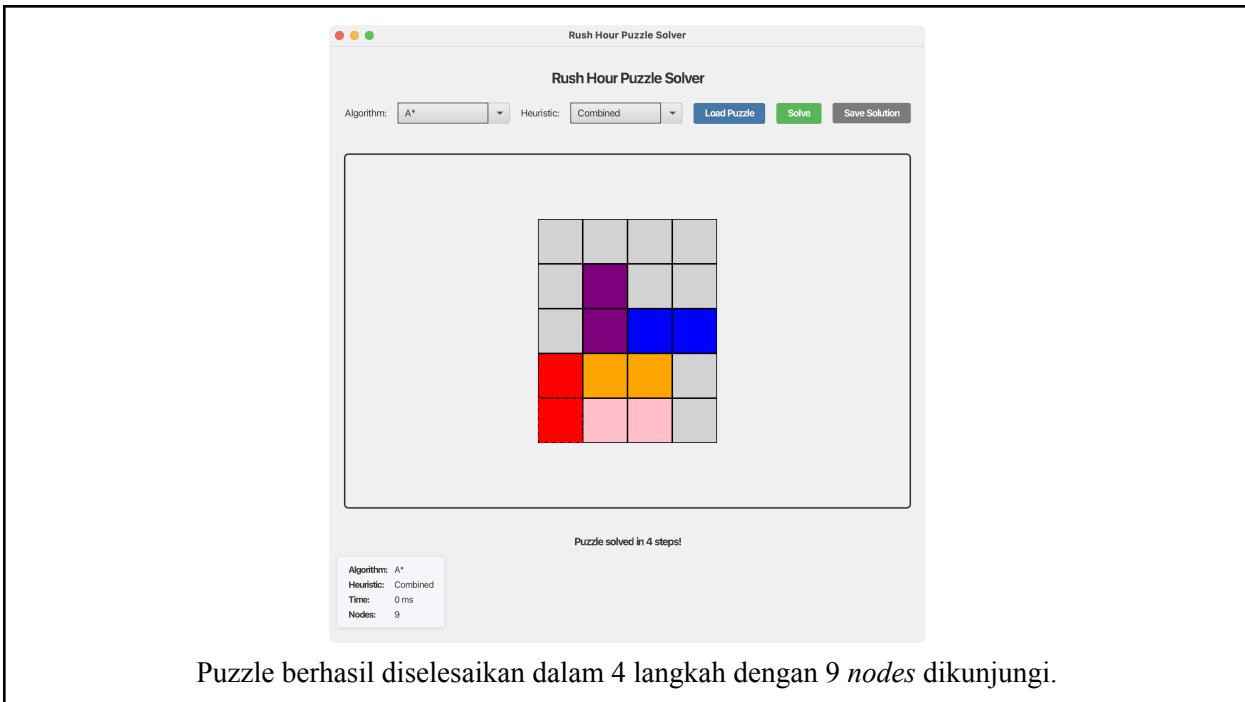
Puzzle berhasil diselesaikan dalam 4 langkah dengan 13 *nodes* dikunjungi.

Heuristik : Blocking Vehicles



Puzzle berhasil diselesaikan dalam 4 langkah dengan 13 *nodes* dikunjungi.

Heuristik : Combined

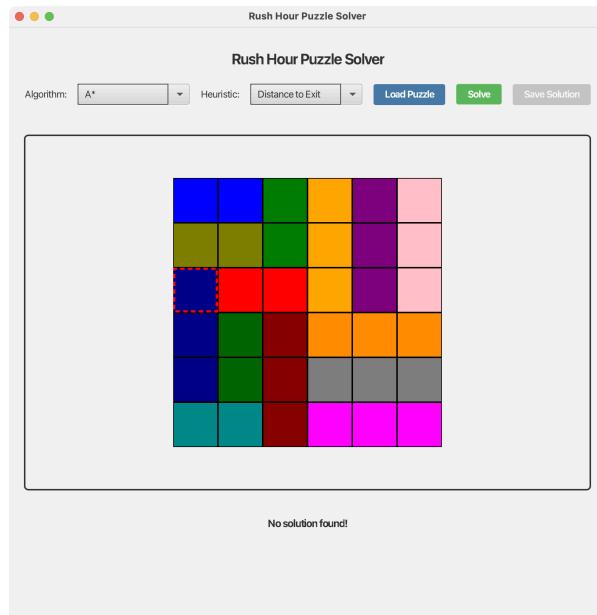


Puzzle berhasil diselesaikan dalam 4 langkah dengan 9 nodes dikunjungi.

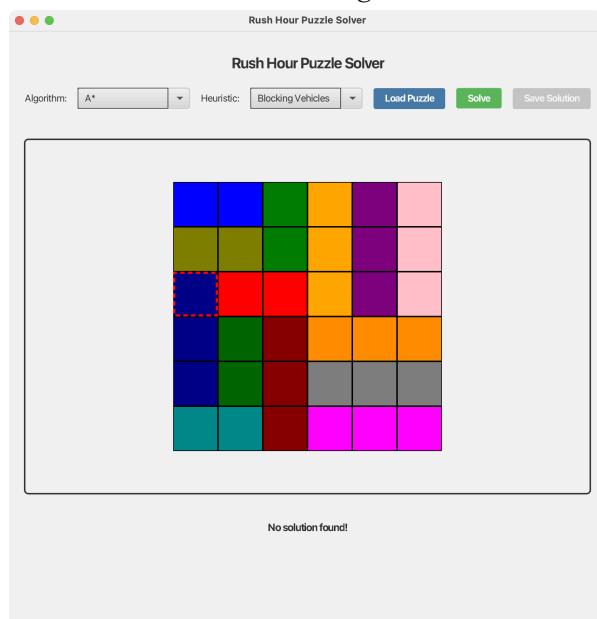
Kasus 5 : Kasus *unsolvable*.

Kasus	
Dalam file .txt	GUI
<pre>6 6 13 AABCDF 00BCDF KGPPCDF GHJIII GHJR LLJMM</pre>	<p>The screenshot shows the Rush Hour Puzzle Solver application window. The title bar says 'Rush Hour Puzzle Solver'. The algorithm dropdown is set to 'UCS'. The main area shows a 6x6 grid with various colored cars. The cars include blue, green, yellow, purple, red, and pink. Some cars are partially visible or overlapping. A blue border surrounds the central 4x4 area. Below the grid, a message says 'Puzzle loaded: test.txt'.</p>
Solusi	

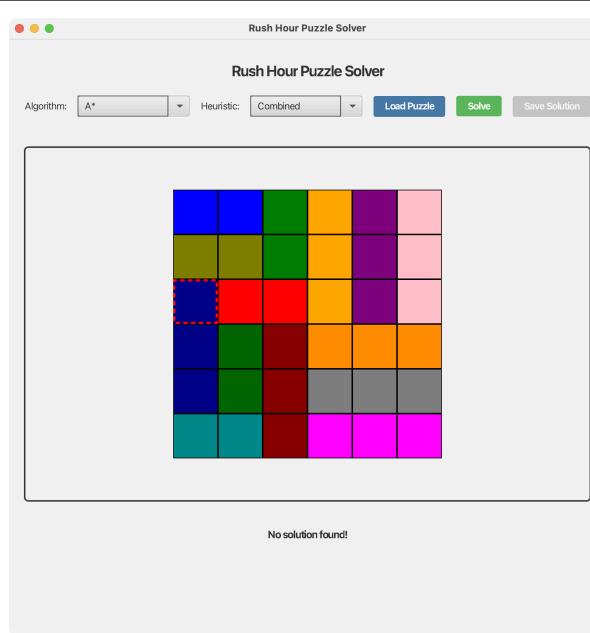
Heuristik : *Distance to Exit*



Heuristik : *Blocking Vehicles*



Heuristik : *Combined*



Ketika heuristik menghasilkan hasil yang sama, yaitu tidak ditemukan solusi. Hasil ini sesuai ekspektasi karena tidak ada ruang gerak untuk *piece* manapun.

4.3 Analisis

Dalam konteks algoritma pencarian jalur seperti UCS, Greedy Best First Search, dan A*, fungsi $g(n)$ merepresentasikan total biaya akumulatif dari simpul awal hingga simpul n . Pada permainan Rush Hour, $g(n)$ diartikan sebagai total jumlah gerakan yang telah dilakukan oleh semua kendaraan untuk mencapai konfigurasi papan tertentu dari kondisi awal. Sementara itu, fungsi $f(n)$ adalah fungsi evaluasi yang digunakan untuk memandu proses pencarian. Dalam UCS, $f(n) = g(n)$, sedangkan dalam A*, $f(n) = g(n) + h(n)$, yang berarti kombinasi antara biaya sebenarnya dari awal dan estimasi biaya menuju tujuan.

4.3.1 Algoritma UCS

Dalam konteks permainan Rush Hour, UCS akan berperilaku sama seperti BFS. Hal ini terjadi karena setiap gerakan kendaraan dianggap memiliki biaya yang sama, yaitu satu langkah tiap gerakan satu sel. Dengan $g(n)$ yang selalu bertambah sebesar satu untuk setiap langkah, dan tidak adanya perbedaan bobot antar aksi, maka urutan eksplorasi simpul serta path yang dihasilkan oleh UCS akan identik dengan BFS. Algoritma UCS dalam permainan Rush Hour akan selalu menemukan langkah optimal untuk mengeluarkan Primary Piece, meskipun dengan penelusuran node yang relatif lebih banyak dibandingkan algoritma BGFS dan A*.

Untuk menghitung kompleksitas algoritma UCS, digunakan pendekatan sebagai berikut. Misalkan bahwa b merupakan *branching factor* maksimum (jumlah maksimal gerakan yang dapat dilakukan dari satu node), d merupakan kedalaman dari solusi, dan n jumlah piece.

1. Pemanggilan *generatePath* menghasilkan sebanyak $2n - 1$ kemungkinan pergerakan (-1 karena gerakan tidak boleh kembali ke *parent*), sehingga $b = 2n - 1$.
2. Untuk iterasi setiap node, akan dipanggil kembali *generatePath*, yang akan menghasilkan $2n - 1$ kemungkinan lain hingga ke *depth* paling dalam, sehingga $d = 2n - 1$.
3. Dengan ini, jumlah iterasi yang diperlukan akan $\approx (2n - 1)^{(2n-1)}$ atau $O(b^d)$.

4.3.2 Algoritma BGFS

Dalam konteks permainan Rush Hour, $h(n)$ didefinisikan sebagai fungsi estimasi kedekatan suatu konfigurasi papan (BoardState) terhadap kondisi solusi, yaitu ketika kendaraan utama berhasil mencapai pintu keluar. Beberapa pendekatan heuristic yang digunakan antara lain: Distance to Exit (jarak sel ujung kendaraan utama ke posisi exit), Total Cars Blocking (jumlah kendaraan yang menghalangi jalur utama), serta Combined, yaitu kombinasi dari keduanya. Pemilihan heuristic ini memungkinkan GBFS untuk lebih cepat mengarahkan pencarian pada konfigurasi papan yang tampak lebih menjanjikan untuk menuju solusi, walaupun tanpa jaminan bahwa jalur tersebut adalah jalur optimal.

Meskipun GBFS unggul dalam kecepatan dan efisiensi ruang pencarian pada banyak kasus, algoritma ini memiliki keterbatasan penting, yaitu tidak menjamin pencapaian solusi optimal. GBFS hanya mempertimbangkan nilai heuristic ($h(n)$) tanpa memperhitungkan langkah yang telah dilakukan ($g(n)$), sehingga berpotensi terjebak dalam solusi local optimum. Dalam konteks Rush Hour, GBFS bisa saja memilih langkah yang terlihat menjanjikan (misalnya, memajukan kendaraan yang menghalangi jalan utama), namun ternyata justru menutup jalur dan memaksa langkah-langkah mundur yang tidak efisien. Oleh karena itu, meskipun GBFS sering cepat dalam menemukan solusi, solusi yang dihasilkan belum tentu merupakan solusi terbaik.

Untuk menghitung kompleksitas algoritma GBFS, digunakan pendekatan sebagai berikut. Misalkan bahwa b merupakan *branching factor* maksimum (jumlah maksimal gerakan yang dapat dilakukan dari satu node), d merupakan kedalaman dari solusi, dan n jumlah piece.

4. Pemanggilan *generatePath* menghasilkan sebanyak $2n - 1$ kemungkinan pergerakan (-1 karena gerakan tidak boleh kembali ke *parent*), sehingga $b = 2n - 1$.
5. Untuk iterasi setiap node, akan dipanggil kembali *generatePath*, yang akan menghasilkan $2n - 1$ kemungkinan lain hingga ke *depth* paling dalam, sehingga $d = 2n - 1$.
6. Dengan ini, jumlah iterasi yang diperlukan akan $\approx (2n - 1)^{(2n-1)}$ atau $O(b^d)$.

4.3.3 Algoritma A*

Dalam konteks permainan Rush Hour, fungsi $g(n)$ direpresentasikan sebagai jumlah total langkah (gerakan) yang telah dilakukan oleh semua kendaraan sejak konfigurasi awal hingga mencapai simpul n . Sementara itu, $h(n)$ dihitung berdasarkan estimasi pendekatan terhadap solusi, yang dalam implementasi ini mencakup beberapa opsi heuristic seperti Distance to Exit, Total Cars Blocking, maupun Combined. Dengan $f(n)$ sebagai penjumlahan dari kedua komponen tersebut, algoritma A* akan mengarahkan pencarian secara selektif: memperhatikan efisiensi langkah yang telah diambil ($g(n)$) sekaligus mengevaluasi seberapa menjanjikan suatu konfigurasi menuju solusi ($h(n)$).

A* memiliki keunggulan dibandingkan algoritma seperti UCS atau GBFS karena mampu memperhitungkan biaya historis dan estimasi masa depan secara bersamaan. Secara teoritis, algoritma A* akan lebih efisien dibandingkan UCS dalam menyelesaikan permainan Rush Hour. Hal ini dikarenakan meskipun A* tetap mempertimbangkan total langkah ($g(n)$), ia juga menggunakan heuristic ($h(n)$) yang memberikan keunggulan dalam memprioritaskan perluasan simpul yang lebih dekat ke tujuan. Dengan kata lain, A* dapat menghindari eksplorasi terhadap simpul yang tidak menjanjikan, sehingga ruang pencarian menjadi lebih sempit dan jumlah node yang dikunjungi lebih sedikit. Keunggulan ini terlihat pada penghematan jumlah node yang di-expand selama proses pencarian berlangsung.

Sebuah heuristic $h(n)$ dikatakan admissible jika untuk setiap simpul n , berlaku $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah biaya sebenarnya minimum dari simpul n menuju simpul tujuan. Heuristic dikatakan admissible apabila ia tidak pernah melebih-lebihkan estimasi biaya ke tujuan (bersifat optimistik). Dalam konteks permainan Rush Hour, heuristic yang digunakan dalam program — yaitu *Distance to Exit*, *Total Cars Blocking*, dan *Combined* — tergolong admissible karena tidak pernah menghasilkan estimasi biaya lebih tinggi dari sebenarnya. Bahkan dalam banyak kasus, nilai $h(n)$ cenderung rendah dan tidak dominan dalam menentukan eksplorasi node, terutama pada papan dengan konfigurasi sederhana. Oleh karena itu, ketiga heuristic tersebut dapat dikategorikan sebagai heuristic admissible.

Untuk menghitung kompleksitas algoritma GBFS, digunakan pendekatan sebagai berikut. Misalkan bahwa b merupakan *branching factor* maksimum (jumlah maksimal gerakan yang dapat dilakukan dari satu node), d merupakan kedalaman dari solusi, dan n jumlah piece.

7. Pemanggilan *generatePath* menghasilkan sebanyak $2n - 1$ kemungkinan pergerakan (-1 karena gerakan tidak boleh kembali ke *parent*), sehingga $b = 2n - 1$.
8. Untuk iterasi setiap node, akan dipanggil kembali *generatePath*, yang akan menghasilkan $2n - 1$ kemungkinan lain hingga ke *depth* paling dalam, sehingga $d = 2n - 1$.
9. Dengan ini, jumlah iterasi yang diperlukan akan $\approx (2n - 1)^{(2n-1)}$ atau $O(b^d)$.

5 Lampiran

5.1 Pranala Github

https://github.com/MaheswaraKaindra/Tucil3_13523015_13523039

6 Kesimpulan

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.	✓	

7 Referensi

Maulidevi, N. U. (2025). *Penentuan rute (route/path planning): Bagian 1: BFS, DFS, UCS, Greedy Best First Search* [Bahan kuliah]. Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung.
[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)

Maulidevi, N. U., Munir, R. (2025). *Penentuan rute (route/path planning): Bagian 2: Algoritma A** [Bahan Kuliah]. Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung.
[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)

Fogleman, M. (n.d.). *Rush Hour solver*. Diakses 19 Mei , 2025, dari
<https://www.michaelfogleman.com/rush/>