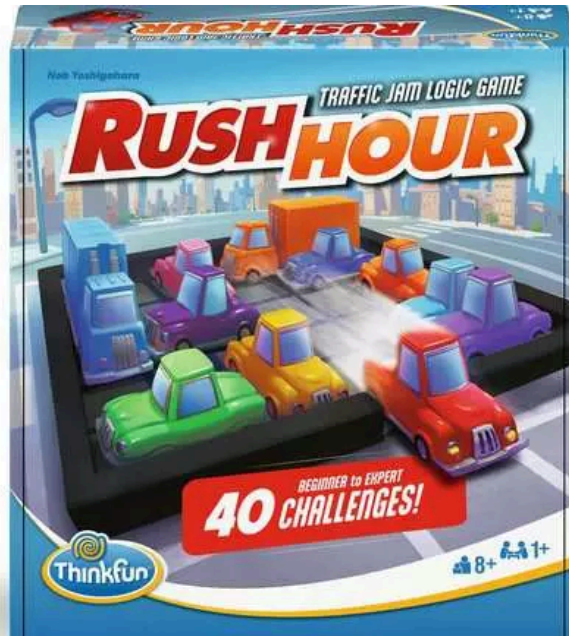


# **LAPORAN TUGAS KECIL 3**

IF2211 Strategi Algoritma

## **Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding**



**Disusun oleh:**

Muh. Rusmin Nurwadin (13523068)

Reza Ahmad Syarif (13523119)

**Program Studi Teknik Informatika**

**Sekolah Teknik Elektro dan Informatika**

**Institut Teknologi Bandung**

**2025**

## A. Pendahuluan

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.

## B. Algoritma Pencarian Jalur

### 1. Uniform Cost Search

Uniform Cost Search (UCS) adalah algoritma pencarian yang bertujuan untuk menemukan jalur dengan biaya (*cost*) terendah dari *state* awal ke *state* tujuan. UCS merupakan varian dari algoritma *Breadth-First Search* yang memperhitungkan biaya dari setiap langkah yang diambil. Pada kasus Puzzle Rush Hour ini, biaya dari setiap langkah yang diambil adalah tetap, maka UCS dan BFS akan menghasilkan jalur yang sama dan mengunjungi *node* yang sama dalam urutan yang sama pula. Algoritma UCS menggunakan struktur data Priority Queue untuk mengurutkan *node* berdasarkan biaya kumulatif ( $g(n)$ ), yaitu total biaya yang dikeluarkan untuk mencapai *node* tersebut. *Node* dengan biaya terendah akan dieksplorasi lebih dahulu. Algoritma UCS bersifat *Uninformed Search* atau *Blind Search* karena tidak menggunakan heuristik apapun. Algoritma ini hanya berfokus pada biaya yang telah ditempuh sehingga memastikan bahwa solusi yang ditemukan adalah solusi optimal.

Langkah Algoritma:

- 1) Inisialisasi antrian prioritas yang berisi *node* awal dengan biaya 0.
- 2) Ambil *node* dengan biaya terendah dari antrian yang telah diurutkan berdasarkan biaya total dari *state* awal ke *node* tersebut.
- 3) Jika *node* tersebut adalah *goal node*, kembalikan jalur menuju solusi tersebut.
- 4) Jika bukan, *generate* semua tetangga dari *node* saat ini dan periksa apakah jalur yang baru lebih murah dibandingkan jalur sebelumnya.
- 5) Jika jalur lebih murah, perbarui *cost* dan tambahkan tetangga ke antrian.
- 6) Ulangi langkah algoritma hingga *goal node* ditemukan atau antrian kosong yang berarti tidak ada solusi.

### 2. Greedy Best First Search

Greedy Best First Search (GBFS) adalah algoritma pencarian dengan memilih *node* yang dieksplorasi berdasarkan heuristik atau perkiraan jarak ke *goal node*. Algoritma GBFS tidak mempertimbangkan biaya yang dibayar untuk mencapai *node* saat ini ( $g(n)$ ), tetapi hanya fokus pada kecepatan menuju *goal node* dan heuristik ( $h(n)$ ). Algoritma GBFS menggunakan struktur data Priority Queue untuk mengurutkan *node* berdasarkan heuristik ( $h(n)$ ), yaitu perkiraan biaya dari *node* saat ini menuju *goal node*. *Node* dengan heuristik terkecil dipilih untuk dieksplorasi terlebih dahulu. Secara teoritis, algoritma GBFS tidak menjamin akan mendapatkan solusi optimal karena hanya mengandalkan heuristik untuk memilih *node* yang akan dieksplorasi. Jika heuristik yang dipilih tidak akurat atau jika ada beberapa jalur yang terlihat lebih pendek tetapi sebenarnya lebih mahal, GBFS bisa saja terjebak pada jalur yang tidak optimal.

Langkah Algoritma:

- 1) Inisialisasi antrian prioritas yang berisi *node* awal dengan nilai heuristik  $h(n)$ .
- 2) Ambil *node* dengan nilai heuristik terkecil dari antrian.
- 3) Jika *node* yang diambil adalah *goal node*, kembalikan jalur menuju solusi tersebut.

- 4) Jika bukan, *generate* semua tetangga dan tentukan nilai heuristik untuk setiap tetangga.
- 5) Tambahkan tetangga ke antrian dengan prioritas berdasarkan heuristik.
- 6) Ulangi langkah algoritma hingga *goal node* ditemukan atau antrian kosong yang berarti tidak ada solusi.

### 3. A\*

A\* adalah algoritma pencarian jalur yang digunakan untuk menemukan rute terpendek atau biaya minimum dari titik awal ke tujuan dalam sebuah graf atau peta, dengan memanfaatkan pendekatan heuristik. Ide untuk algoritma ini adalah menghindari jalur yang sudah “mahal” sebelumnya. Fungsi evaluasi pada algoritma ini didefinisikan dengan  $f(n) = g(n) + h(n)$ , dimana  $f(n)$  adalah perkiraan total biaya jalur melalui  $n$  ke tujuan,  $g(n)$  adalah biaya sejauh ini untuk mencapai  $n$ , dan  $h(n)$  adalah perkiraan biaya dari  $n$  ke tujuan atau nilai heuristik. Heuristik  $h(n)$  dalam algoritma A\* dikatakan *admissible* jika nilai estimasinya tidak pernah melebihi biaya sebenarnya dari simpul  $n$  ke tujuan, dengan kata lain  $h(n)$  kurang dari  $h^*(n)$ , dimana  $h^*(n)$  adalah jarak sebenarnya dari  $n$  ke tujuan.

Langkah Algoritma :

- 1) Inisialisasi antrian prioritas dengan *node* awal
- 2) Hitung nilai  $f(n) = g(n) + h(n)$  untuk *node* awal
- 3) Selama antrian tidak kosong, maka :
  - a. Ambil *node* dengan nilai  $f(n)$  terkecil.
  - b. Jika *node* adalah tujuan, kembalikan jalur solusi.
  - c. Generate semua tetangga dari *node* saat ini.
  - d. Hitung  $f(n)$  untuk tiap tetangga.
  - e. Jika jalur lebih baik, simpan dan tambahkan ke antrian.
  - f. Ulangi langkah hingga tujuan atau antrian kosong.
- 4) Jika antrian kosong, dan tujuan belum ditemukan, artinya tidak ada solusi.

## C. Source Code

1. Algoritma
  - a. Uniform Cost Search

Source Code	Keterangan
<pre>package backend.algorithm;  import java.util.*; import backend.model.Board; import backend.util.Heuristic;  public class UCS implements PathfindingAlgorithm {     private int nodesVisited;     private long executionTime;      @Override     public List&lt;Board&gt; solve(Board initialBoard, Heuristic     heuristic) {         nodesVisited = 0;         long startTime = System.currentTimeMillis();          Queue&lt;UCSNode&gt; queue = new LinkedList&lt;&gt;();         Set&lt;Board&gt; visited = new HashSet&lt;&gt;();          queue.add(new UCSNode(initialBoard, null, 0));         visited.add(initialBoard);          while (!queue.isEmpty()) {             UCSNode currentNode = queue.poll();             nodesVisited++;              if (currentNode.getBoard().isGoal()) {                 executionTime = System.currentTimeMillis() -                 startTime;                 return reconstructPath(currentNode);             }              for (Board neighbor :             currentNode.getBoard().generateNeighbors(currentNode.getBoar             d().getZobristTable())) {                 if (!visited.contains(neighbor)) {                     visited.add(neighbor);                     queue.add(new UCSNode(neighbor,                     currentNode, currentNode.getCost() + 1));                 }             }         }          executionTime = System.currentTimeMillis() -         startTime;     } }</pre>	<p>Fungsi: solve(Board initialBoard): Fungsi utama yang memulai pencarian untuk menemukan solusi.</p> <p>reconstructPath(UCSNode node): Fungsi: Membentuk jalur dari solusi yang ditemukan, mulai dari goal hingga start, dengan melacak parent node.</p>

```

        return Collections.emptyList();
    }

    private static class UCSNode implements SearchNode {
        private final Board board;
        private final SearchNode parent;
        private final int cost;

        public UCSNode(Board board, SearchNode parent, int
cost) {
            this.board = board;
            this.parent = parent;
            this.cost = cost;
        }

        @Override
        public Board getBoard() {
            return board;
        }

        @Override
        public SearchNode getParent() {
            return parent;
        }

        @Override
        public int getCost() {
            return cost;
        }

        @Override
        public int getHeuristic() {
            return 0;
        }

        @Override
        public int getPriority() {
            return cost;
        }
    }

    private List<Board> reconstructPath(UCSNode node) {
        LinkedList<Board> path = new LinkedList<>();
        while (node != null) {
            path.add(node.getBoard());
            node = (UCSNode) node.getParent();
        }
        Collections.reverse(path);
        return path;
    }

    @Override
    public String getName() {

```

<pre>         return "Uniform Cost Search (UCS)";     }      @Override     public int getNodesVisited() {         return nodesVisited;     }      @Override     public long getExecutionTime() {         return executionTime;     }      @Override     public String getHeuristicName() {         return "None";     } } </pre>	
--	--

#### b. Greedy Best First Search

Source Code	Keterangan
<pre> package backend.algorithm;  import java.util.*; import backend.model.Board; import backend.util.Heuristic;  public class GBFS implements PathfindingAlgorithm {     private int nodesVisited;     private long executionTime;     private Heuristic heuristic;      public GBFS(Heuristic heuristic) {         this.heuristic = heuristic;     }      @Override     public List&lt;Board&gt; solve(Board initialBoard, Heuristic     heuristic) {         nodesVisited = 0;         long startTime = System.currentTimeMillis();          PriorityQueue&lt;GBFSNode&gt; openSet = new         PriorityQueue&lt;&gt;(             Comparator.comparingInt(node -&gt;             node.heuristicValue));          Map&lt;Board, Integer&gt; bestHeuristics = new         HashMap&lt;&gt;();          openSet.add(new GBFSNode(initialBoard, null, </pre>	<p>solve(Board initialBoard, Heuristic heuristic): Fungsi utama untuk menyelesaikan puzzle menggunakan GBFS dengan memilih node berdasarkan heuristik.</p> <p>reconstructPath(GBFSNode node): Berfungsi untuk menyusun jalur dari solusi yang ditemukan, dengan melacak parent node dan mengembalikan path dari start hingga goal.</p>

```

    heuristic.estimate(initialBoard), 0));
    bestHeuristics.put(initialBoard,
    heuristic.estimate(initialBoard));

    while (!openSet.isEmpty()) {
        GBFSNode currentNode = openSet.poll();
        nodesVisited++;

        if (bestHeuristics.get(currentNode.getBoard()) <
currentNode.heuristicValue) {
            continue;
        }

        if (currentNode.getBoard().isGoal()) {
            executionTime = System.currentTimeMillis() -
startTime;
            return reconstructPath(currentNode);
        }

        for (Board neighbor :
currentNode.getBoard().generateNeighbors(
currentNode.getBoard().getZobristTable())) {

            int newHeuristic =
heuristic.estimate(neighbor);

            if (!bestHeuristics.containsKey(neighbor) ||
newHeuristic <
bestHeuristics.get(neighbor)) {

                bestHeuristics.put(neighbor,
newHeuristic);
                openSet.add(new GBFSNode(neighbor,
currentNode, newHeuristic, currentNode.cost + 1));
            }
        }

        executionTime = System.currentTimeMillis() -
startTime;
        return Collections.emptyList();
    }

    private List<Board> reconstructPath(GBFSNode node) {
        List<Board> path = new ArrayList<>();
        while (node != null) {
            path.add(node.getBoard());
            node = (GBFSNode) node.getParent();
        }
        Collections.reverse(path);
        return path;
    }

```



```

@Override
public String getName() {
    return "Greedy Best-First Search (GBFS)";
}

@Override
public int getNodesVisited() {
    return nodesVisited;
}

@Override
public long getExecutionTime() {
    return executionTime;
}

@Override
public String getHeuristicName() {
    return heuristic.getName();
}

private static class GBFSNode implements SearchNode {
    private final Board board;
    private final SearchNode parent;
    int heuristicValue;
    int cost;

    public GBFSNode(Board board, SearchNode parent, int
heuristicValue, int cost) {
        this.board = board;
        this.parent = parent;
        this.heuristicValue = heuristicValue;
        this.cost = cost;
    }

    @Override
    public Board getBoard() {
        return board;
    }

    @Override
    public SearchNode getParent() {
        return parent;
    }

    @Override
    public int getHeuristic() {
        return heuristicValue;
    }

    @Override
    public int getCost() {
        return cost;
    }
}

```

<pre>     }      @Override     public int getPriority() {         return heuristicValue;     } } </pre>	
---	--

### c. A\*

Source Code	Keterangan
<pre> package backend.algorithm;  import backend.model.Board; import backend.util.Heuristic;  import java.util.*;  public class AStar implements PathfindingAlgorithm {     private Heuristic heuristic; // Heuristik yang digunakan     private int visitedNodes = 0; // Jumlah node yang     dikunjungi     private long execTime = 0; // Waktu eksekusi algoritma     (ms)      public AStar(Heuristic heuristic) {         this.heuristic = heuristic;     }      @Override     public List&lt;Board&gt; solve(Board initBoard, Heuristic     heuristic) {         // 1. Inisialisasi waktu mulai untuk pengukuran         performa         long startTime = System.currentTimeMillis();         visitedNodes = 0;          // 2. Priority queue untuk open set, berdasarkan         nilai f(n) = g(n) + h(n)         PriorityQueue&lt;ANode&gt; openSet = new PriorityQueue&lt;&gt; (             Comparator.comparingInt(node -&gt; node.f));          // 3. Simpan state yang sudah dieksplorasi (closed         set)         Set&lt;Board&gt; closedSet = new HashSet&lt;&gt;();          // 4. Mapping untuk rekonstruksi jalur         Map&lt;Board, Board&gt; prevBoard = new HashMap&lt;&gt;();          // 5. Mapping untuk g score (biaya dari start ke         node) </pre>	<p>solve(Board initialBoard, Heuristic heuristic): Fungsi utama untuk menyelesaikan puzzle menggunakan A*, yang mempertimbangkan g(n) dan h(n) dalam perhitungan prioritas node.</p> <p>buildPath(Map&lt;Board, Board&gt; prevBoard, Board current): Berfungsi untuk merekonstruksi jalur solusi dari node goal dengan melacak parent node dari goal ke start.</p>

```

        Map<Board, Integer> gScore = new HashMap<>();

        // 6. Inisialisasi node awal dengan g=0 dan
        h=heuristik
        ANode startNode = new ANode(initBoard, 0,
        heuristic.estimate(initBoard));
        openSet.add(startNode);
        gScore.put(initBoard, 0);

        // 7. Loop utama algoritma A*
        while (!openSet.isEmpty()) {
            // 8. Ambil node dengan f(n) terkecil
            ANode current = openSet.poll();
            visitedNodes++;

            // 9. Cek apakah sudah mencapai goal state
            if (isGoalState(current.board)) {
                execTime = System.currentTimeMillis() -
startTime;
                return buildPath(prevBoard, current.board);
            }

            // 10. Tambahkan ke closed set
            closedSet.add(current.board);

            // 11. Generate semua tetangga (neighbor) state
            List<Board> neighbors =
current.board.generateNeighbors(current.board.getZobristTabl
e());

            // 12. Proses setiap tetangga
            for (Board neighbor : neighbors) {
                // 13. Lewati jika sudah dieksplorasi
                if (closedSet.contains(neighbor)) {
                    continue;
                }

                // 14. Hitung g score sementara (biaya dari
start ke tetangga melalui current)
                int tentG = gScore.get(current.board) + 1; //
1 adalah cost untuk satu gerakan

                // 15. Jika ini node baru atau kita menemukan
jalur yang lebih baik
                if (!gScore.containsKey(neighbor) || tentG <
gScore.get(neighbor)) {
                    // 16. Perbarui jalur
                    prevBoard.put(neighbor, current.board);
                    gScore.put(neighbor, tentG);

                    // 17. Tambahkan ke open set dengan f = g
+ h
                    openSet.add(new ANode(neighbor, tentG,

```

```

        heuristic.estimate(neighbor));
    }
}

// 18. Tidak ada solusi yang ditemukan
execTime = System.currentTimeMillis() - startTime;
return new ArrayList<>();
}

private boolean isGoalState(Board board) {
    return board.isGoal();
}

private List<Board> buildPath(Map<Board, Board>
prevBoard, Board current) {
    // 34. Inisialisasi path dengan goal
    List<Board> path = new ArrayList<>();
    path.add(current);

    // 35. Rekonstruksi jalur dari goal ke start
    while (prevBoard.containsKey(current)) {
        current = prevBoard.get(current);
        path.add(0, current); // Tambahkan di awal list
    }

    return path;
}

@Override
public String getName() {
    return "A* Search";
}

@Override
public int getNodesVisited() {
    return visitedNodes;
}

@Override
public long getExecutionTime() {
    return execTime;
}

@Override
public String getHeuristicName() {
    return heuristic.getName();
}

private static class ANode implements SearchNode{
    Board board; // State board
    int g;        // Biaya dari start ke node ini
    int h;        // Perkiraan heuristik dari node ini ke

```

<pre> goal int f;          // f(n) = g(n) + h(n)  public ANode(Board board, int g, int h) {     this.board = board;     this.g = g;     this.h = h;     this.f = g + h; }  @Override public Board getBoard() {     return board; }  @Override public SearchNode getParent() {     return null; }  @Override public int getCost() {     return g; }  @Override public int getHeuristic() {     return h; }  @Override public int getPriority() {     return f; }  } </pre>	
---	--

## 2. Heuristik

Source Code	Keterangan
<pre> package backend.util;  import backend.model.Board;  public interface Heuristic {     int estimate(Board board);      String getName(); } </pre>	<p>Interface Heuristic digunakan untuk mengabstraksi fungsi heuristik sehingga algoritma pencarian dapat menggunakan berbagai macam jenis heuristik tanpa mengubah algoritma. Method estimate berfungsi untuk menghitung estimasi jarak dari papan saat ini hingga</p>

	<p>mencapai <i>goal</i> atau <i>exit</i>. Method <code>getName</code> berfungsi untuk mendapatkan nama dari heuristik yang digunakan.</p>
<pre> package backend.util;  import backend.model.Board; import backend.model.Car;  public class HeuristicBlocking implements Heuristic {     @Override     public int estimate(Board board) {         Car player = board.getCar('P');         return countBlocking(board, player); // Menghitung         jumlah mobil yang menghalangi     }      private int countBlocking(Board board, Car player) {         int count = 0;         int primRow = player.getRow();         int primEndCol = player.getCol() +         player.getLength() - 1;         int exitRow = board.getExitRow();         int exitCol = board.getExitCol();          // Jika mobil utama bergerak horizontal (exit         horizontal)         if (player.isHorizontal()) {             for (Car car : board.getCars()) {                 if (car.isPrimary()) continue;                  if (car.isVertical() &amp;&amp; car.getRow() &lt;=                 primRow &amp;&amp;                     car.getRow() + car.getLength() - 1                 &gt;= primRow) {                     if (exitCol &gt; primEndCol &amp;&amp; car.getCol()                 &gt; primEndCol &amp;&amp; car.getCol() &lt;= exitCol) {                         count++;                     } else if (exitCol &lt; player.getCol() &amp;&amp;                 car.getCol() &lt; player.getCol() &amp;&amp; car.getCol() &gt;= exitCol) {                         count++;                     }                 }             }         }          // Jika mobil utama bergerak vertikal (exit         vertikal)         else {             for (Car car : board.getCars()) {                 if (car.isPrimary()) continue;                  if (!car.isVertical() &amp;&amp; car.getCol() &lt;= </pre>	<p>Heuristik Blocking menghitung jumlah mobil yang menghalangi jalur langsung dari mobil <i>player</i> (P) ke <i>exit</i> (K).</p>

```

player.getCol() + player.getLength() - 1 &&
    car.getCol() >= player.getCol()) {
    if (exitRow > primRow && car.getRow() >
primRow && car.getRow() <= exitRow) {
        count++;
    } else if (exitRow < player.getRow() &&
car.getRow() < player.getRow() && car.getRow() >= exitRow) {
        count++;
    }
}
}
return count;
}

@Override
public String getName() {
    return "Blocking Heuristic";
}
}

```

```

package backend.util;

import backend.model.Board;
import backend.model.Car;

public class HeuristicManhattan implements Heuristic {
    @Override
    public int estimate(Board board) {
        // Dapatkan posisi mobil utama 'P'
        Car player = board.getCar('P');
        int exitRow = board.getExitRow();
        int exitCol = board.getExitCol();

        // Jika mobil utama bergerak horizontal
        if (player.isHorizontal()) {
            int rightmostCol = player.getCol() +
player.getLength() - 1;
            if (exitCol > rightmostCol) {
                return Math.abs(player.getRow() - exitRow) +
Math.abs(rightmostCol - exitCol);
            }
            else {
                return Math.abs(player.getRow() - exitRow) +
Math.abs(player.getCol() - exitCol);
            }
        }
        // Jika mobil utama bergerak vertikal
        else {
            int bottomRow = player.getRow() +
player.getLength() - 1;
            if (exitRow > bottomRow) {
                return Math.abs(player.getCol() - exitCol) +

```

Heuristik Manhattan mengukur jarak terpendek baik secara horizontal atau vertikal yang diperlukan untuk *player* (P) mencapai tujuan atau *exit* (K).

<pre> Math.abs(bottomRow - exitRow);         }         else {             return Math.abs(player.getCol() - exitCol) + Math.abs(player.getRow() - exitRow);         }     }      @Override     public String getName() {         return "Manhattan Heuristic";     } } </pre>	
---	--

### 3. Model

Source Code	Keterangan
<pre> package backend.model;  import java.util.*;  public class Board {     private final int rows;     private final int cols;     private final Map&lt;Character, Car&gt; cars;     private final char[][] grid;     private final long[][][] zobristTable;     private final int exitRow, exitCol;     private final long zobristKey;      public Board(int rows, int cols, List&lt;Car&gt; cars, long[][][] zobristTable, int exitRow, int exitCol) {         this.rows = rows;         this.cols = cols;         this.zobristTable = zobristTable;         this.cars = new HashMap&lt;&gt;();         this.grid = new char[rows][cols];          for (char[] row : grid) {             Arrays.fill(row, '.');         }          for (Car c : cars) {             this.cars.put(c.getId(), c);             placeOnGrid(c);         }          this.zobristKey = computeZobrist(zobristTable);         this.exitRow = exitRow; </pre>	<p>Kelas Board digunakan untuk merepresentasikan papan permainan yang berisi grid, posisi mobil, dan informasi terkait status permainan seperti posisi exit dan jumlah mobil.</p> <p>Fungsi:</p> <ul style="list-style-type: none"> <li>generateNeighbors(): Menghasilkan semua tetangga dari posisi board saat ini, yaitu semua kemungkinan pergerakan yang dapat dilakukan oleh mobil.</li> <li>applyMove(): Menerapkan pergerakan mobil pada board dan mengembalikan board baru setelah pergerakan tersebut.</li> <li>isGoal(): Mengecek apakah kondisi board saat ini adalah solusi (apakah mobil utama mencapai exit).</li> <li>placeOnGrid(): Menempatkan mobil pada posisi yang benar di dalam grid.</li> <li>copy(): Membuat salinan dari board saat ini.</li> </ul>



```

        this.exitCol = exitCol;
    }

    private void placeOnGrid(Car c) {
        int r = c.getRow(), c0 = c.getCol();
        for (int i = 0; i < c.getLength(); i++) {
            grid[r + (c.isHorizontal() ? 0 : i)]
                [c0 + (c.isHorizontal() ? i : 0)] = c.getId();
        }
    }

    public List<Board> generateNeighbors(long[][][]
zobristTable) {
        List<Board> neighbors = new ArrayList<>();
        for (Car c : cars.values()) {
            int delta = -1;
            int lastValidDelta = 0;
            while (canMove(c, delta)) {
                lastValidDelta = delta;
                delta--;
            }
            if (lastValidDelta != 0) {
                neighbors.add(applyMove(c.getId(),
lastValidDelta, zobristTable));
            }

            delta = 1;
            lastValidDelta = 0;
            while (canMove(c, delta)) {
                lastValidDelta = delta;
                delta++;
            }
            if (lastValidDelta != 0) {
                neighbors.add(applyMove(c.getId(),
lastValidDelta, zobristTable));
            }
        }
        return neighbors;
    }

    private boolean canMove(Car c, int delta) {
        int newR = c.getRow() + (c.isHorizontal() ? 0 : delta);
        int newC = c.getCol() + (c.isHorizontal() ? delta : 0);
        int endR = newR + (c.isHorizontal() ? 0 : c.getLength()
- 1);
        int endC = newC + (c.isHorizontal() ? c.getLength() - 1
: 0);

        if (newR < 0 || newC < 0 || endR >= rows || endC >=
cols) {
            return false;
        }
    }

```

```

        Set<String> oldPositions = new HashSet<>();
        int r0 = c.getRow();
        int c0 = c.getCol();
        for (int i = 0; i < c.getLength(); i++) {
            int rr = r0 + (c.isHorizontal() ? 0 : i);
            int cc = c0 + (c.isHorizontal() ? i : 0);
            oldPositions.add(rr + "," + cc);
        }

        for (int i = 0; i < c.getLength(); i++) {
            int rr = newR + (c.isHorizontal() ? 0 : i);
            int cc = newC + (c.isHorizontal() ? i : 0);
            char cell = grid[rr][cc];
            if (cell != '.' && !oldPositions.contains(rr + ","
+ cc)) {
                return false;
            }
        }

        return true;
    }

    public Board applyMove(char carId, int delta, long[][][]
zobristTable) {
        List<Car> newCars = new ArrayList<>();
        for (Car c : cars.values()) newCars.add(c.copy());

        for (Car c : newCars) {
            if (c.getId() == carId) {
                c.move(delta);
                break;
            }
        }

        return new Board(rows, cols, newCars, zobristTable,
exitRow, exitCol);
    }

    private long computeZobrist(long [][][] zobristTable) {
        long h = 0L;
        for (Car c : cars.values()) {
            int r = c.getRow();
            int c0 = c.getCol();
            int len = c.getLength();
            int idx = c.getId() - 'A';

            for (int i = 0; i < len; i++) {
                int rr = r + (c.isHorizontal() ? 0 : i);
                int cc = c0 + (c.isHorizontal() ? i : 0);
                h ^= zobristTable[rr][cc][idx];
            }
        }

        return h;
    }
}

```

```

@Override
public boolean equals(Object o) {
    if (!(o instanceof Board)) return false;
    Board board = (Board) o;
    return this.zobristKey == board.zobristKey;
}

public boolean isGoal() {
    Car player = cars.get('P');
    if (player == null) return false;

    if (player.isHorizontal()) {
        if (getExitCol() < 0) {
            int leftEndCol = player.getCol() - 1;
            return (player.getRow() == exitRow) &&
(leftEndCol == exitCol);
        } else {
            int rightEndCol = player.getCol() +
player.getLength();
            return (player.getRow() == exitRow) &&
(rightEndCol == exitCol);
        }
    } else {
        if (getExitRow() < 0) {
            int topEndRow = player.getRow() - 1;
            return (topEndRow == exitRow) &&
(player.getCol() == exitCol);
        } else {
            int bottomEndRow = player.getRow() +
player.getLength();
            return (bottomEndRow == exitRow) &&
(player.getCol() == exitCol);
        }
    }
}

public void printDebugInfo() {
    System.out.println("\n[DEBUG BOARD STATE]");
    System.out.println("Exit position: (" + exitRow + "," +
exitCol + ")");
    System.out.println("Zobrist key: " + zobristKey);
    System.out.println("Cars positions:");

    cars.forEach((id, car) -> {
        System.out.printf("  %s: %s\n", id,
car.toString());
    });

    System.out.println("\nCurrent grid:");
    System.out.println(this.toString());
}

```

```

public int hashCode() {
    return Long.hashCode(zobristKey);
}

public Car getCar(char id) {
    return cars.get(id).copy();
}

public int getExitRow() {
    return exitRow;
}

public int getExitCol() {
    return exitCol;
}

public long[][][] getZobristTable() {
    return zobristTable;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (char[] row : grid) {
        for (char cell : row) {
            sb.append(cell).append(" ");
        }
        sb.append("\n");
    }
    return sb.toString();
}

public List<Car> getCars() {
    return new ArrayList<>(cars.values());
}

public int getRows() {
    return rows;
}

public int getCols() {
    return cols;
}
}

```

```

package backend.model;

public class Car {
    private final char id;
    private final boolean isHorizontal;
    private final int length;
    private int row;
    private int col;
}

```

Kelas Car digunakan untuk merepresentasikan setiap mobil yang ada di grid permainan, termasuk mobil utama (P) dan mobil-mobil lain yang menghalangi jalur.

Fungsi:

```

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

    public char getId() {
        return id;
    }

    public boolean isHorizontal() {
        return isHorizontal;
    }

    public int getLength() {
        return length;
    }

    public int getRow() {
        return row;
    }

    public int getCol() {
        return col;
    }

    public void setRow(int row) {
        this.row = row;
    }

    public void setCol(int col) {
        this.col = col;
    }

    public void move(int delta) {
        if (isHorizontal) {
            col += delta;
        } else {
            row += delta;
        }
    }

    public Car copy() {
        return new Car(id, isHorizontal, length, row, col);
    }

    @Override
    public String toString() {
        return "Car{" +

```

move(delta) digunakan untuk memindahkan mobil ke posisi baru berdasarkan delta, yang menunjukkan perpindahan mobil (maju atau mundur).  
 copy() membuat salinan (*clone*) dari mobil.

<pre>         "id=" + id +         ", isHorizontal=" + isHorizontal +         ", length=" + length +         ", row=" + row +         ", col=" + col +         '&gt;';      }      public boolean isVertical() {         return !isHorizontal;     }      public boolean isPrimary() {         return this.id == 'P';     } } </pre>	
<pre> package backend.model;  public class Move {     private final char carId;     private final int delta;      public Move(char carId, int delta) {         this.carId = carId;         this.delta = delta;     }      public char getCarId() {         return carId;     }      public int getDelta() {         return delta;     }      public int getCost() {         return 1;     }      @Override     public String toString() {         return "Move{" +             "carId=" + carId +             ", delta=" + delta +             '}';     } } </pre>	<p>Kelas Move digunakan untuk menyimpan informasi tentang langkah-langkah yang dilakukan oleh mobil di dalam permainan.</p> <p>Fungsi:          applyMove():          Mengaplikasikan gerakan pada board dan mengembalikan board baru setelah pergerakan.</p>
<pre> package backend.model;  import backend.exception.InvalidInputException; import backend.exception.ParserException; </pre>	<p>Parse file .txt menjadi Board, mendeteksi exit 'K' di atas/ samping/ bawah.</p> <p>Format:</p>

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class Parser {
    @SuppressWarnings("unused")
    public static Board parse(String filePath)
        throws IOException, InvalidInputException {
        try (BufferedReader br = new BufferedReader(new
FileReader(filePath))) {
            // 1) Baca dimensi
            String header = br.readLine();
            if (header == null) throw new
InvalidInputException("File kosong");
            String[] dims = header.trim().split("\\s+");
            if (dims.length != 2)
                throw new InvalidInputException("Baris dimensi
harus berisi 2 angka");
            int rows = Integer.parseInt(dims[0]);
            int cols = Integer.parseInt(dims[1]);

            // 2) Validasi jumlah Car
            br.mark(1024);
            String countLine = br.readLine();
            if (countLine == null) throw new
InvalidInputException("Tidak ada baris jumlah mobil");
            countLine = countLine.trim();
            if (!countLine.matches("\\d+")) {
                throw new InvalidInputException("Baris kedua
harus angka jumlah mobil");
            }
            int declaredCars = Integer.parseInt(countLine);
            if (declaredCars < 1 || declaredCars > 24) {
                throw new InvalidInputException(
                    "Jumlah mobil harus antara 1 dan 24,
ditemukan: " + declaredCars);
            }

            // 3) Cek exit di atas board
            br.mark(1024);
            String firstRaw = br.readLine();
            int exitRow = Integer.MIN_VALUE, exitCol = -1;
            boolean exitAbove = false;
            if (firstRaw != null) {
                long countK = firstRaw.chars().filter(ch -> ch
== 'K').count();
                boolean validChars = firstRaw.chars()
                    .allMatch(ch -> ch == 'K' || ch == '.' ||
Character.isWhitespace(ch));
                if (countK == 1 && validChars &&
firstRaw.length() <= cols) {

```

Baris 1 : ROWS COLS

Baris 2 : count mobil

Baris selanjutnya:

memparsing Board dari input

Parameter filePath berupa path ke file input

*Return* berupa Board yang sudah terbangun

```

        exitRow = -1;
        exitCol = firstRow.indexOf('K');
        exitAbove = true;
    }
}
if (!exitAbove) br.reset();

// 4) Baca rows baris grid
List<String> rawGrid = new ArrayList<>();
for (int r = 0; r < rows; r++) {
    String line = br.readLine();
    if (line == null)
        throw new InvalidInputException("Grid
kurang dari " + rows + " baris");

    String compact = line.replaceAll("\\s+", "");

    if (compact.length() == cols) {
        if (compact.contains("K")) {
            if (compact.chars().filter(ch -> ch ==
'K').count() != 1) {
                throw new
InvalidInputException("Grid harus mengandung tepat satu 'K'");
            }
            exitRow = r;
            exitCol = compact.indexOf('K');
            compact = compact.replace('K', '.');
        }
        rawGrid.add(compact);
    }
    else if (compact.length() == cols + 1) {
        long countK = compact.chars().filter(ch ->
ch == 'K').count();
        if (countK != 1)
            throw new InvalidInputException(
                "Baris grid ke-" + (r+1) + " ekstra
harus tepat satu 'K'");
        int kIdx = compact.indexOf('K');
        if (kIdx == 0) {
            exitRow = r;
            exitCol = -1;
            rawGrid.add(compact.substring(1));
        } else if (kIdx == cols) {
            exitRow = r;
            exitCol = cols;
            rawGrid.add(compact.substring(0,
cols));
        } else {
            throw new InvalidInputException(
                "Baris grid ke-" + (r+1) + " ekstra
'K' harus di tepi");
        }
    }
}

```



```

        else {
            throw new InvalidInputException(
                "Panjang baris grid ke-" + (r+1) +
                " harus " + cols + " atau " +
(cols+1));
        }
    }

    // 5) Cek exit di bawah board
    br.mark(1024);
    String lastRaw = br.readLine();
    boolean exitBelow = false;
    if (lastRaw != null) {
        long countK = lastRaw.chars().filter(ch -> ch
== 'K').count();
        boolean validChars = lastRaw.chars()
.allMatch(ch -> ch == 'K' || ch == '.' ||
Character.isWhitespace(ch));
        if (countK == 1 && validChars &&
lastRaw.length() <= cols) {
            exitRow = rows;
            exitCol = lastRaw.indexOf('K');
            exitBelow = true;
        }
    }
    if (!exitBelow) br.reset();

    // 6) Validasi exit ditemukan
    if (exitRow == Integer.MIN_VALUE) {
        throw new InvalidInputException(
            "Tidak ditemukan exit 'K' di atas, samping,
atau bawah");
    }

    // 7) Kumpulkan posisi tiap mobil
    Map<Character, List<int[]>> posMap = new
HashMap<>();

    char[][] grid = new char[rows][cols];
    for (int r = 0; r < rows; r++) {
        String rowStr = rawGrid.get(r);
        grid[r] = rowStr.toCharArray();
        for (int c = 0; c < cols; c++) {
            char ch = grid[r][c];
            if (ch == '.' || ch == 'K') continue;
            posMap.computeIfAbsent(ch, k -> new
ArrayList<>())
                .add(new int[]{r, c});
        }
    }

    // 8) Bangun daftar Car
    List<Car> cars = new ArrayList<>();
    for (var e : posMap.entrySet()) {

```

```

        char id = e.getKey();
        List<int[]> coords = e.getValue();
        int length = coords.size();

        if (length == 1) {
            throw new InvalidInputException("Mobil
dengan ukuran 1x1 tidak diperbolehkan.");
        }

        int minR = coords.stream().mapToInt(p ->
p[0]).min().getAsInt();
        int minC = coords.stream().mapToInt(p ->
p[1]).min().getAsInt();
        boolean horiz = coords.stream().allMatch(p ->
p[0] == minR);

        cars.add(new Car(id, horiz, length, minR,
minC));
    }

    if (cars.size() - (posMap.containsKey('P') ? 1 : 0)
!= declaredCars) {
        throw new InvalidInputException(
            "Jumlah mobil terdeteksi (" + cars.size() +
            ") tidak sesuai deklarasi (" + declaredCars
+ ")");
    }

    // 9) Validasi exit sejajar dengan mobil player (P)
    Car playerCar = cars.stream()
        .filter(car -> car.getId() == 'P')
        .findFirst()
        .orElseThrow(() -> new
InvalidInputException("Mobil pemain (P) tidak ditemukan"));

    boolean exitAligned = false;
    if (playerCar.isHorizontal()) {
        int playerRow = playerCar.getRow();
        exitAligned = (exitRow == playerRow);
    } else {
        int playerCol = playerCar.getCol();
        exitAligned = (exitCol == playerCol);
    }

    if (!exitAligned) {
        throw new InvalidInputException(
            "Pintu keluar (K) harus sejajar dengan
mobil pemain (P): " +
            (playerCar.isHorizontal() ? "horizontal" :
"vertikal"));
    }

    // 10) Generate Zobrist & kembalikan Board

```

<pre>         long[][][] zTable = generateZobristTable(rows, cols);          return new Board(rows, cols, cars, zTable, exitRow, exitCol);     }      private static long[][][] generateZobristTable(int rows, int cols) {         Random rnd = new Random(0);         long[][][] table = new long[rows][cols][26];         for (int r = 0; r &lt; rows; r++) {             for (int c = 0; c &lt; cols; c++) {                 for (int k = 0; k &lt; 26; k++) {                     table[r][c][k] = rnd.nextLong();                 }             }         }         return table;     }      public static Board parseFile(String filePath) throws backend.exception.ParserException {         try {             return parse(filePath);         } catch (IOException   InvalidInputException e) {             throw new ParserException("Error parsing file: " + e.getMessage(), e);         }     } } </pre>	
---	--

#### 4. Graphical User Interface

Source Code	Keterangan
<pre> package gui;  import javafx.application.*; import javafx.geometry.*; import javafx.scene.*; import javafx.scene.control.*; import javafx.scene.layout.*; import javafx.stage.*;  import java.io.*; import java.nio.file.*; import java.util.*;  import backend.model.*; import backend.util.*; import backend.exception.*; import backend.algorithm.*; </pre>	<p>Fungsi start pada kelas RushHourGUI merupakan titik masuk utama aplikasi GUI yang dijalankan ketika aplikasi dimulai. Fungsi ini membangun struktur dasar antarmuka pengguna dengan membuat dua panel utama: panel kontrol di sisi kiri yang berisi berbagai elemen interaksi (tombol pemilihan file, dropdown algoritma dan heuristik, tombol eksekusi dan statistik hasil) dan panel tampilan di bagian tengah yang</p>

```

public class RushHourGUI extends Application {
    private BoardView boardView;
    private File selectedFile;
    private Board board;
    private Label fileNameLabel;
    private Button runButton;
    private Button saveButton;
    private ComboBox<String> algorithmCombo;
    private ComboBox<String> heuristicCombo;
    private Label statsLabel;
    private HBox animationControls;
    private List<Board> solution;
    private PathfindingAlgorithm lastAlgorithm;

    @Override
    public void start(Stage primaryStage) {
        BorderPane root = new BorderPane();

        // Left panel with controls
        VBox controlPanel =
createControlPanel(primaryStage);
        root.setLeft(controlPanel);

        // Center panel with board
        boardView = new BoardView();
        boardView.setPrefSize(500, 500);
        boardView.setStyle("-fx-background-color: white;
-fx-border-color: black; -fx-border-width: 2px;");

        // Add some padding around the board
        StackPane centerWrapper = new StackPane(boardView);
        centerWrapper.setPadding(new Insets(20));

        // Add controls below the board
        animationControls = createAnimationControls();

        VBox centerPanel = new VBox(10);
        centerPanel.getChildren().addAll(centerWrapper,
animationControls);
        centerPanel.setAlignment(Pos.CENTER);
        root.setCenter(centerPanel);

        Scene scene = new Scene(root, 800, 600);

        // Add basic styling
        scene.getRoot().setStyle("-fx-font-family: 'Arial';
-fx-background-color: #f5f5f5;");

        primaryStage.setTitle("Rush Hour Puzzle Solver");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

menampung visualisasi papan dan kontrol animasi. Fungsi ini juga mengatur styling dasar dan ukuran jendela aplikasi, serta mempersiapkan ScrollPane untuk mendukung navigasi papan berukuran besar, memastikan antarmuka aplikasi berfungsi dengan baik pada berbagai kondisi konfigurasi permainan.

Fungsi initializeBoard pada kelas BoardView bertanggung jawab untuk mempersiapkan visualisasi papan permainan berdasarkan konfigurasi yang diterima. Fungsi ini melakukan inisialisasi ulang papan dengan menghapus semua elemen visual sebelumnya, kemudian menyesuaikan ukuran sel berdasarkan dimensi papan. Selanjutnya, fungsi ini membangun representasi visual dengan menggambar grid, penanda pintu keluar, dan menempatkan elemen mobil sesuai dengan posisi dan orientasinya. Setiap mobil diberi warna unik dengan primary piece ditandai dengan warna merah, serta sudut yang dibulatkan untuk estetika visual yang lebih baik.

Dengan adanya GUI ini visualisasi hasil akan terlihat, dengan adanya animasi perlangkah.

```

package gui;

import backend.model.Board;
import backend.model.Car;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.animation.TranslateTransition;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.util.Duration;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class BoardView extends Pane {
    private double cellSize = 60.0;
    private Map<Character, Rectangle> carRectangles = new
HashMap<>();
    private Board currentBoard;
    private List<Board> solution;
    private IntegerProperty currentStep = new
SimpleIntegerProperty(0);
    private Timeline animation;
    private Rectangle exitMarker;

    /**
     * Initialize the board view with initial board state.
     * @param board The initial board configuration
     */
    public void initializeBoard(Board board) {
        this.currentBoard = board;
        this.getChildren().clear();
        carRectangles.clear();

        // Set view size based on board dimensions
        int rows = board.getRows();
        int cols = board.getCols();
        setPrefSize(cols * cellSize, rows * cellSize);

        // Draw board grid
        drawGrid(rows, cols);

        // Draw exit marker
        drawExitMarker(board);

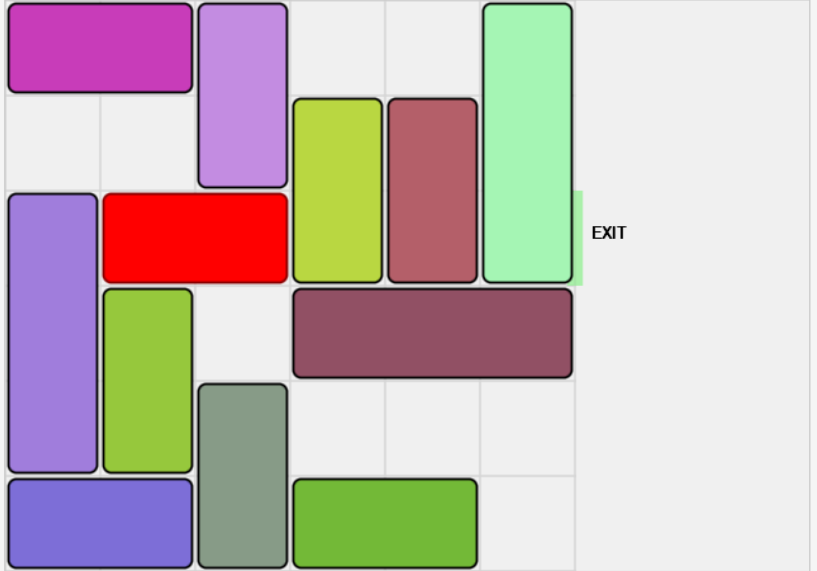
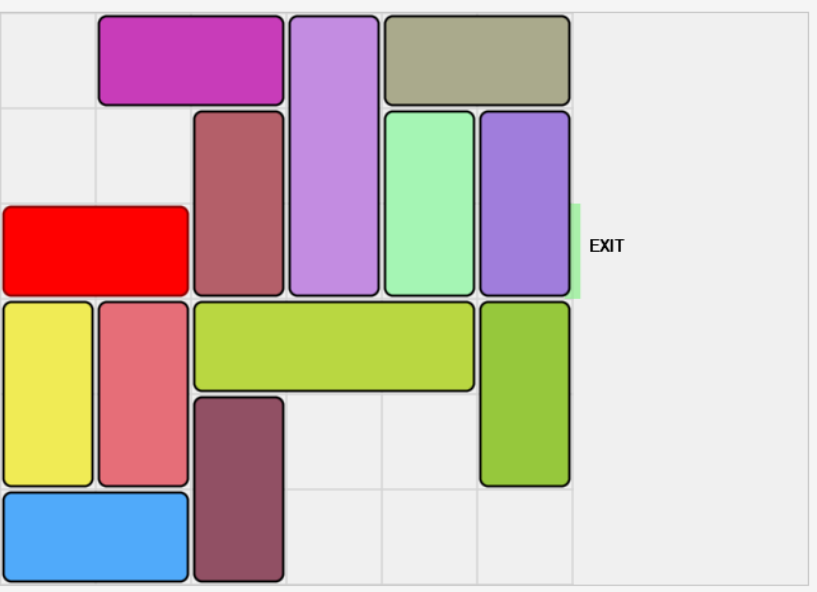
        // Draw cars
        for (Car car : board.getCars()) {
            Rectangle rect = createCarRectangle(car);
            carRectangles.put(car.getId(), rect);
            this.getChildren().add(rect);
        }
    }

```

```
    }  
}  
  
/**  
 * Set the solution path for animation.  
 * @param solution List of board states from initial to  
goal  
 */  
public void setSolution(List<Board> solution) {  
    this.solution = solution;  
    currentStep.set(0);  
    updateBoardState(solution.get(0));  
}
```

## D. Pengujian

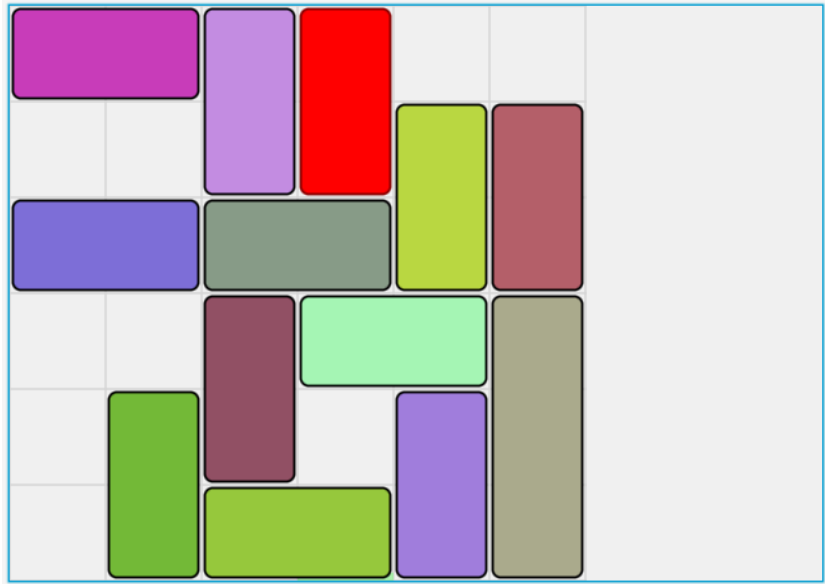
Pengujian untuk masing-masing Algoritma, menggunakan testcase berikut.

testcase1	
6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	 <p>The diagram for testcase1 shows a 6x6 grid. The blocks are: a purple 2x2 block at (1,1)-(2,2); a light purple 2x1 block at (2,2)-(3,2); a light green 2x1 block at (1,5)-(2,5); a red 2x1 block at (2,3)-(3,3); a yellow-green 2x1 block at (2,4)-(3,4); a brown 2x1 block at (2,5)-(3,5); a purple 2x1 block at (3,1)-(4,1); a green 2x1 block at (3,3)-(4,3); a brown 2x1 block at (3,4)-(4,4); a purple 2x1 block at (4,1)-(5,1); a green 2x1 block at (4,3)-(5,3); a brown 2x1 block at (4,4)-(5,4); a yellow-green 2x1 block at (5,3)-(6,3); a brown 2x1 block at (5,4)-(6,4); and a light green 2x1 block at (5,5)-(6,5). The EXIT label is at (3,6).</p>
testcase2	
6 6 12 .AABEE ..DBFG PPDBFGK XYCCCH XYI..H RRI...	 <p>The diagram for testcase2 shows a 6x6 grid. The blocks are: a purple 2x2 block at (1,2)-(2,3); a light purple 2x1 block at (2,3)-(3,3); a brown 2x1 block at (1,4)-(2,4); a yellow-green 2x1 block at (2,4)-(3,4); a purple 2x1 block at (2,5)-(3,5); a light green 2x1 block at (3,4)-(4,4); a purple 2x1 block at (3,5)-(4,5); a red 2x1 block at (3,1)-(4,1); a yellow 2x1 block at (4,1)-(5,1); a pink 2x1 block at (4,2)-(5,2); a brown 2x1 block at (4,3)-(5,3); a yellow-green 2x1 block at (4,4)-(5,4); a brown 2x1 block at (5,1)-(6,1); a brown 2x1 block at (5,2)-(6,2); a light green 2x1 block at (5,4)-(6,4); and a light green 2x1 block at (5,5)-(6,5). The EXIT label is at (3,6).</p>
testcase3	

```

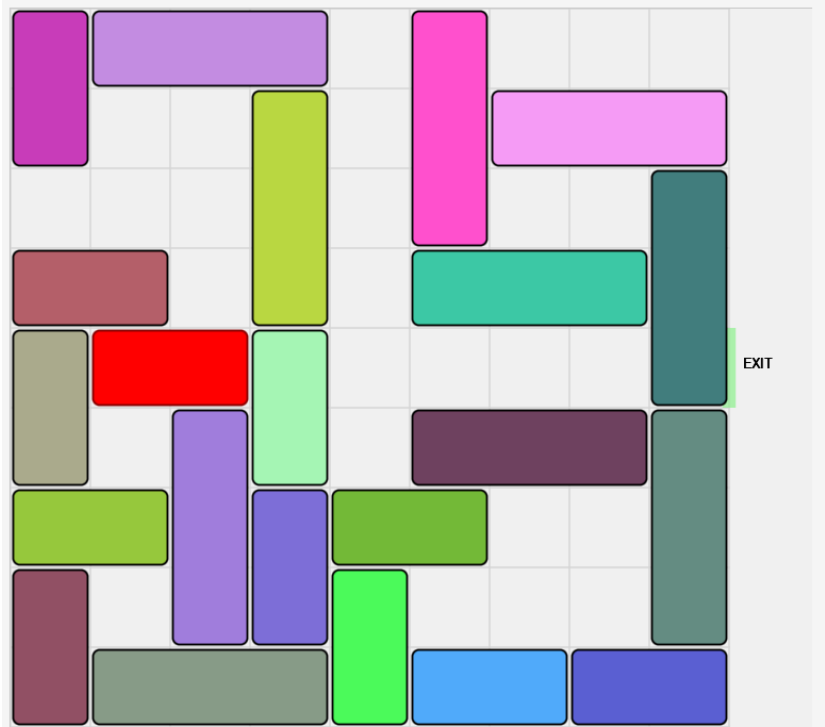
6 6
12
AABP..
..BPCD
LLJJCD
..IFFE
..MI.GE
..MHHGE
      K

```



testcase4

9 9  
21  
ABBB.V...  
A..C.VWWW  
...C.V..T  
DD.C.UUUT  
EPPF....TK  
E.GF.NNNO  
HHGLMM..O  
I.GLS...O  
IJJJSRRQQ



## 1. Uniform Cost Search

testcase1







Rush Hour Puzzle Solver

Select Puzzle File

test8.txt

Select Algorithm:

Uniform Cost Search (UCS)

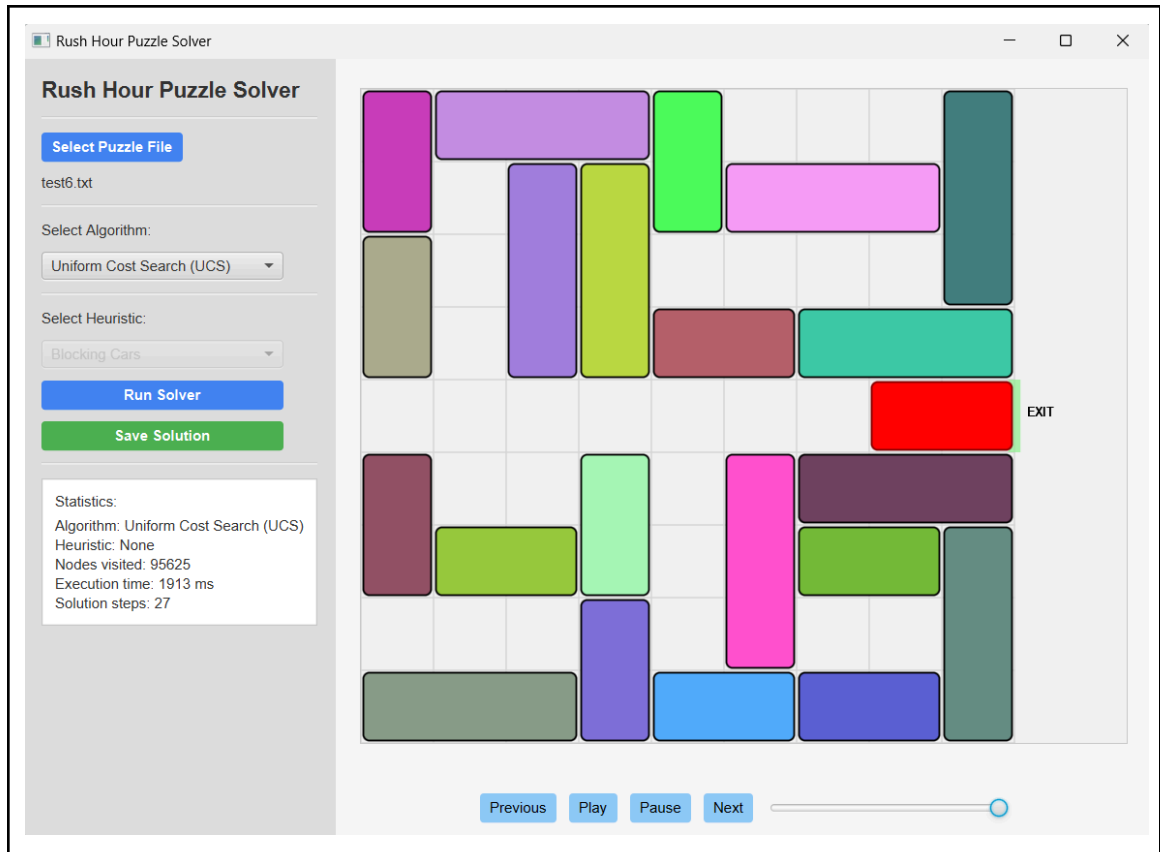
Select Heuristic:

Manhattan Distance

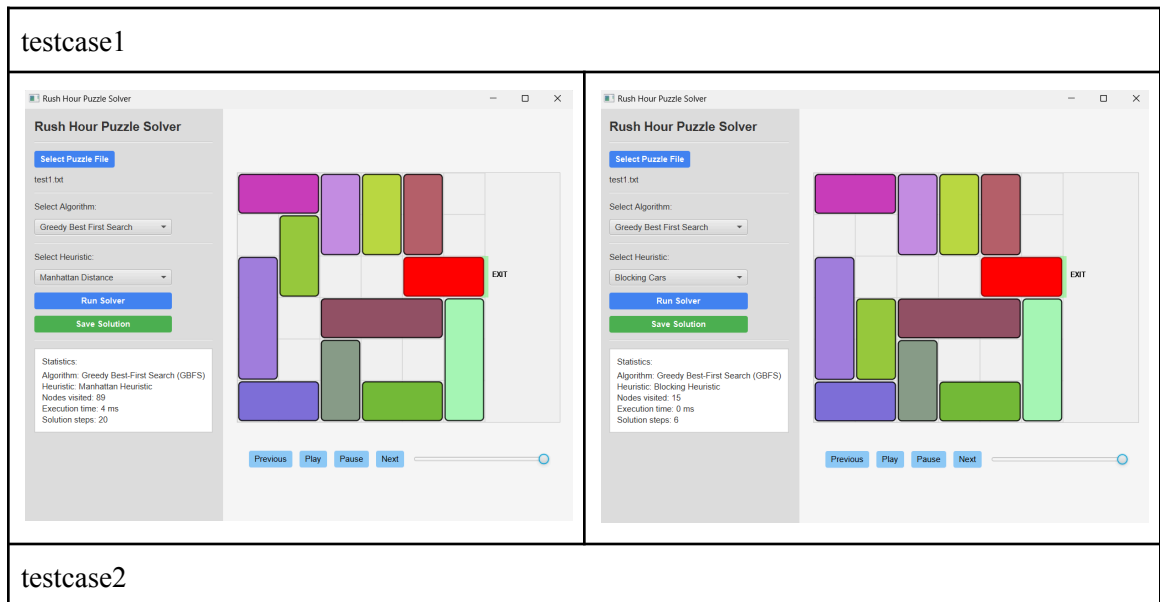
Run Solver

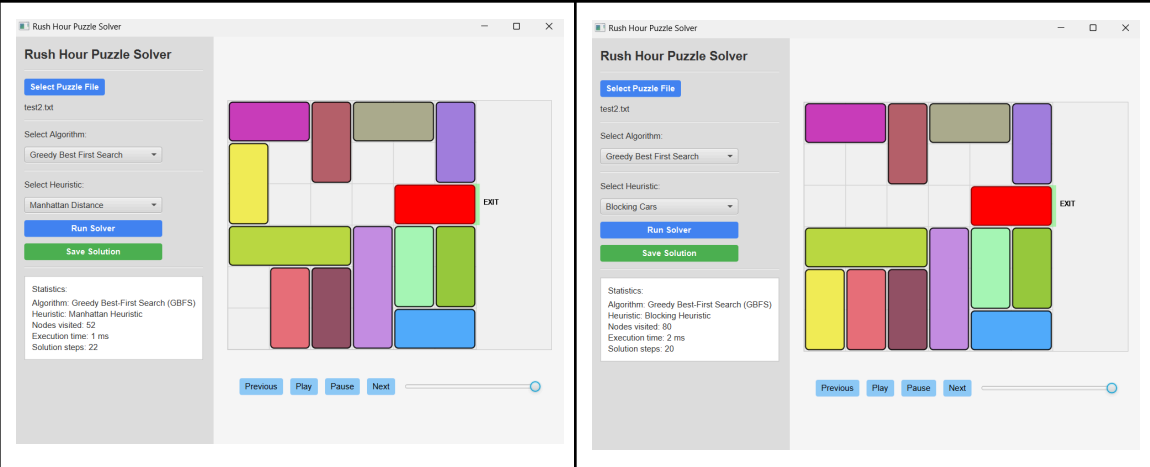
Save Solution

Statistics:  
Algorithm: Uniform Cost Search (UCS)  
Heuristic: None  
Nodes visited: 553  
Execution time: 47 ms  
Solution steps: 21

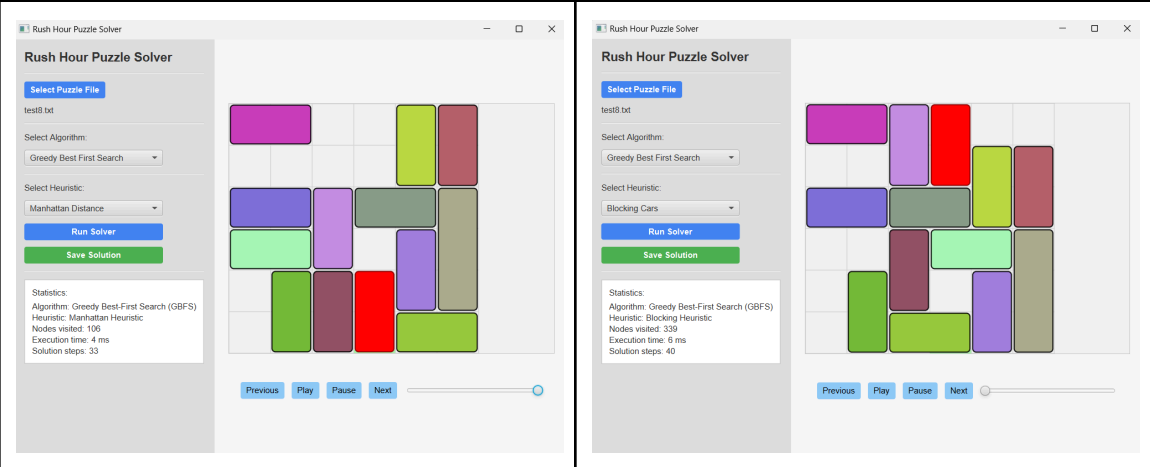


## 2. Greedy Best First Search

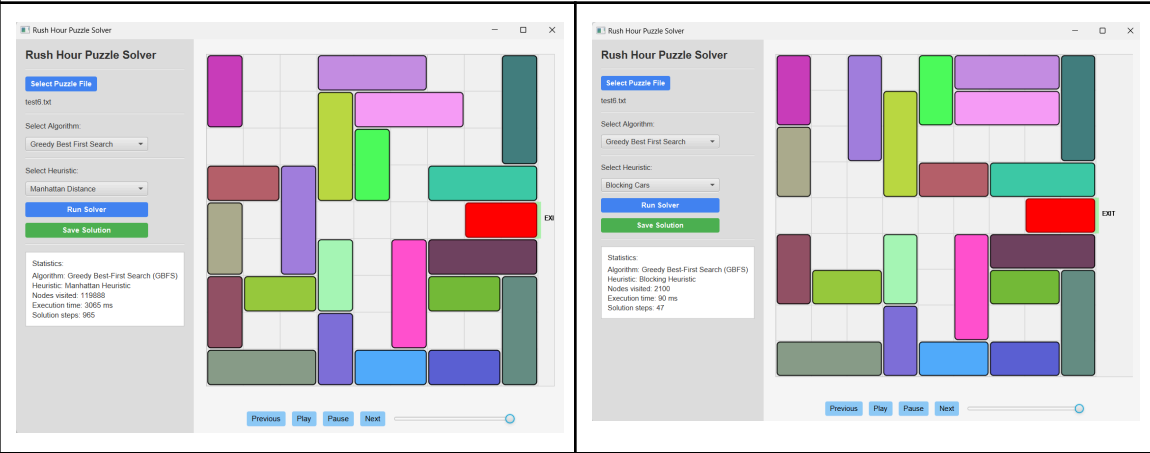




testcase3

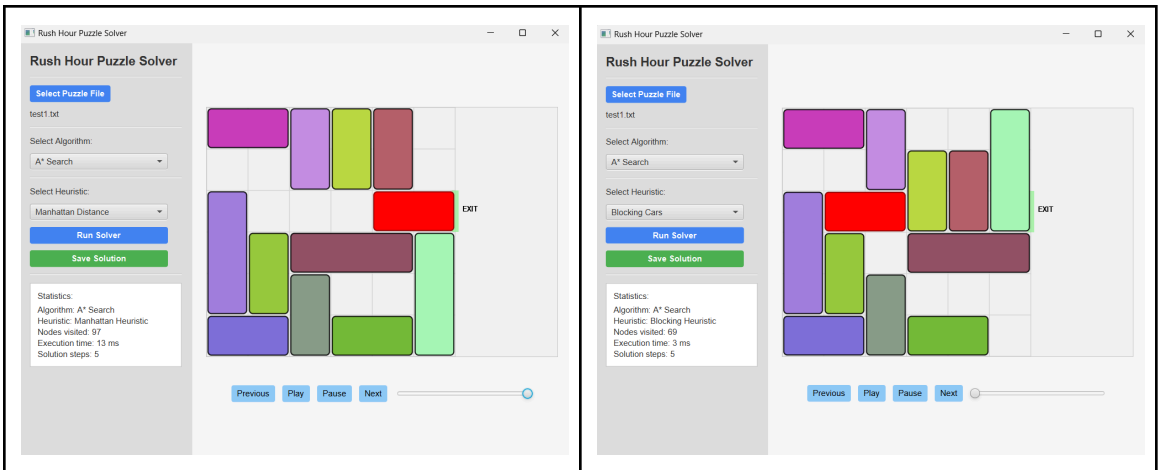


testcase4

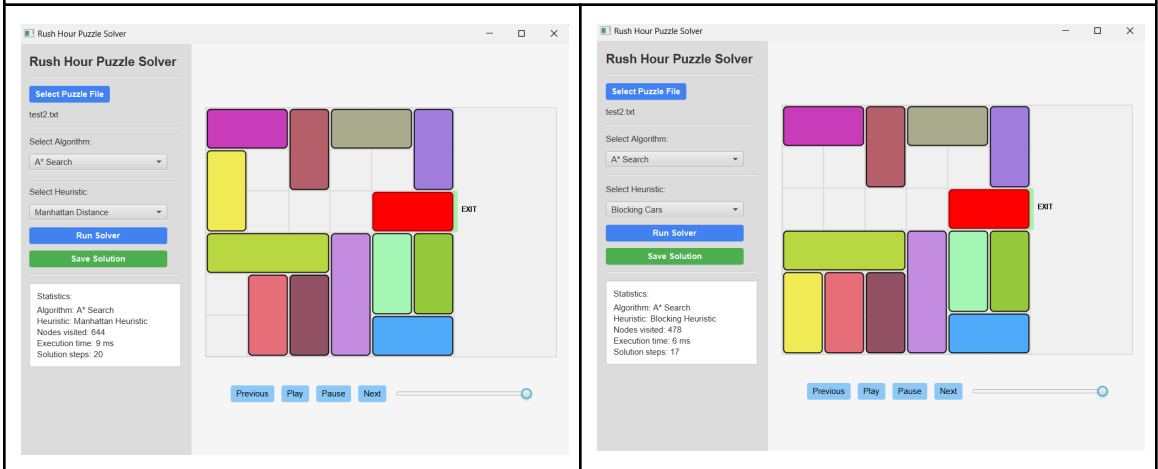


3. A\*

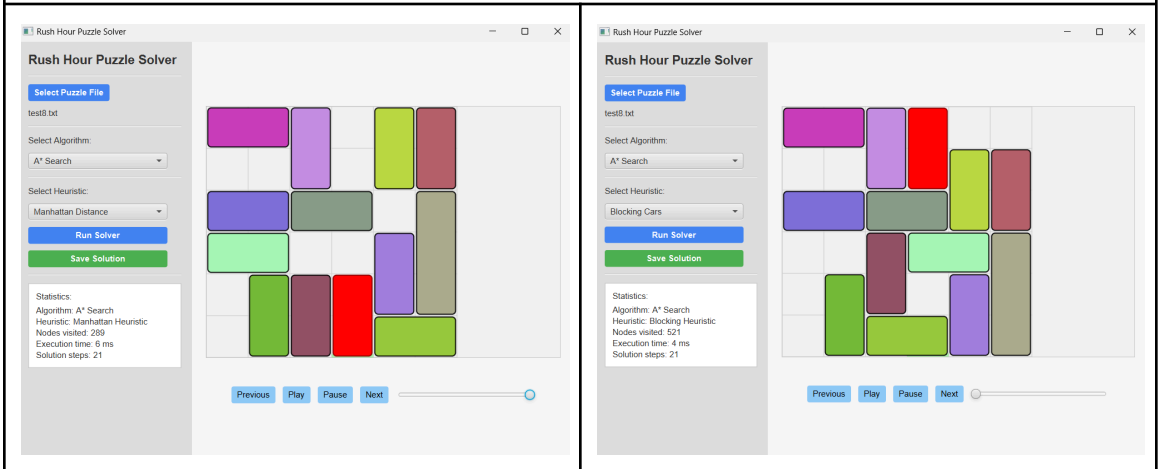
testcase1



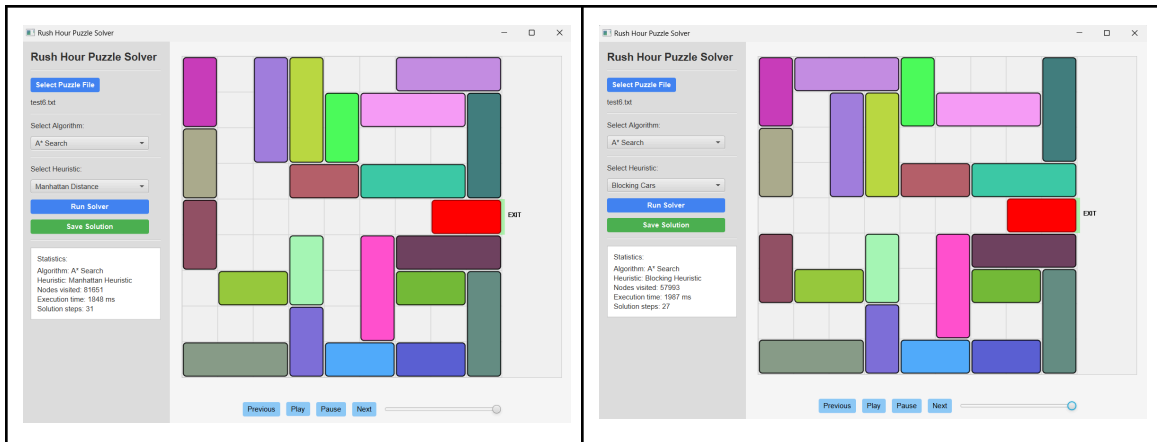
## testcase2



## testcase3



## testcase4



#### 4. Handling

Tidak ada K	
<pre> 6 6 12 GBB.L. GHI.LM GHIPPM CCCZ.M ..JZDD EEJFF. </pre>	
Jumlah N tidak sesuai dengan jumlah mobil	

6 6  
11  
GBB.L.  
GHI.LM  
GHIPPMK  
CC CZ.M  
. .JZDD  
EEJFF.

Rush Hour Puzzle Solver

**Rush Hour Puzzle Solver**

Select Puzzle File

Error: Error parsing file: Jumlah mobil terdeteksi (12) tidak sesuai deklarasi (11)

Select Algorithm:

A\* Search

Select Heuristic:

Manhattan Distance

Run Solver

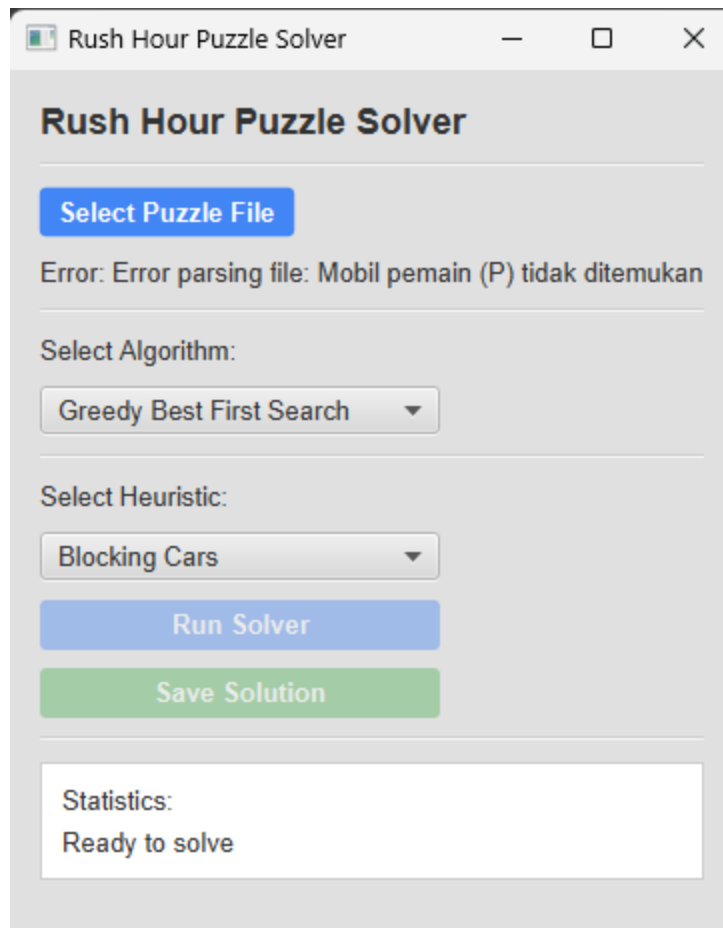
Save Solution

Statistics:  
Ready to solve

Tidak ada primary car

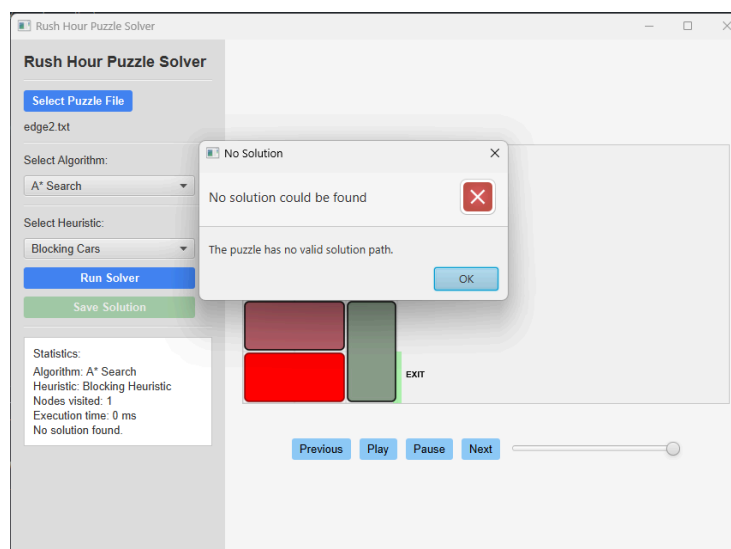


6 6  
12  
GBB.L.  
GHI.LM  
GHI..MK  
CCCZ.M  
..JZDD  
EEJFF.

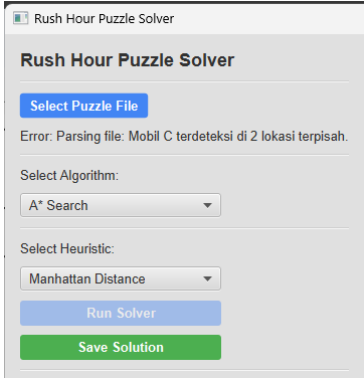


Tidak ada solusi

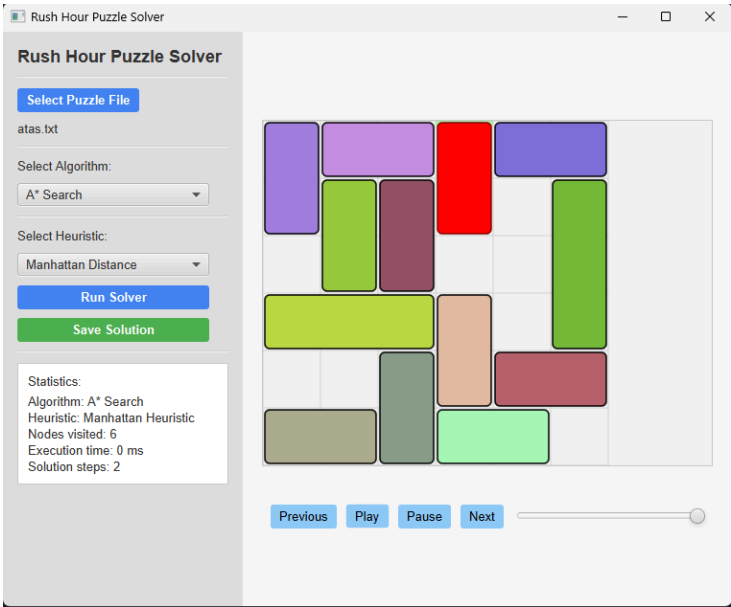
5 3  
5  
AAA  
BBB  
CCC  
DDJ  
PPJK

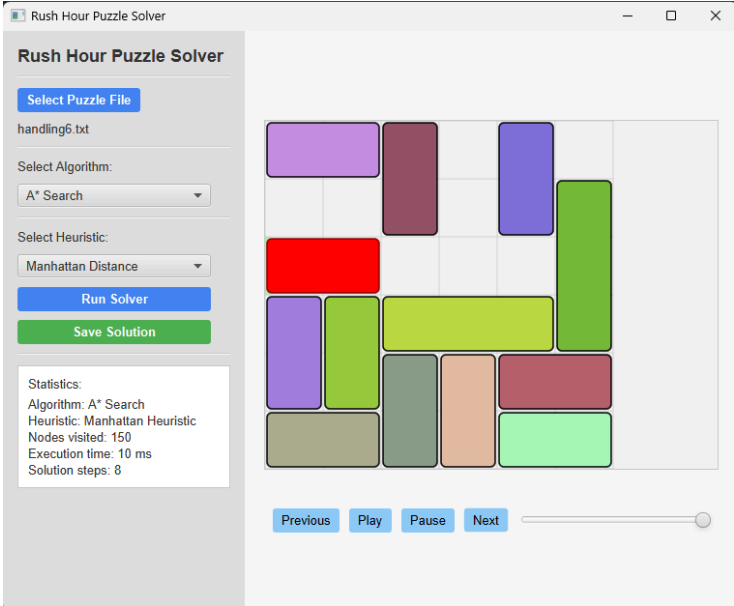
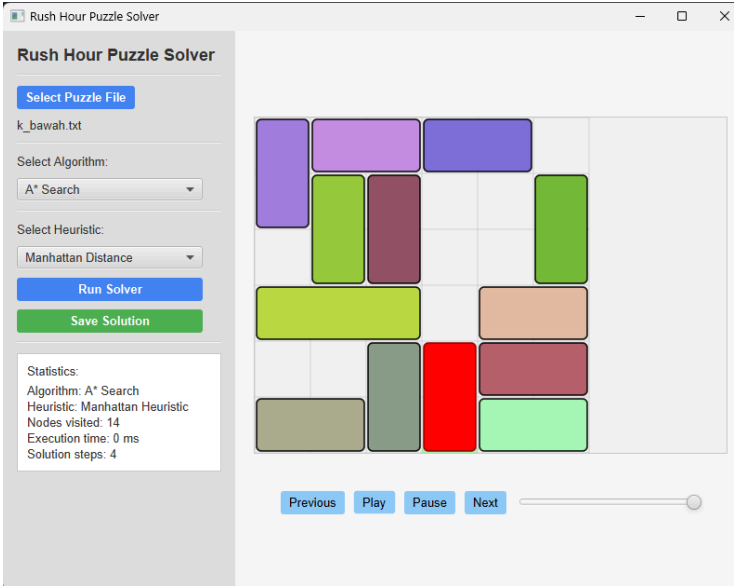


Ada dua mobil identik (huruf sama, tempat berbeda)

6 6 13 GBB.L. GHI.LM GHIPPMK CCCZ.M ..JZDD EEJCC.	
--	---

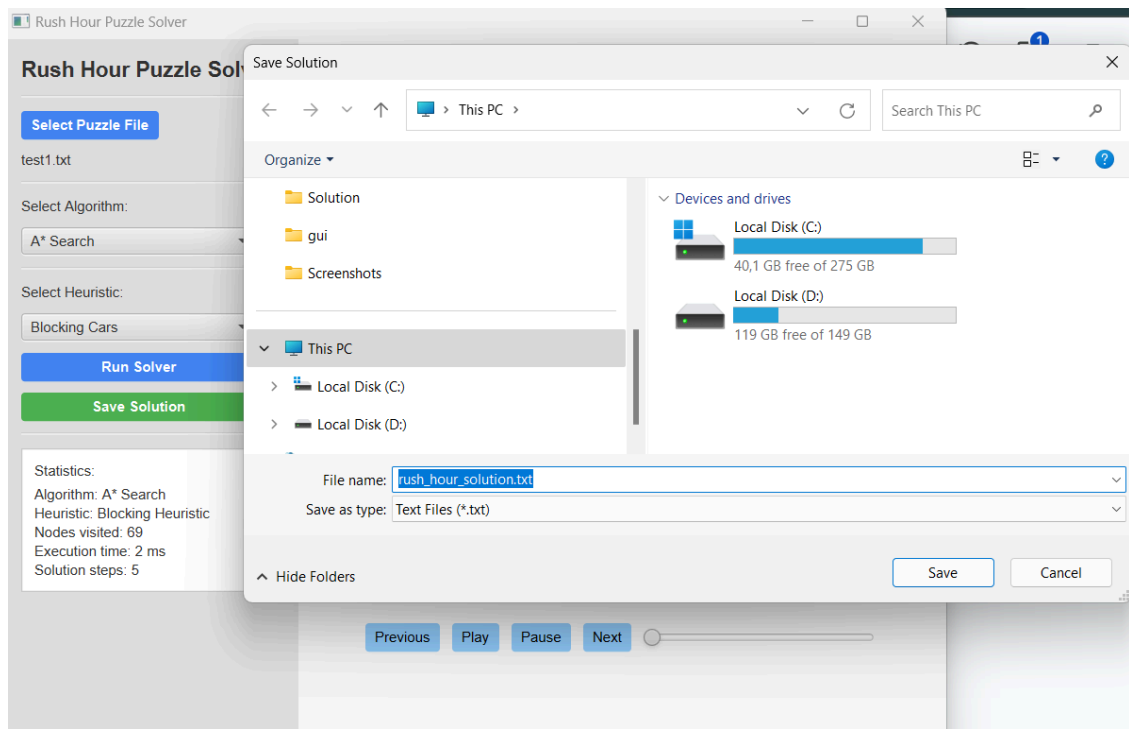
Pintu (K) dari berbagai arah

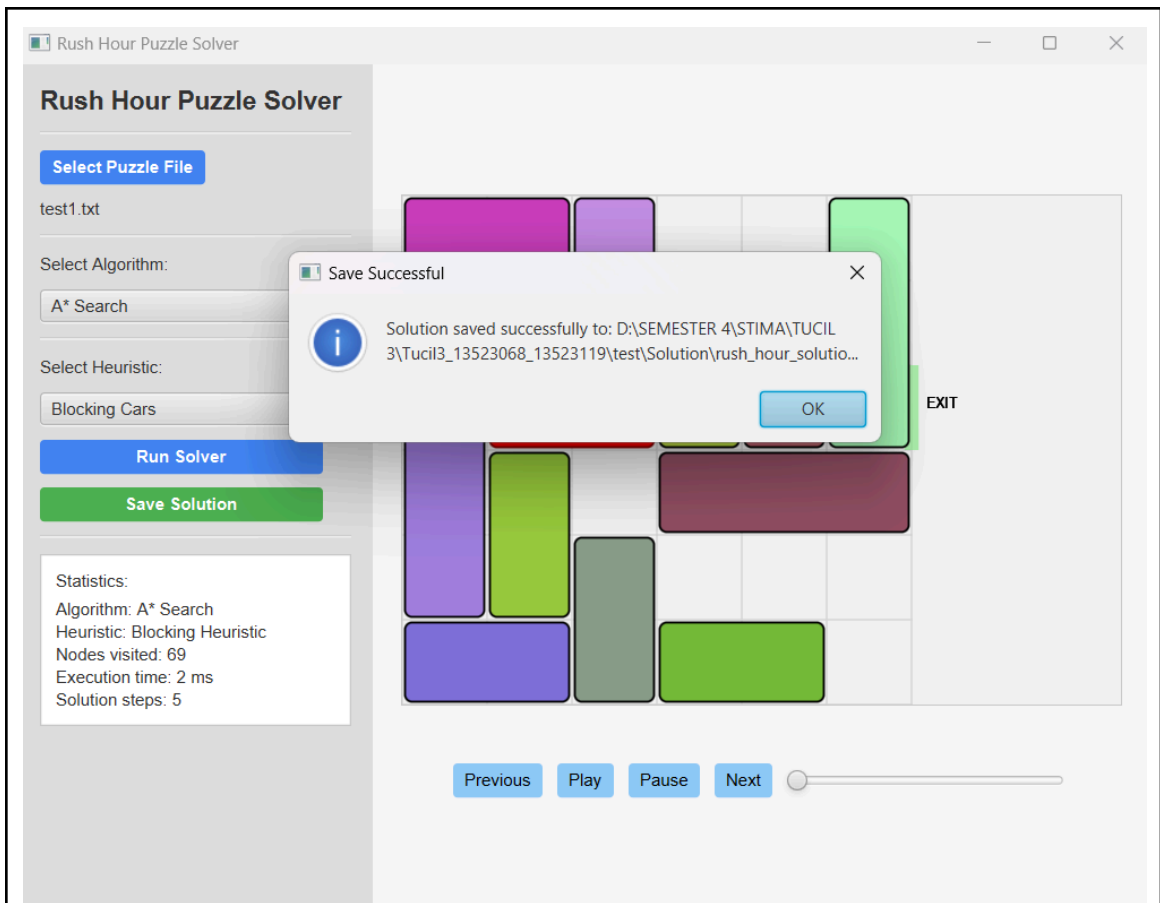
Atas	6 6 12 K GBBLL. GHIP.M .HIP.M CCCZ.M ..JZDD EEJFF.	
------	--	---

Kiri	<pre>6 6 12 GBB.L. GHI.LM K.HIPPM CC CZ.M ..JZDD EEJFF.</pre>	
Bawah	<pre>6 6 12 GBBL.L. GHIP.M .HIP.M CC CZ.Z. ..J.DD EEJFF. K</pre>	

<p>Kanan</p>	<p>6 6 12 GBB.L. GHI.LM .HIPPMK CC CZ.M ..JZDD EEJFF.</p>	
--------------	---	--

Save (yang digunakan adalah testcase1)





## Isi File .txt

```
Algorithm: A* Search
Heuristic: Blocking Cars
Nodes visited: 69
Execution time: 2 ms
```

### Papan Awal:

```
A A B . . F
. . B C D F
G P P C D F
G H . I I I
G H J . . .
L L J M M .
```

### Gerakan 1: C - Atas

```
A A B C . F
. . B C D F
G P P . D F
G H . I I I
G H J . . .
L L J M M .
```

### Gerakan 2: D - Atas

```
A A B C D F
. . B C D F
C D D . . F
```

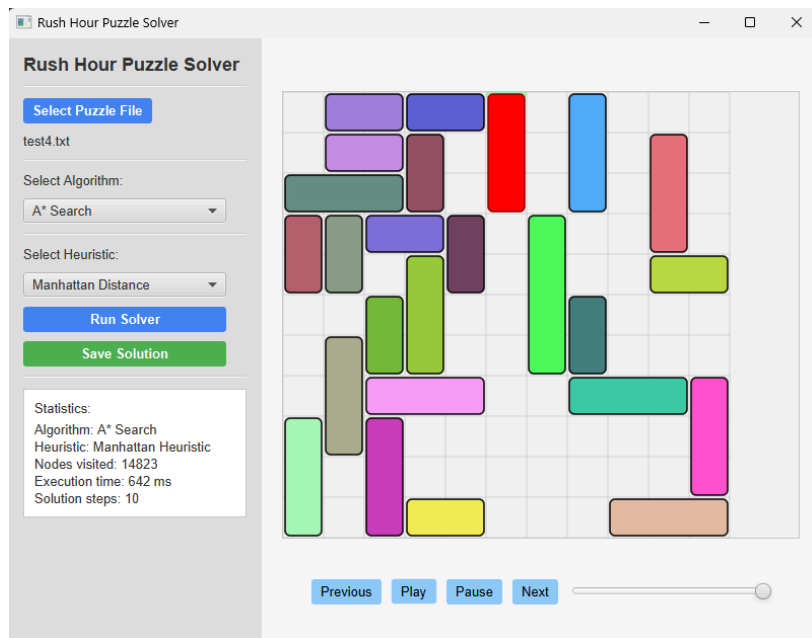
## Animasi Penentuan Solusi (testcase1)



```

11 11
24
      K
.GGI.QQR...
.BBI...R.Y.
....OOR.Y.
DJLLN.S...Y.
DJ..N.S..CC
..MH..ST...
.EMH..ST...
.EAHWWUUUV
FEA..P....V
F.A..P....V
F.XX.P...ZZZ

```

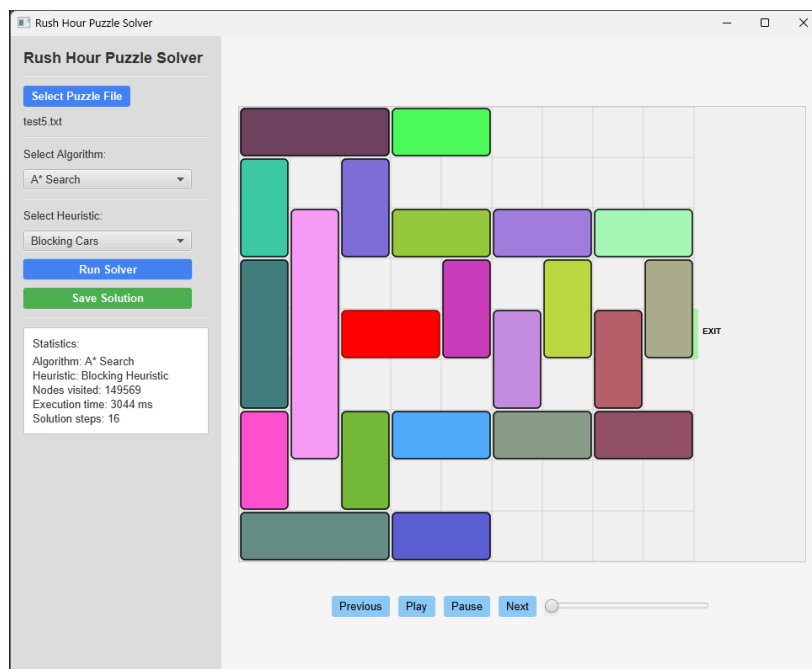


\*Memilih heuristik Manhattan Distance karena sesuai dengan test case yang memiliki jarak yang jauh dari exit (K). Dari hasil juga, lebih optimal untuk heuristik ini dibanding Blocking Cars.

```

9 9
21
NNNSS....
U.L.....
UWLHHGGFF
TW..A.C.E
TWPPABCDEK
TW...B.D.
VWMRRJJII
V.M.....
OOOQQ....

```



\*Memilih heuristik Blocking Cars karena sesuai dengan test case yang terdapat banyak mobil yang menghalangi jalur ke exit (K). Dari hasil juga, lebih optimal untuk heuristik ini dibanding Manhattan Distance.

## E. Pembahasan

Berdasarkan hasil implementasi dan pengujian, heuristik yang digunakan dalam algoritma A\* terbukti bersifat admissible. Ini berarti bahwa estimasi yang diberikan oleh heuristik tidak pernah melebihi biaya sebenarnya dari suatu simpul ke tujuan. Dalam kasus permainan Rush Hour, pendekatan seperti Manhattan Distance atau menghitung jumlah mobil yang menghalangi jalan keluar menghasilkan estimasi yang optimistik. Karena itulah, heuristik ini memenuhi syarat admissibility dan membuat A\* tetap dapat menjamin bahwa solusi yang ditemukan adalah solusi optimal.

A\* dan UCS bekerja dengan pendekatan yang serupa. Keduanya menggunakan antrian prioritas dan memperluas simpul berdasarkan nilai evaluasi tertentu. Perbedaannya adalah, A\* menggabungkan biaya yang sudah ditempuh ( $g(n)$ ) dengan estimasi menuju tujuan ( $h(n)$ ), sedangkan UCS hanya mempertimbangkan biaya yang sudah ditempuh saja. Saat  $h(n)$  diatur nol, A\* akan bertindak sama seperti UCS. Selain itu, karena biaya langkah dalam puzzle Rush Hour bersifat tetap, maka UCS dan BFS akan menghasilkan urutan simpul dan jalur yang sama.

Dalam praktiknya, A\* terbukti lebih efisien dibanding UCS karena memiliki arahan eksplorasi dari heuristik. A\* dapat memfokuskan pencarian pada jalur yang lebih menjanjikan, sehingga tidak perlu mengunjungi simpul-simpul yang kurang relevan. Pengujian menunjukkan bahwa A\* dapat menyelesaikan masalah dengan waktu lebih cepat dan simpul yang lebih sedikit selama heuristik yang digunakan sesuai. Hal ini menjadikan A\* unggul secara konsisten dalam menemukan solusi yang optimal dan efisien.

Sementara itu, Greedy Best First Search menunjukkan kelemahan yang cukup mencolok. Karena hanya mengandalkan heuristik tanpa mempertimbangkan jarak yang sudah ditempuh, GBFS cenderung terjebak pada jalur yang tampak dekat dengan tujuan, namun sebenarnya tidak efisien. Dari hasil pengujian, GBFS memang lebih cepat dalam beberapa kasus, tetapi solusi yang dihasilkan tidak selalu pendek atau optimal. Ini membuktikan bahwa kecepatan bukan satu-satunya ukuran keberhasilan dalam pencarian solusi.

Dari sisi kompleksitas, A\* memiliki kompleksitas waktu dan ruang sebesar  $O(b^d)$ , dengan  $b$  sebagai jumlah cabang rata-rata dan  $d$  sebagai kedalaman solusi optimal. Ini karena A\* menyimpan semua simpul yang dievaluasi untuk menjamin optimalitas. UCS juga memiliki kompleksitas  $O(b^d)$  karena tidak menggunakan heuristik dan mengevaluasi seluruh jalur yang mungkin. Meski demikian, UCS dapat menjadi sangat tidak efisien karena tidak memiliki arah dalam pencarian. GBFS juga memiliki kompleksitas waktu yang setara, yaitu  $O(b^d)$  dalam kasus terburuk, namun bisa bekerja lebih cepat dalam kasus tertentu. Sayangnya, karena tidak menjamin solusi terbaik, efisiensi GBFS lebih bergantung pada keberuntungan dan kualitas heuristik semata.



## **F. Kesimpulan**

Berdasarkan uraian diatas, dapat dilihat jikalau permainan Puzzle Rush Hour dapat diselesaikan dengan algoritma Uniform Cost Search (UCS), Greedy Best First Search, dan A\*. Berdasarkan hasil dari beberapa *test case* yang sudah dilakukan, algoritma A\* memiliki performa paling bagus dibandingkan dengan algoritma GBFS dan UCS. Secara umum, heuristik Blocking Cars juga memiliki hasil yang lebih baik daripada heuristik Manhattan Distance. Namun, itu kembali lagi sesuai *case board* awal, seperti pada test case handling board ukuran besar.

## G. Lampiran

Link Github : [https://github.com/Rejaah/Tucil3\\_13523068\\_13523119](https://github.com/Rejaah/Tucil3_13523068_13523119)

Tabel Ceklist:

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	✓	