

**LAPORAN TUGAS KECIL 3**  
**IF2211 STRATEGI ALGORITMA**

Penyelesaian Puzzle Rush Hour  
Menggunakan Algoritma Pathfinding



Disusun oleh :

Barru Adi Utomo	13523101
Azfa Radhiyya Hakim	13523115

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

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>2</b>
<b>BAB I.....</b>	<b>3</b>
<b>BAB II.....</b>	<b>5</b>
2.1 Greedy Best Search.....	5
2.2 Uniform Cost Search (UCS).....	5
2.3 Algoritma A-Star.....	5
2.4 Bidirectional Search.....	6
2.5 Graphical User Interface.....	6
<b>BAB III.....</b>	<b>7</b>
3.1 Penjelasan Algoritma.....	7
3.2 Heuristik yang digunakan.....	7
3.3 Implementasi Program.....	8
<b>BAB IV.....</b>	<b>14</b>
4.1 Tata Cara Penggunaan Program.....	14
4.2 Hasil Pengujian.....	14
<b>BAB V.....</b>	<b>20</b>
5.1 Analisis Pertanyaan.....	20
5.2 Analisis Algoritma.....	21
<b>BAB VI.....</b>	<b>22</b>
6.1 Kesimpulan.....	22
6.2 Saran.....	22
<b>LAMPIRAN.....</b>	<b>23</b>
Tautan Repository Github.....	23
Tabel Kelengkapan Spesifikasi.....	23
<b>DAFTAR PUSTAKA.....</b>	<b>24</b>

## BAB I

### DESKRIPSI TUGAS



**Gambar 1.** Rush Hour Puzzle

(Sumber: <https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

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

Komponen penting dari permainan Rush Hour terdiri dari:

1. Papan – Papan merupakan tempat permainan dimainkan.

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

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

2. Piece – Piece adalah sebuah kendaraan di dalam papan.

Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece. Secara standar, variasi ukuran sebuah piece adalah 2-piece (menempati 2 cell) atau 3-piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

3. Primary Piece – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.

4. Pintu Keluar – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan

5. Gerakan — Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

## **BAB II**

### **LANDASAN TEORI**

#### **2.1 Greedy Best Search**

Greedy Best First Search (GBFS) adalah algoritma pencarian yang menggunakan pendekatan heuristik untuk mencari simpul terbaik yang memiliki estimasi harga paling rendah menuju tujuan. Algoritma ini berfokus pada pengambilan keputusan yang didasarkan pada nilai heuristik dari simpul saat ini, tanpa mempertimbangkan informasi tentang biaya yang telah ditempuh sebelumnya. Fungsi estimasi tersebut adalah  $h(n)$ . Fungsi estimasi ini menjadi harga yang akan dipilih minimumnya berdasarkan simpul-simpul yang dapat dituju, sehingga terbentuk  $f(n) = h(n)$ , dengan  $f(n)$  adalah total harga sebuah simpul ekspansi dan  $h(n)$  adalah fungsi heuristik. GBFS akan memilih simpul hidup dengan  $f(n)$  minimum saat itu,  $f(n)$  saat minimum disebut local minima atau *plateau*.

#### **2.2 Uniform Cost Search (UCS)**

Algoritma Uniform Cost Search atau UCS adalah salah satu algoritma route planning atau penentuan rute tanpa adanya informasi tambahan mengenai tujuan pencarian (Uninformed Search/Blind Search). Pencarian dilakukan berdasarkan cost atau ongkos untuk sampai ke suatu simpul pada graf. Cost atau ongkos pada UCS biasanya dinotasikan sebagai  $g(n)$ , di mana  $n$  adalah simpul pada graf pencarian UCS. Pada pencarian cost minimum, simpul yang memiliki cost paling rendah hingga sampai ke suatu titik akan dibangkitkan terlebih dahulu. Pada proses penentuan cost, tidak diketahui informasi tambahan apapun mengenai tujuan pencarian, sehingga algoritma ini masuk ke dalam Uninformed Search. Pada algoritma ini, simpul yang memiliki cost paling rendah akan dibangkitkan terlebih dulu, maka implementasi dari algoritma ini biasanya menggunakan Priority Queue, di mana simpul paling yang memiliki prioritas paling tinggi akan berada di paling depan. Nilai prioritas dari algoritma ini sama dengan ongkos, sehingga digunakan persamaan sebagai berikut.  $f(n) = g(n)$ , di mana  $f(n)$  adalah fungsi evaluasi yang menjadi nilai prioritas suatu simpul dan  $g(n)$  adalah nilai cost atau ongkos suatu simpul.

#### **2.3 Algoritma A-Star**

A\* adalah algoritma pencarian yang menggabungkan pendekatan UCS dan GBFS dengan menggunakan fungsi biaya total  $f(n) = g(n) + h(n)$ , dengan  $g(n)$  adalah biaya kumulatif dari simpul awal ke simpul  $n$ , dan  $h(n)$  adalah nilai heuristik yang memperkirakan biaya dari simpul  $n$  ke tujuan.

## 2.4 Bidirectional Search

Bidirectional Search adalah algoritma graph traversal yang bekerja dengan cara menjalankan dua pencarian secara sekaligus, satu dimulai dari start node menuju goal node, dan yang satu lagi dimulai dari simpul tujuan menuju simpul awal. Pencarian ini dianggap selesai ketika kedua proses pencarian bertemu di satu simpul.

Ide utama di balik bidirectional search adalah mengurangi kompleksitas pencarian secara signifikan. Jika kita mengasumsikan branching factor dari graf adalah  $b$  dan jarak dari awal ke tujuan adalah  $d$ , maka pencarian satu arah seperti BFS atau DFS mungkin perlu menjelajahi sekitar  $bd$  simpul. Dengan pencarian dua arah, masing-masing pencarian idealnya hanya perlu menjelajah hingga kedalaman  $d/2$ . Oleh karena itu, total simpul yang dieksplorasi dapat berkurang menjadi  $2 * b^{(d/2)}$ , yang jauh lebih kecil daripada  $b^d$ , terutama untuk nilai  $d$  yang besar.

## 2.5 Graphical User Interface

Graphical User Interface (GUI) yang digunakan untuk visualisasi hasil menggunakan website. Arsitektur website yang dibangun memiliki pola client-server yang terdiri dari tiga komponen utama: frontend, backend, dan Java sebagai penyelesaian puzzle-nya. Bagian frontend menggunakan React.js, yang menyediakan tampilan visual, menerima input berupa file txt, serta menampilkan solusinya.

Backend menggunakan Node.js dengan framework Express sebagai jembatan antara antarmuka pengguna dan program Java solver. Setelah menerima file dari frontend, backend menyimpannya secara lokal, lalu menjalankan program Java dengan perintah terminal melalui modul `child_process.execSync`. Program Java yang berfungsi sebagai solver membaca file input tersebut, menjalankan algoritma penyelesaian, lalu menuliskan hasilnya ke dalam file dengan format JSON yang berisi urutan langkah-langkah penyelesaian puzzle. Setelah Java selesai menjalankan prosesnya, frontend akan mengambil file hasil tersebut dan menampilkannya setiap langkah.

## BAB III

### IMPLEMENTASI ALGORITMA

#### 3.1 Penjelasan Algoritma

Langkah yang dilakukan untuk ketiga algoritma secara garis besar sama, namun terdapat perbedaan dalam mengevaluasi nilai kondisi board dalam suatu state tertentu. Langkah pertama dalam setiap algoritma adalah menginisialisasi node awal dari kondisi papan permainan. Di tahap ini, node awal akan diberikan nilai evaluasi tertentu tergantung jenis algoritma yang digunakan, kemudian dimasukkan ke dalam priority queue. Selanjutnya, algoritma akan terus mengambil node dengan nilai prioritas tertinggi (dalam hal ini, nilai terkecil), dan mengembangkannya menjadi semua kemungkinan gerakan (successor states). Setiap hasil gerakan akan diperiksa: jika belum pernah dikunjungi atau memiliki jalur yang lebih murah, maka node tersebut akan diperbarui dan dimasukkan ke dalam antrian. Proses ini akan terus berlanjut hingga ditemukan kondisi akhir yang memenuhi kriteria sebagai solusi.

Perbedaan utama dari ketiga algoritma ini terletak pada cara mereka menghitung nilai prioritas untuk tiap node. Pada Uniform Cost Search (UCS), nilai prioritas dihitung hanya berdasarkan  $g(n)$ , yaitu total biaya dari titik awal menuju node saat ini. UCS tidak menggunakan estimasi ke tujuan, sehingga cocok untuk mencari solusi dengan jalur termurah tanpa memperkirakan jarak ke titik akhir.

Sementara itu, pada Greedy Best-First Search (GBFS), nilai prioritas hanya dihitung dari  $h(n)$ , yaitu estimasi jarak dari node saat ini ke tujuan berdasarkan fungsi heuristik. GBFS tidak mempertimbangkan biaya perjalanan dari titik awal, sehingga bisa lebih cepat namun tidak menjamin solusi optimal.

Berbeda dari keduanya, A Search\* mengkombinasikan kekuatan UCS dan GBFS dengan menggunakan fungsi evaluasi  $f(n) = g(n) + h(n)$ . Artinya, A\* mempertimbangkan baik biaya yang telah ditempuh ( $g(n)$ ) maupun estimasi sisa jarak ke tujuan ( $h(n)$ ), sehingga menghasilkan pencarian yang efisien dan tetap optimal asalkan heuristik yang digunakan admissible (tidak melebihi nilai sebenarnya).

#### 3.2 Heuristik yang digunakan

Berikut adalah dua heuristik yang digunakan.

- a. **Heuristik 1:** Mencari jarak antara primary key dengan pintu keluar

Heuristik pertama menghitung jarak mobil utama (*primary block*) menuju posisi keluar. Heuristik ini menghitung jarak lurus antara ujung mobil utama dengan titik keluar berdasarkan orientasi dan posisi mobil pada papan permainan. Heuristik ini dapat digunakan dalam algoritma seperti A\* sebagai estimasi dasar seberapa

dekat suatu kondisi dengan solusi, meskipun tidak selalu mencerminkan realita sepenuhnya jika banyak kendaraan menghalangi jalur.

- b. **Heuristik 2:** Mencari banyak mobil yang menghalangi primary key relatif terhadap pintu keluar.

Heuristik kedua menghitung jumlah blok yang menghalangi jalur keluar dari mobil utama (primary block), sehingga dapat memberikan estimasi seberapa sulit bagi primary block untuk mencapai. Semakin banyak penghalang, maka semakin besar nilai heuristik yang menunjukkan bahwa kondisi tersebut lebih jauh dari solusi. Fungsi ini dapat digunakan dalam algoritma pencarian seperti A\*, karena memberikan estimasi biaya tambahan yang diperlukan untuk mencapai solusi. Dengan demikian, heuristik ini membantu algoritma fokus pada simpul-simpul yang lebih menjanjikan untuk diperluas lebih awal.

### 3.3 Implementasi Program

#### Algoritma AStar

```
public class AStar {

    public static List<Board> AStarAlgorithm(Board initialBoard) {
        PriorityQueue<BoardNode> openSet = new PriorityQueue<>();
        Set<String> closedSet = new HashSet<>();
        Map<String, BoardNode> allNodes = new HashMap<>();

        BoardNode startNode = new BoardNode(initialBoard, null, 0,
        Heuristic.calculateHeuristic(initialBoard));

        openSet.add(startNode);
        allNodes.put(BoardNode.boardToString(initialBoard), startNode);

        while (!openSet.isEmpty()) {
            BoardNode current = openSet.poll();

            if (current.board.isFinish()) {
                System.out.println("Found a solution!");
                return reconstructPath(current);
            }

            String currentBoardString =
            BoardNode.boardToString(current.board);
            closedSet.add(currentBoardString);

            List<Board> possibleMoves =
            current.board.getAllPossibleMove();

            for (Board nextBoard : possibleMoves) {
                nextBoard.updateBoardData();
                String nextBoardString =
```



```

BoardNode.boardToString(nextBoard);

        if (closedSet.contains(nextBoardString)) {
            continue;
        }
        int tentativeGScore = current.gScore + 1;

        BoardNode nextNode = allNodes.get(nextBoardString);
        boolean isNewNode = (nextNode == null);

        if (isNewNode || tentativeGScore < nextNode.gScore) {
            if (isNewNode) {
                nextNode = new BoardNode(
                    nextBoard,
                    current,
                    tentativeGScore,
                    Heuristic.calculateHeuristic(nextBoard)
                );
                allNodes.put(nextBoardString, nextNode);
                // System.out.println("Adding new node: " +
nextBoardString);
            } else {
                nextNode.parent = current;
                nextNode.gScore = tentativeGScore;
                nextNode.fScore = tentativeGScore +
nextNode.hScore;

                openSet.remove(nextNode);
            }

            openSet.add(nextNode);
        }
    }
    return new ArrayList<>();
}
}

```

## Algoritma UCS

```

public class UCS {

    public static List<Board> UCSAlgorithm(Board initialBoard) {
        PriorityQueue<BoardNode> openSet = new PriorityQueue<>();
        Set<String> closedSet = new HashSet<>();
        Map<String, BoardNode> allNodes = new HashMap<>();

        BoardNode startNode = new BoardNode(initialBoard, null, 0, 0); //
h = 0
        openSet.add(startNode);
        allNodes.put(BoardNode.boardToString(initialBoard), startNode);

        while (!openSet.isEmpty()) {
            BoardNode current = openSet.poll();

```

```

        if (current.board.isFinish()) {
            return reconstructPath(current);
        }

        String currentBoardString =
BoardNode.boardToString(current.board);
        closedSet.add(currentBoardString);

        for (Board nextBoard : current.board.getAllPossibleMove()) {
            nextBoard.updateBoardData();
            String nextBoardString =
BoardNode.boardToString(nextBoard);

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

            int tentativeGScore = current.gScore + 1;
            BoardNode nextNode = allNodes.get(nextBoardString);
            boolean isNewNode = (nextNode == null);

            if (isNewNode || tentativeGScore < nextNode.gScore) {
                if (isNewNode) {
                    nextNode = new BoardNode(nextBoard, current,
tentativeGScore, 0);
                    allNodes.put(nextBoardString, nextNode);
                } else {
                    nextNode.parent = current;
                    nextNode.gScore = tentativeGScore;
                    nextNode.fScore = tentativeGScore;
                    openSet.remove(nextNode);
                }
                openSet.add(nextNode);
            }
        }

        return new ArrayList<>();
    }
}

```

### Algoritma Greedy Best First Search

```

public class GreedyBestFirstSearch {

    public static List<Board> GreedyAlgorithm(Board initialBoard) {
        PriorityQueue<BoardNode> openSet = new PriorityQueue<>();
        Set<String> closedSet = new HashSet<>();
        Map<String, BoardNode> allNodes = new HashMap<>();

        int h = Heuristic.calculateHeuristic(initialBoard);
        BoardNode startNode = new BoardNode(initialBoard, null, 0, h);
        openSet.add(startNode);
    }
}

```

```

        allNodes.put(BoardNode.boardToString(initialBoard), startNode);

        while (!openSet.isEmpty()) {
            BoardNode current = openSet.poll();

            if (current.board.isFinish()) {
                return reconstructPath(current);
            }

            String currentBoardString =
BoardNode.boardToString(current.board);
            closedSet.add(currentBoardString);

            for (Board nextBoard : current.board.getAllPossibleMove()) {
                nextBoard.updateBoardData();
                String nextBoardString =
BoardNode.boardToString(nextBoard);

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

                int hScore = Heuristic.calculateHeuristic(nextBoard);
                BoardNode nextNode = allNodes.get(nextBoardString);
                boolean isNewNode = (nextNode == null);

                if (isNewNode || hScore < nextNode.hScore) {
                    if (isNewNode) {
                        nextNode = new BoardNode(nextBoard, current, 0,
hScore);

                        allNodes.put(nextBoardString, nextNode);
                    } else {
                        nextNode.parent = current;
                        nextNode.hScore = hScore;
                        nextNode.fScore = hScore;
                        openSet.remove(nextNode);
                    }
                    openSet.add(nextNode);
                }
            }
        }

        return new ArrayList<>();
    }
}

```

### Algoritma Bidirectional

```

public class BidirectionalSolver {

    public List<Board> solve(Board startBoard) {
        Queue<Board> forwardQueue = new LinkedList<>();
        Queue<Board> backwardQueue = new LinkedList<>();

        Map<String, BoardNode> forwardVisited = new HashMap<>();
    }
}

```

```

        Map<String, BoardNode> backwardVisited = new HashMap<>();

        forwardQueue.add(startBoard);
        forwardVisited.put(BoardNode.boardToString(startBoard), new
BoardNode(startBoard, null, 0, 0));

        Board goalBoard = generateGoalBoard(startBoard);
        if (goalBoard == null) return null;

        backwardQueue.add(goalBoard);
        backwardVisited.put(BoardNode.boardToString(goalBoard), new
BoardNode(goalBoard, null, 0, 0));

        while (!forwardQueue.isEmpty() && !backwardQueue.isEmpty()) {
            Board fCurrent = forwardQueue.poll();
            String fKey = BoardNode.boardToString(fCurrent);

            for (Board next : fCurrent.getAllPossibleMove()) {
                String nextKey = BoardNode.boardToString(next);
                if (forwardVisited.containsKey(nextKey)) continue;

                BoardNode nextNode = new BoardNode(next,
forwardVisited.get(fKey), 0, 0);
                forwardVisited.put(nextKey, nextNode);
                forwardQueue.add(next);

                if (backwardVisited.containsKey(nextKey)) {
                    return mergePath(nextNode,
backwardVisited.get(nextKey));
                }
            }

            Board bCurrent = backwardQueue.poll();
            String bKey = BoardNode.boardToString(bCurrent);

            for (Board next : bCurrent.getAllPossibleMove()) {
                String nextKey = BoardNode.boardToString(next);
                if (backwardVisited.containsKey(nextKey)) continue;

                BoardNode nextNode = new BoardNode(next,
backwardVisited.get(bKey), 0, 0);
                backwardVisited.put(nextKey, nextNode);
                backwardQueue.add(next);

                if (forwardVisited.containsKey(nextKey)) {
                    return mergePath(forwardVisited.get(nextKey),
nextNode);
                }
            }
        }

        return null;
    }

    private List<Board> mergePath(BoardNode forwardNode, BoardNode
backwardNode) {
        LinkedList<Board> path = new LinkedList<>();
        BoardNode curr = forwardNode;
        while (curr != null) {

```

```

        path.addFirst(curr.board);
        curr = curr.parent;
    }

    curr = backwardNode.parent;
    while (curr != null) {
        path.addLast(curr.board);
        curr = curr.parent;
    }

    return path;
}

private Board generateGoalBoard(Board from) {
    Board copy = from.clone();
    copy.setKiriAtas(from.isKiriAtas());
    Block primary = copy.getPrimaryBlock();

    if (primary == null) return null;

    Block goalPrimary;
    if (primary.isBlockVertical()) {
        int goalRow = from.isKiriAtas() ? 0 : from.getHeight() -
primary.getBlockSize();
        goalPrimary = primary.makeBlockNewPosition(goalRow,
primary.getBlockColIndex());
    } else {
        int goalCol = from.isKiriAtas() ? 0 : from.getWidth() -
primary.getBlockSize();
        goalPrimary =
primary.makeBlockNewPosition(primary.getBlockRowIndex(), goalCol);
    }

    copy.updateBlock(goalPrimary);
    return copy;
}
}

```

## BAB IV PENGUJIAN

### 4.1 Tata Cara Penggunaan Program

- a. *Clone repository Github*

```
git clone https://github.com/barruadi/Tucil3_13523101_13523115
```

- b. Mengunduh *dependencies* yang dibutuhkan

```
cd src/frontend  
npm install
```

- c. Menjalankan program

```
npm start
```

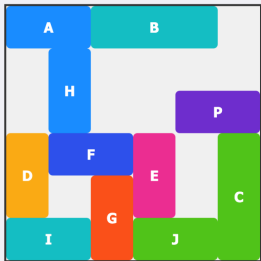
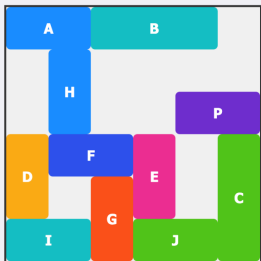
### 4.2 Hasil Pengujian

Tabel 1 Hasil pengujian testcase 1

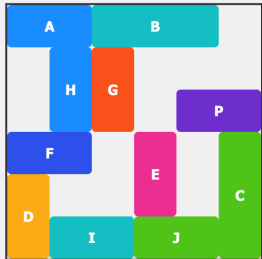
Testcase 1	
6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	
algoritma	Hasil

A Star	<div><div>Tucil3 - RushHour</div><div></div><div>Step: 6 / 6</div><div><div>Play</div><div>Previous Move</div><div>Next Move</div></div></div> <div><div>Upload Puzzle File:</div><div>Choose File tes.txt</div><div>Algorithm:</div><div>A*</div><div>Heuristic:</div><div>Menghitung jarak</div><div>Start</div></div>
--------	---

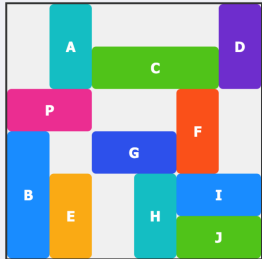
Tabel 2 Hasil pengujian testcase 2

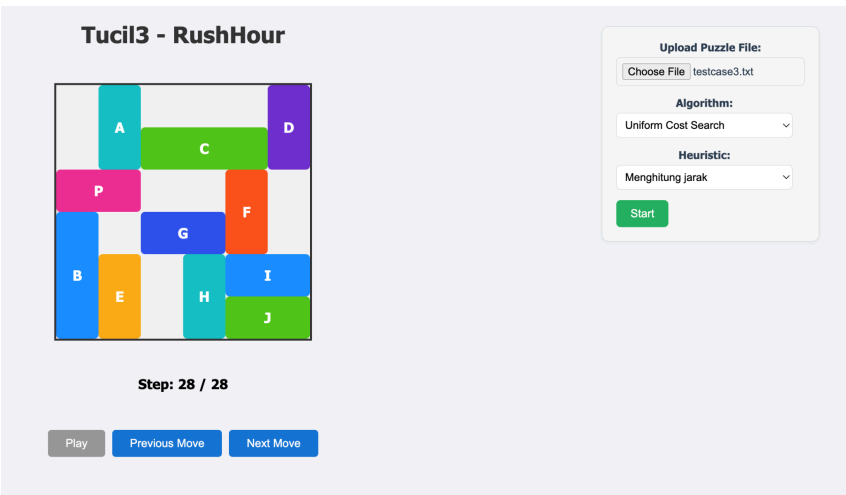
Testcase 2	
<div>6 6 10 AABBBC D..E.C DPPE.CK FFG... .HG... .HIIJJ</div>	
algoritma	Hasil
A Star	<div><div><div>Tucil3 - RushHour</div><div></div><div>Step: 24 / 24</div><div><div>Play</div><div>Previous Move</div><div>Next Move</div></div></div><div><div>Upload Puzzle File:</div><div><div>Choose File</div>testcase2.txt</div><div>Algorithm:</div><div>A*</div><div>Heuristic:</div><div>Menghitung jarak</div><div>Start</div></div></div>
UCS	<div><div><div>Tucil3 - RushHour</div><div></div><div>Step: 24 / 24</div><div><div>Play</div><div>Previous Move</div><div>Next Move</div></div></div><div><div>Upload Puzzle File:</div><div><div>Choose File</div>testcase2.txt</div><div>Algorithm:</div><div>Uniform Cost Search</div><div>Heuristic:</div><div>Menghitung jarak</div><div>Start</div></div></div>



Greedy Best First Search	<div> <div> <div>Tucil3 - RushHour</div>  <div>Step: 223 / 223</div> <div> <div>Play</div> <div>Previous Move</div> <div>Next Move</div> </div> </div> </div>
--------------------------	--

Tabel 3 Hasil pengujian testcase 3

Testcase 3	
<pre> 6 6 10 BA.... BA.CCC KBE.PPD .EGGFD .IIHF. .JJH.. </pre>	
algoritma	Hasil
A Star	<div> <div> <div>Tucil3 - RushHour</div>  <div>Step: 28 / 28</div> <div> <div>Play</div> <div>Previous Move</div> <div>Next Move</div> </div> </div> </div>

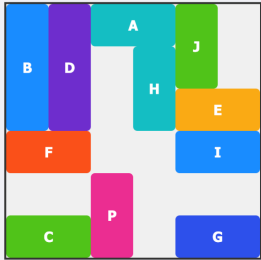
UCS	
Greedy Best First Search	

Tabel 4 Hasil pengujian testcase 4

Testcase 4	
<pre> 6 6 10 AAPH.. BDPH.. BDEE.. BDFFI ....J. CCGGJ. K </pre>	
algoritma	Hasil

A Star

**Tucil3 - RushHour**



Step: 35 / 35

Play Previous Move Next Move

Upload Puzzle File:  
Choose File testcase4.txt

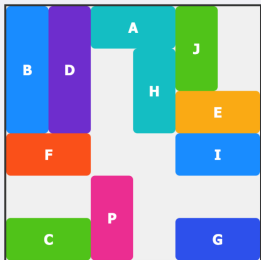
Algorithm:  
A\*

Heuristic:  
Menghitung jarak

Start

UCS

**Tucil3 - RushHour**



Step: 35 / 35

Play Previous Move Next Move

Upload Puzzle File:  
Choose File testcase4.txt

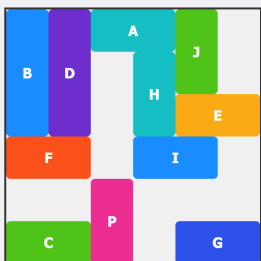
Algorithm:  
Uniform Cost Search

Heuristic:  
Menghitung jarak

Start

Greedy Best First Search

**Tucil3 - RushHour**



Step: 147 / 147

Play Previous Move Next Move

Upload Puzzle File:  
Choose File testcase4.txt

Algorithm:  
Greedy Best First Search

Heuristic:  
Menghitung jarak

Start

## BAB V

### ANALISIS

#### 5.1 Analisis Pertanyaan

a. Definisi dari  $f(n)$  dan  $g(n)$

Dalam algoritma A\*, fungsi evaluasi ditulis sebagai  $f(n) = g(n) + h(n)$ . Nilai  $g(n)$  merepresentasikan biaya riil atau jumlah langkah dari node awal hingga node  $n$  saat ini. Nilai  $h(n)$  adalah heuristik, yaitu estimasi biaya minimum dari node  $n$  ke goal. Oleh karena itu,  $f(n)$  adalah estimasi total biaya dari start hingga goal jika melalui node  $n$ . Pada UCS, karena tidak menggunakan heuristik,  $f(n) = g(n)$ . Sedangkan pada GBFS, hanya  $h(n)$  yang digunakan tanpa memperhatikan  $g(n)$ , sehingga  $f(n) = h(n)$ .

b. Apakah heuristik yang digunakan pada algoritma A\* admissible?

Ya, heuristik yang digunakan dalam algoritma A\* dapat dianggap *admissible*, selama ia tidak melebihi-lebihkan biaya aktual ke goal. Dalam implementasi Rush Hour, dua heuristik yang digunakan adalah:

- Jarak lurus dari ujung mobil utama ke posisi goal.
- Jumlah kendaraan yang menghalangi jalur keluar mobil utama.

Heuristik pertama adalah *admissible* karena merupakan estimasi paling dasar dan tidak pernah lebih besar dari jumlah langkah aktual yang dibutuhkan. Heuristik kedua juga *admissible* apabila dianggap setiap penghalang memerlukan minimal satu langkah untuk disingkirkan, tanpa mengasumsikan langkah tambahan atau kompleksitas ekstra. Sesuai definisi di salindia kuliah, heuristik *admissible* adalah heuristik yang tidak pernah overestimate biaya ke goal.

c. Apakah UCS sama dengan BFS dalam konteks Rush Hour?

Dalam konteks puzzle Rush Hour, di mana setiap aksi (langkah kendaraan) memiliki biaya yang sama, maka algoritma UCS secara praktis akan berperilaku sama seperti BFS. Keduanya akan mengeksplorasi simpul berdasarkan jumlah langkah (depth), tanpa mempertimbangkan arah solusi. Artinya, urutan node yang dibangkitkan dan path yang dihasilkan oleh UCS dan BFS akan identik dalam kasus ini. Perbedaan di antara keduanya hanya akan tampak bila langkah memiliki bobot berbeda.

d. Apakah A\* lebih efisien dibanding UCS untuk Rush Hour (secara teoritis)?

Secara teoritis, ya. A\* lebih efisien dibanding UCS karena A\* menggunakan heuristik yang mengarahkan pencarian ke arah tujuan. UCS memperluas simpul hanya berdasarkan path cost (jumlah langkah dari start), sehingga banyak node yang diperluas secara "buta". Sebaliknya, A\* memperhitungkan juga estimasi ke goal, yang membantu mempersempit ruang pencarian. Dengan heuristik yang *admissible* dan cukup informatif, A\* dapat mencapai solusi lebih cepat dengan lebih sedikit node yang dibangkitkan, terutama dalam konfigurasi papan yang lebih kompleks.

e. Apakah Greedy Best First Search menjamin solusi optimal untuk Rush Hour?

Tidak. GBFS hanya mempertimbangkan  $h(n)$  dan mengabaikan  $g(n)$ . Artinya, ia hanya melihat seberapa "dekat" suatu simpul ke goal berdasarkan heuristik, tanpa memperhatikan biaya nyata yang sudah dikeluarkan. Hal ini membuat GBFS sering memilih simpul yang tampak dekat ke goal tapi sebenarnya memerlukan lebih banyak langkah secara keseluruhan. Oleh karena itu, Greedy BFS tidak menjamin solusi optimal, dan meskipun sering lebih cepat, ia rentan memberikan solusi yang lebih panjang atau bahkan terjebak pada jalur yang salah.

## 5.2 Analisis Algoritma

GBFS menggunakan fungsi heuristik  $h(n)$  untuk memilih simpul yang tampak paling dekat ke tujuan. Karena hanya memperhatikan estimasi jarak ke goal dan mengabaikan biaya yang sudah ditempuh, algoritma ini memiliki kompleksitas waktu yang lebih efisien dalam banyak kasus, tetapi rentan terhadap jebakan lokal minimum. GBFS sangat cepat jika heuristiknya baik, tetapi bisa gagal menemukan solusi optimal. Kompleksitas waktunya mendekati  $O(b^m)$ , di mana  $b$  adalah branching factor dan  $m$  adalah kedalaman solusi terbaik yang ditemukan berdasarkan estimasi heuristik.

UCS mengeksplorasi simpul berdasarkan biaya kumulatif  $g(n)$  tanpa memperhitungkan estimasi ke tujuan. Karena mempertimbangkan seluruh jalur, UCS menjamin menemukan solusi optimal, namun cenderung lebih lambat dibanding GBFS terutama pada graf besar. Kompleksitas waktunya juga  $O(b^d)$ , dengan  $d$  adalah kedalaman solusi optimal, namun eksplorasi bisa menjadi sangat luas karena tidak memiliki arah menuju goal secara heuristik.

A\* menggabungkan kelebihan GBFS dan UCS dengan fungsi  $f(n) = g(n) + h(n)$ . A\* sangat efisien dan juga optimal jika heuristik yang digunakan adalah admissible dan consistent. Dalam praktik, A\* menunjukkan performa terbaik di antara algoritma lain jika heuristik dirancang dengan baik. Kompleksitas waktu A\* juga  $O(b^d)$  di kasus terburuk, tetapi dengan heuristik yang baik, jumlah simpul yang dieksplorasi bisa jauh lebih sedikit dibanding UCS.

Bidirectional search menjalankan dua pencarian dari arah start dan goal secara bersamaan. Jika struktur graf memungkinkan, algoritma ini secara signifikan mengurangi ruang dan waktu pencarian, karena kedalaman pencarian dari kedua arah menjadi  $d/2$ . Kompleksitas waktunya dapat turun menjadi  $O(b^{(d/2)})$ , menjadikannya sangat efisien dibanding pencarian satu arah. Namun, algoritma ini lebih kompleks dalam implementasi, terutama dalam menentukan titik pertemuan dan menyinkronkan dua pencarian.

## **BAB VI**

### **KESIMPULAN, SARAN, DAN REFLEKSI**

#### **6.1 Kesimpulan**

Tucil ini berhasil menerapkan algoritma pathfinding dalam pemecahan puzzle Rush Hour, yang terdiri dari tiga komponen utama: antarmuka pengguna (React.js), server backend (Node.js + Express), dan logic (Java). Program memungkinkan pengguna mengunggah file konfigurasi puzzle, memilih algoritma pencarian yang diinginkan (UCS, Greedy Best First Search, A\*, atau Bidirectional Search), serta menampilkan langkah-langkah solusi secara interaktif dan visual. Serta terdapat dua jenis heuristik yang diterapkan, yaitu jarak block primary ke exit dan banyaknya block yang menghalangi block primary

Melalui pengujian berbagai konfigurasi puzzle, diperoleh bahwa A\* dengan heuristik yang admissible dapat secara signifikan mengurangi jumlah node yang dibangkitkan dibanding UCS dan BFS, tanpa mengorbankan optimalitas. Sementara itu, GBFS terbukti lebih cepat namun tidak selalu menghasilkan solusi terpendek. Sistem ini mampu memvisualisasikan proses penyelesaian dengan baik, memberikan pemahaman lebih intuitif tentang bagaimana algoritma bekerja dalam menyelesaikan permasalahan pencarian jalur.

#### **6.2 Saran**

Saran untuk pengembangan lebih lanjut adalah dengan:

- Membuat program lebih modular dan lebih terstruktur, sehingga dapat dikembangkan jika ingin menerapkan algoritma yang lain.
- Membuat antarmuka yang lebih interaktif.
- Memperlihatkan riwayat pencarian solusi.

## LAMPIRAN

### Tautan Repository Github

[https://github.com/barruadi/Tucil3\\_13523101\\_13523115](https://github.com/barruadi/Tucil3_13523101_13523115)

### Tabel Kelengkapan Spesifikasi

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	V	
2. Program berhasil dijalankan	V	
3. Solusi yang diberikan program benar dan mematuhi aturan permainan	V	
4. Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	V	
5. [Bonus] Implementasi algoritma pathfinding alternatif	V	
6. [Bonus] Implementasi 2 atau lebih heuristik alternatif	V	
7. [Bonus] Program memiliki GUI	V	
8. Program dan laporan dibuat (kelompok) sendiri	V	

## DAFTAR PUSTAKA

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Makalah/Makalah-Stima-2023-\(47\).pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Makalah/Makalah-Stima-2023-(47).pdf)

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Makalah/Makalah-IF2211-Stima-2024%20\(19\).pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Makalah/Makalah-IF2211-Stima-2024%20(19).pdf)

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Makalah/Makalah-IF2211-Stima-2024%20\(25\).pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Makalah/Makalah-IF2211-Stima-2024%20(25).pdf)