

IF2211 Strategi Algoritma

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

### **Laporan Tugas Kecil 3**

Disusun untuk memenuhi tugas mata kuliah Strategi Algoritma pada Semester 4 (empat)

Tahun Akademik 2024/2025



Disusun Oleh:

**Hasri Fayadh Muqaffa (13523156)**

**Filbert Engyo (13523163)**

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

**BANDUNG 2025**

# Daftar Isi

<b>BAB I</b>	
<b>DESKRIPSI MASALAH.....</b>	<b>5</b>
<b>BAB II</b>	
<b>DASAR TEORI.....</b>	<b>9</b>
2.1 Uniform Cost Search.....	9
2.2 Greedy Best First Search.....	10
2.3 A* Search.....	10
2.4. Iterative Deepening A* .....	11
<b>BAB III</b>	
<b>ANALISIS ALGORITMA.....</b>	<b>12</b>
3.1 Definisi f(n) dan g(n).....	12
3.2 Admissibility Heuristik A* pada Puzzle Rush Hour.....	12
3.3 Kesamaan UCS dan BFS pada Penyelesaian Puzzle Rush Hour.....	12
3.4 Keefisienan A* dibandingkan UCS dalam Rush Hour.....	12
3.5 Apakah GBFS menjamin solusi optimal pada Rush Hour?.....	13
<b>BAB IV</b>	
<b>IMPLEMENTASI PROGRAM.....</b>	<b>14</b>
4.1 Model.....	15
4.1.1 Board.....	15
4.1.2 Node.....	17
4.1.3 Piece.....	19
4.2 Algoritma.....	19
4.2.1 Uniform Cost Search (UCS).....	19
4.2.2 GBFS.....	21
4.2.3 A*.....	24
4.2.4 IDA*.....	26
<b>BAB V</b>	
<b>PENGUJIAN.....</b>	<b>28</b>
5.1. UCS.....	29
5.2. GBFS.....	30
5.2.1. Manhattan Distance Heuristic.....	30
5.2.2. Obstacle Heuristic.....	31
5.3 A*.....	32
5.3.1 Manhattan Distance Heuristic.....	32
5.3.2 Obstacle Heuristic.....	34
5.3.3 Combined Heuristic.....	35
5.4 IDA* .....	36

5.4.1 Manhattan Distance Heuristic.....	36
5.4.2 Obstacle Heuristic.....	37
5.4.3 Combined Heuristic.....	38
<b>BAB VI</b>	
<b>KESIMPULAN &amp; SARAN.....</b>	<b>39</b>
6.1 Kesimpulan.....	39
6.2 Saran.....	39
<b>LAMPIRAN.....</b>	<b>40</b>
<b>DAFTAR PUSTAKA.....</b>	<b>41</b>

## Daftar Gambar

Gambar 2.1 Penyelesaiannya Rute dengan UCS.....	9
Gambar 2.2 Contoh Penyelesaian Rute dengan GBFS.....	10
Gambar 2.3 Contoh Penyelesaian Puzzle 8 dengan A*.....	11
Gambar 4.1.1.1 Struktur data board.....	18
Gambar 4.1.1.2 Metode Kelas Board.....	19
Gambar 4.1.2.1 Struktur Data Node.....	20
Gambar 4.1.2.2 Metode Kelas Node.....	21
Gambar 4.1.3 Struktur Data & Konstruktor Piece.....	22
Gambar 4.2.1. Implementasi Fungsi solve.....	23
Gambar 4.2.2.1 Fungsi heuristicManhattan.....	24
Gambar 4.2.2.2 Fungsi heuristicBlockers.....	25
Gambar 4.2.2.3 Fungsi calculateHeuristic.....	25
Gambar 4.2.2.4 Implementasi Fungsi solve.....	26
Gambar 4.2.3.1 Fungsi Kalkulasi Heuristik.....	28
Gambar 4.2.3.2 Implementasi Fungsi solve.....	28

## BAB I

### DESKRIPSI MASALAH



**Gambar 1. Rush Hour Puzzle**

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

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

Komponen penting dari permainan Rush Hour terdiri dari:

1. **Papan** – Papan merupakan tempat permainan dimainkan.

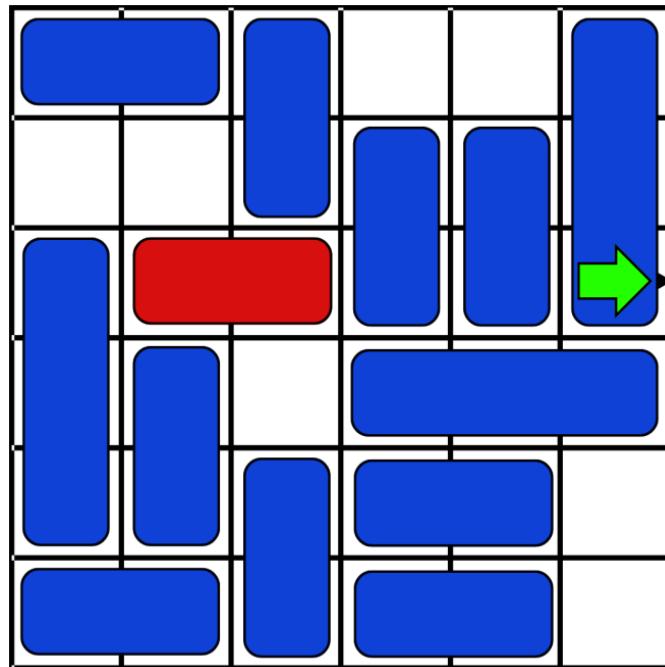
Papan terdiri atas cell, yaitu sebuah singular point dari papan. Sebuah piece akan menempati cell-cell pada papan. Ketika permainan dimulai, semua piece telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi piece dan orientasi, antara horizontal atau vertikal.

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

2. **Piece** – Piece adalah sebuah kendaraan di dalam papan. Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal—tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece. Secara standar, variasi ukuran sebuah piece adalah 2-piece (menempati 2 cell) atau 3-piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.
3. **Primary Piece** – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.
4. **Pintu Keluar** – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan
5. **Gerakan** — Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

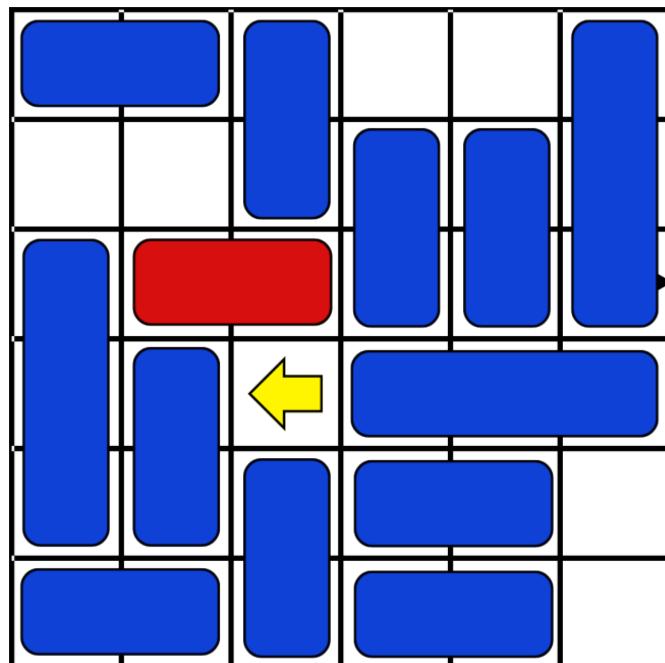
#### **Ilustrasi kasus :**

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

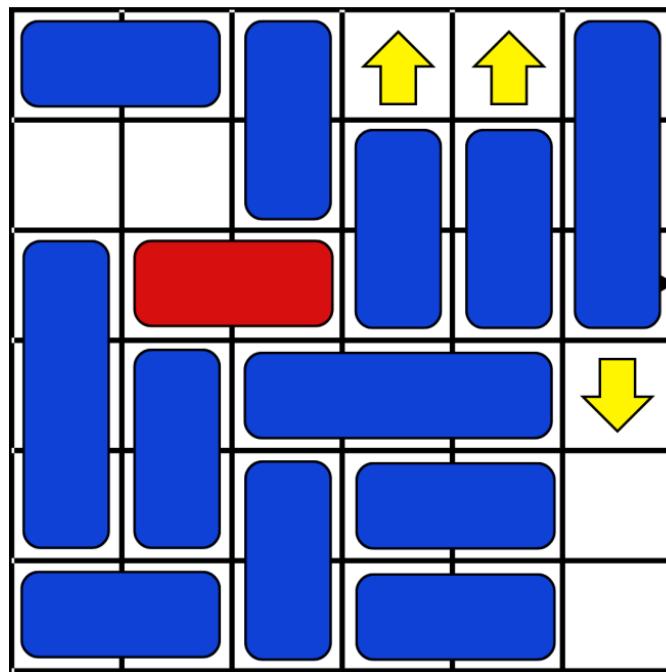


Gambar 2. Awal Permainan Game Rush Hour

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

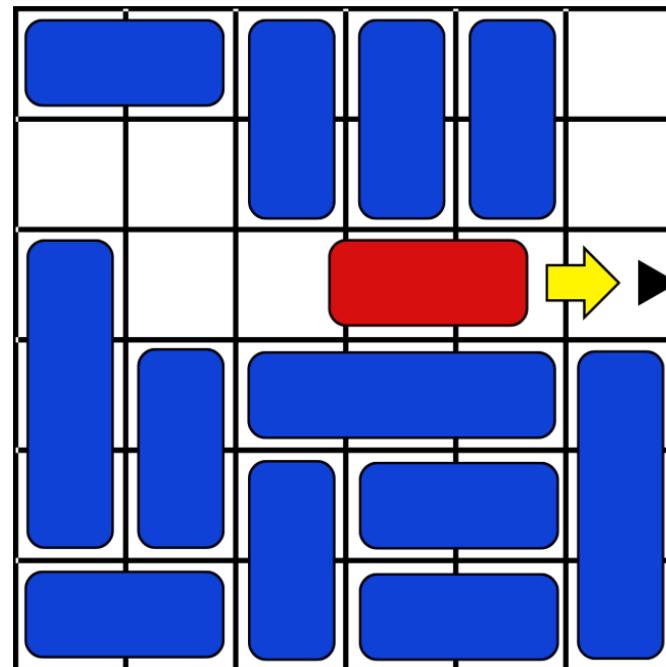


Gambar 3. Gerakan Pertama Game Rush Hour



Gambar 4. Gerakan Kedua Game Rush Hour

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



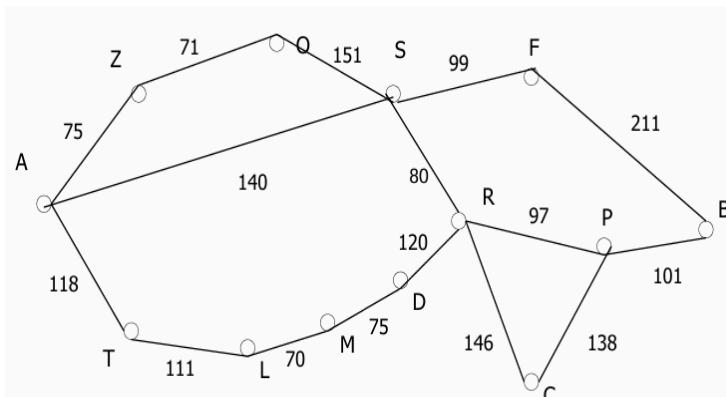
Gambar 5. Pemain Menyelesaikan Permainan

## BAB II

# DASAR TEORI

### 2.1 Uniform Cost Search

Uniform Cost Search (UCS) adalah algoritma pencarian yang bertujuan menemukan jalur dengan biaya terendah dari titik awal ke tujuan dalam graf atau tree, dengan cara secara sistematis memperluas node yang memiliki total biaya kumulatif terkecil dari awal. Algoritma ini menggunakan antrian prioritas untuk mengelola node yang akan dieksplorasi, selalu memilih node dengan biaya terendah untuk diperluas selanjutnya, dan jika menemukan beberapa jalur ke node yang sama, ia hanya mempertahankan jalur dengan biaya terendah. Proses ini menjamin bahwa ketika node tujuan ditemukan, jalur yang dihasilkan adalah solusi optimal dengan biaya minimum.



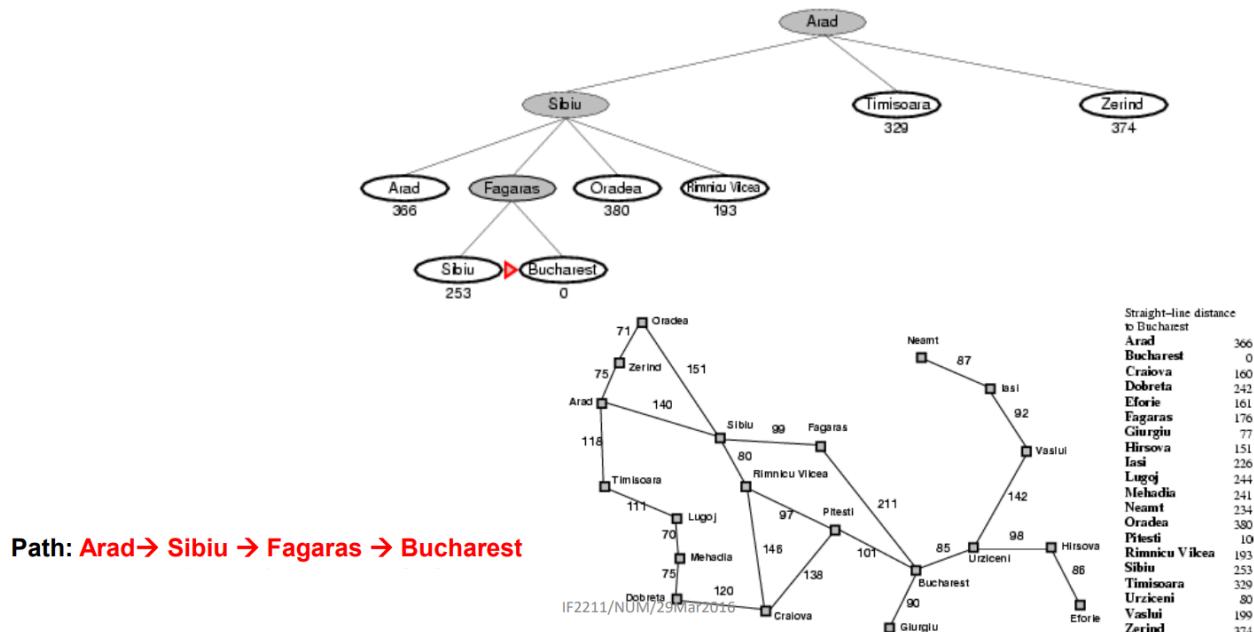
A	$Z_{A-75}, T_{A-118}, S_{A-140}$
$Z_{A-75}$	$T_{A-118}, S_{A-140}, O_{AZ-146}$
$T_{A-118}$	$S_{A-140}, O_{AZ-146}, L_{AT-229}$
$S_{A-140}$	$O_{AZ-146}, R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}$
$O_{AZ-146}$	$R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}$
$R_{AS-220}$	$L_{AT-229}, F_{AS-239}, O_{AS-291}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}$
$L_{AT-229}$	$F_{AS-239}, O_{AS-291}, M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}$
$F_{AS-239}$	$O_{AS-291}, M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}, B_{ASF-450}$
$O_{AS-291}$	$M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}, B_{ASF-450}$
$M_{ATL-299}$	$P_{ASR-317}, D_{ASR-340}, D_{ATLM-364}, C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$P_{ASR-317}$	$D_{ASR-340}, D_{ATLM-364}, C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$D_{ASR-340}$	$D_{ATLM-364}, C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$D_{ATLM-364}$	$C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$C_{ASR-366}$	$B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$B_{ASRP-418}$	Solusi ketemu

Gambar 2.1 Penyelesaiannya Rute dengan UCS

(Sumber : [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf))

## 2.2 Greedy Best First Search

Greedy Best-First Search (GBFS) adalah algoritma pencarian heuristik yang bersifat "rakus" karena memprioritaskan ekspansi node semata-mata berdasarkan perkiraan jarak terdekat ke tujuan menggunakan fungsi heuristik  $h(n)$ , tanpa memperhitungkan biaya akumulatif dari node awal. Algoritma ini menggunakan antrian prioritas yang diurutkan berdasarkan nilai  $h(n)$  untuk selalu memilih node yang tampak paling menjanjikan (nilai  $h(n)$  terkecil) untuk dieksplorasi selanjutnya, sehingga efektif menemukan solusi dengan cepat meskipun tidak menjamin optimalitas, dan akan berhenti ketika node tujuan ditemukan atau semua kemungkinan telah dieksplorasi.



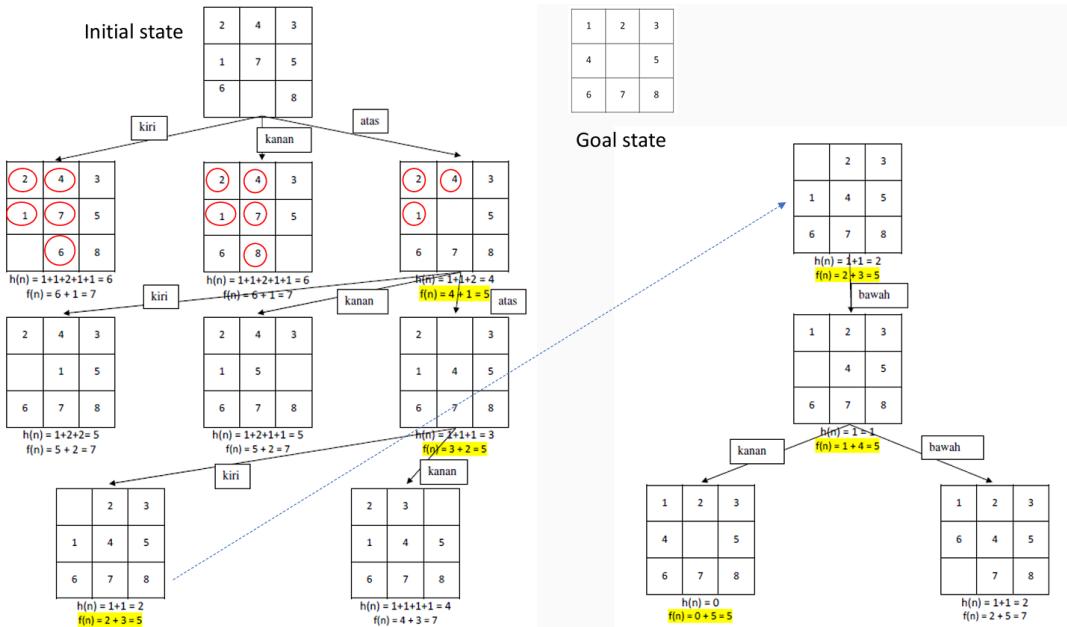
Gambar 2.2 Contoh Penyelesaian Rute dengan GBFS

(Sumber: [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf))

## 2.3 A\* Search

A\* Search adalah algoritma pencarian jalur optimal yang secara cerdas menggabungkan biaya aktual dari titik awal ke node saat ini ( $g(n)$ ) dengan estimasi heuristik biaya dari node tersebut ke tujuan ( $h(n)$ ) melalui fungsi evaluasi  $f(n) = g(n) + h(n)$ . Algoritma ini menggunakan antrian prioritas untuk selalu memilih node dengan nilai  $f(n)$  terendah untuk diekspansi, memperbarui biaya jika jalur yang lebih efisien ditemukan, dan menjamin solusi optimal jika fungsi heuristiknya admissible. Dengan demikian, A\* merupakan gabungan dari Uniform Cost Search (UCS), karena ia mempertimbangkan biaya aktual yang telah ditempuh ( $g(n)$ ) untuk memastikan optimalitas, dan Greedy Best-First Search (GBFS), karena ia menggunakan perkiraan heuristik ( $h(n)$ ) untuk

mengarahkan pencarian secara lebih efisien menuju tujuan, sehingga menyeimbangkan antara optimalitas UCS dan kecepatan terarah GBFS.



Gambar 2.3 Contoh Penyelesaian Puzzle 8 dengan A\*

(Sumber : [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf))

## 2.4. Iterative Deepening A\*

Iterative Deepening A\* (IDA\*) adalah algoritma pencarian heuristik yang menggabungkan efisiensi ruang dari pencarian depth-first dengan optimalitas dari pencarian breadth-first. Algoritma ini dimulai dengan menetapkan batas biaya awal berdasarkan nilai heuristik dari node awal. Kemudian, dilakukan pencarian depth-first secara terbatas, di mana hanya node dengan nilai total biaya  $f(n) = g(n) + h(n)$  yang tidak melebihi batas yang akan dieksplorasi. Jika solusi belum ditemukan, batas ini diperbarui ke nilai minimum biaya yang melebihi batas sebelumnya, dan proses diulang. Pendekatan iteratif ini memastikan bahwa jalur dengan biaya terendah selalu diprioritaskan, sehingga IDA\* mampu menemukan solusi optimal dengan penggunaan memori yang lebih efisien dibandingkan A\*.

## BAB III

# ANALISIS ALGORITMA

### 3.1 Definisi $f(n)$ dan $g(n)$

Fungsi  $g(n)$  merepresentasikan biaya aktual atau sebenarnya yang telah dikeluarkan untuk mencapai node  $n$  dari node awal, seperti jumlah langkah yang sudah dilakukan dalam permainan Rush Hour dari posisi awal hingga konfigurasi papan saat ini. Sementara itu,  $f(n)$  adalah fungsi evaluasi yang merupakan estimasi total biaya untuk mencapai tujuan jika jalur tersebut melalui node  $n$ , dihitung dengan menjumlahkan biaya aktual  $g(n)$  dengan  $h(n)$ , yaitu nilai heuristik yang mengestimasi biaya tersisa dari node  $n$  menuju solusi akhir.

### 3.2 Admissibility Heuristik A\* pada Puzzle Rush Hour

Sebuah heuristik  $h(n)$  dikatakan *admissible* jika untuk setiap node  $n$ , nilai  $h(n)$  tidak pernah melebih-lebihkan (*never overestimates*) biaya sebenarnya ( $h^*(n)$ ) untuk mencapai tujuan dari node  $n$  tersebut, atau dengan kata lain  $h(n) \leq h^*(n)$ . Jarak garis lurus, seperti yang digunakan dalam contoh, merupakan heuristik yang admissible karena jarak jalan sebenarnya tidak akan pernah lebih pendek dari jarak garis lurus, sehingga ia bersifat optimis dan tidak melebih-lebihkan biaya sebenarnya. Penggunaan heuristik yang admissible ini menjamin bahwa algoritma A\* akan menemukan solusi yang optimal.

### 3.3 Kesamaan UCS dan BFS pada Penyelesaian Puzzle Rush Hour

Algoritma UCS yang diimplementasikan memang menggunakan *priority queue* untuk memilih node dengan nilai  $g$  (*cost*) terkecil pada setiap iterasi. Namun, dalam implementasi, setiap langkah pergerakan selalu memiliki *cost* yang sama, yaitu  $g$  bertambah 1 untuk setiap langkah yang dihasilkan. Dengan demikian, prioritas pada *priority queue* akan selalu berdasarkan urutan penambahan *node*, sehingga perilaku UCS identik dengan BFS, baik dalam urutan *node* yang terbuat maupun *path* yang dihasilkan. Hal ini terjadi karena BFS juga mengembangkan *node* berdasarkan level (jumlah langkah dari awal), yang sama dengan nilai  $g$  pada UCS jika *cost* setiap langkah seragam.

### 3.4 Keefisiensi A\* dibandingkan UCS dalam Rush Hour

Secara teoritis, algoritma A\* umumnya lebih efisien dibandingkan dengan UCS pada penyelesaian Rush Hour, asalkan digunakan fungsi heuristik  $h(n)$  yang baik dan admissible (misalnya, estimasi jumlah mobil yang menghalangi jalan mobil target ke pintu keluar). Efisiensi A\* berasal dari kemampuannya menggunakan heuristik untuk mengarahkan pencarian ke node-node yang lebih menjanjikan (lebih dekat ke solusi) sambil tetap mempertimbangkan biaya aktual  $g(n)$  untuk menjamin optimalitas, sehingga

A\* cenderung mengeksplorasi lebih sedikit node dibandingkan UCS yang hanya memperluas *node* berdasarkan biaya kumulatif terendah tanpa panduan heuristik.

### **3.5 Apakah GBFS menjamin solusi optimal pada Rush Hour?**

Algoritma Greedy Best-First Search (GBFS) secara teoritis tidak menjamin solusi optimal untuk penyelesaian Rush Hour. GBFS hanya menggunakan fungsi heuristik  $h(n)$  untuk memandu pencarian, selalu memilih node yang tampak paling dekat dengan tujuan tanpa mempertimbangkan biaya  $g(n)$  yang telah dikeluarkan. Sifat "rakus" ini dapat membuatnya terjebak pada jalur yang secara heuristik terlihat pendek namun secara keseluruhan memiliki total langkah yang lebih banyak, sehingga meskipun mungkin menemukan solusi dengan cepat, solusi tersebut belum tentu yang paling optimal.

## BAB IV

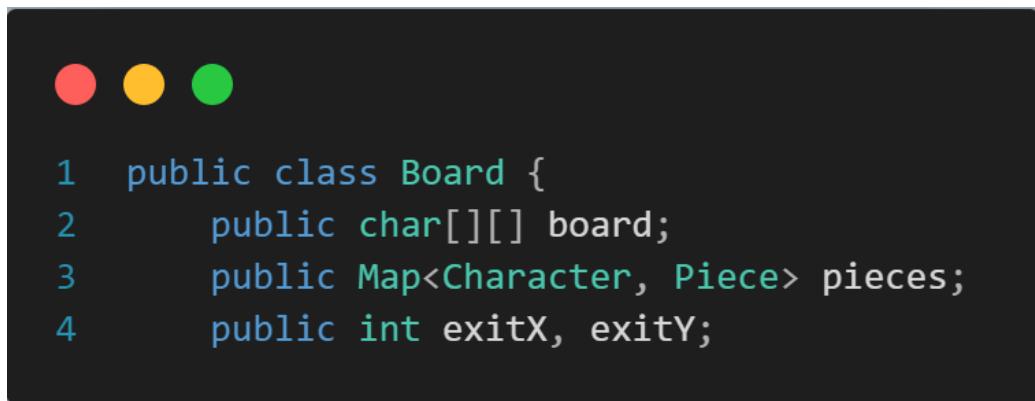
# IMPLEMENTASI PROGRAM

Untuk mempermudah proses implementasi dan penjelasan, maka struktur folder dibuat sedemikian rupa sehingga bisa melakukan pemisahan file secara jelas berdasarkan fungsionalitasnya.

```
Tucil3_13523156_13523163
├── bin/
│   ├── algoritma/
│   ├── main/
│   ├── model/
│   └── utils/
├── doc/
└── src/
    ├── algoritma/
    │   ├── A.java
    │   ├── GBFS.java
    │   ├── IDA.java
    │   └── UCS.java
    ├── main/
    │   └── Main.java
    ├── model/
    │   ├── Board.java
    │   ├── Node.java
    │   └── Piece.java
    ├── utils/
    │   ├── InputParser.java
    │   └── OutputWriter.java
    └── test/
        ├── result/
        └── tc/
├── .gitignore
└── build.bat
└── build.sh
└── makefile
└── README.md
```

## 4.1 Model

### 4.1.1 Board



```
1 public class Board {
2     public char[][] board;
3     public Map<Character, Piece> pieces;
4     public int exitX, exitY;
```

Gambar 4.1.1.1 Struktur data board

Board terdiri dari Matrix of Character, Mapping dari Piece per Character, serta titik koordinat dari titik exit (K). Board memiliki konstruktor, fungsi isGoal yang memeriksa apakah state sudah mencapai exit, fungsi getBoardKey untuk mendapat nilai string unik dari tiap piece, fungsi cloneBoard untuk melakukan deep copy agar bisa mengambil seluruh nilai data dalam board dengan menggunakan memory lain, dan movePiece untuk menggerakan suatu piece entah kemanapun arah yang memungkinkan.

```

● ○ ●
1 // Ctor
2 public Board(char[][] board, Map<Character, Piece> pieces, int exitX, int exitY) {
3     this.board = board;
4     this.pieces = (pieces != null) ? pieces : new HashMap<>();
5     this.exitX = exitX;
6     this.exitY = exitY;
7 }
8
9 // Mengecek state sudah mencapai exit atau belum
10 public boolean isGoal() {
11     Piece p = this.pieces.get('P');
12     for (int i = 0; i < p.length; i++) {
13         int x = p.x + (p.isHorizontal ? i : 0);
14         int y = p.y + (p.isHorizontal ? 0 : i);
15         if (this.exitX == x && this.exitY == y) {
16             return true;
17         }
18     }
19     return false;
20 }
21
22 // Menghasilkan string unik
23 public String getBoardKey() {
24     StringBuilder sb = new StringBuilder();
25     List<Character> sortedKeys = new ArrayList<>(this.pieces.keySet());
26     Collections.sort(sortedKeys);
27     for (char id : sortedKeys) {
28         Piece p = this.pieces.get(id);
29         sb.append(id).append(p.x).append(",").append(p.y).append(";");
30     }
31     return sb.toString();
32 }
33
34 // deep copy
35 public Board cloneBoard() {
36     int rows = board.length;
37     int cols = board[0].length;
38
39     char[][] newBoard = new char[rows][cols];
40     for (int i = 0; i < rows; i++) {
41         newBoard[i] = Arrays.copyOf(board[i], cols);
42     }
43
44     Map<Character, Piece> newPieces = new HashMap<>();
45     if (pieces == null) {
46         throw new IllegalStateException("Pieces ini bernilai null");
47     }
48     for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
49         Piece p = entry.getValue();
50         newPieces.put(entry.getKey(), new Piece(p.id, p.x, p.y, p.length, p.isHorizontal, p.isPrimary));
51     }
52     return new Board(newBoard, newPieces, exitX, exitY);
53 }
54
55 // Menggeser piece dengan id tertentu
56 public void movePiece(char id, int delta) {
57     Piece piece = pieces.get(id);
58     if (piece == null) return;
59
60     int newX = piece.x + (piece.isHorizontal ? delta : 0);
61     int newY = piece.y + (piece.isHorizontal ? 0 : delta);
62
63     // validasi posisi terhadap board
64     if (newX < 0 || newY < 0 || piece.isHorizontal && newX + piece.length - 1 >= board[0].length || (!piece.isHorizontal && newY + piece.length - 1 >= board.length)) {
65         return;
66     }
67
68     // Hapus dari posisi lama
69     if (piece.isHorizontal) {
70         for (int i = 0; i < piece.length; i++) {
71             board[piece.y][piece.x + i] = '.';
72         }
73         // Update posisi
74         piece.x += delta;
75         for (int i = 0; i < piece.length; i++) {
76             board[piece.y][piece.x + i] = piece.id;
77         }
78     } else {
79         for (int i = 0; i < piece.length; i++) {
80             board[piece.y + i][piece.x] = '.';
81         }
82         piece.y += delta;
83         for (int i = 0; i < piece.length; i++) {
84             board[piece.y + i][piece.x] = piece.id;
85         }
86     }
87 }

```

Gambar 4.1.1.2 Metode Kelas Board

#### 4.1.2 Node

Kelas node dibentuk karena setiap algoritma yang dibuat memerlukan pembentukan graf yang berisi list berkait, node memiliki struktur data yang terdiri dari board, node untuk parent, string untuk penjelasan gerakan, nilai koordinat yang digerakkan dari suatu piece.



```
1 public class Node {  
2     public Board board;  
3     public Node parent;  
4     public String moveDesc;  
5     public int g;  
6     public int h;
```

Gambar 4.1.2.1 Struktur Data Node

Selain itu, node memiliki beberapa metode yaitu konstruktor, fungsi reconstructPath untuk membentuk ulang jalur dari awal menuju node akhir, dan fungsi generateNextNodes untuk menambahkan node setiap terjadi pergeseran suatu piece yang menyertakan kondisi board sebagai node baru yang berkaitan dengan basis node sebelumnya.

```

1  public Node(Board board, Node parent, String moveDesc, int g, int h) {
2      this.board = board;
3      this.parent = parent;
4      this.moveDesc = moveDesc;
5      this.g = g;
6      this.h = h;
7  }
8
9  public static List<Node> reconstructPath(Node goalNode) {
10     LinkedList<Node> path = new LinkedList<>();
11     Node current = goalNode;
12     while (current != null) {
13         path.addFirst(current);
14         current = current.parent;
15     }
16     return path;
17 }
18
19 public static List<Node> generateNextNodes(Node currentNode) {
20     Board state = currentNode.board;
21     List<Node> nextNodes = new ArrayList<>();
22     for (Map.Entry<Character, Piece> entry : state.pieces.entrySet()) {
23         Piece piece = entry.getValue();
24         char id = entry.getKey();
25         if (piece.isHorizontal) {
26             // Geser ke kiri
27             for (int step = 1; piece.x - step >= 0; step++) {
28                 int checkX = piece.x - step;
29                 int checkY = piece.y;
30                 if (checkX < 0) break;
31                 if (!(checkX == state.exitX && checkY == state.exitY) && state.board[checkY][checkX] != '.') break;
32                 Board newBoard = state.cloneBoard();
33                 newBoard.movePiece(id, -step);
34                 String desc = "Geser " + id + " ke kiri " + step;
35                 Node nextNode = new Node(newBoard, currentNode, desc, currentNode.g + 1, 0);
36                 nextNodes.add(nextNode);
37             }
38             // Geser ke kanan
39             for (int step = 1; piece.x + piece.length - 1 + step < state.board[0].length; step++) {
40                 int checkX = piece.x + piece.length - 1 + step;
41                 int checkY = piece.y;
42                 if (checkX >= state.board[0].length) break;
43                 if (!(checkX == state.exitX && checkY == state.exitY) && state.board[checkY][checkX] != '.') break;
44                 Board newBoard = state.cloneBoard();
45                 newBoard.movePiece(id, step);
46                 String desc = "Geser " + id + " ke kanan " + step;
47                 Node nextNode = new Node(newBoard, currentNode, desc, currentNode.g + 1, 0);
48                 nextNodes.add(nextNode);
49             }
50         } else {
51             // Geser ke atas
52             for (int step = 1; piece.y - step >= 0; step++) {
53                 int checkX = piece.x;
54                 int checkY = piece.y - step;
55                 if (checkY < 0) break;
56                 if (!(checkX == state.exitX && checkY == state.exitY) && state.board[checkY][checkX] != '.') break;
57                 Board newBoard = state.cloneBoard();
58                 newBoard.movePiece(id, -step);
59                 String desc = "Geser " + id + " ke atas " + step;
60                 Node nextNode = new Node(newBoard, currentNode, desc, currentNode.g + 1, 0);
61                 nextNodes.add(nextNode);
62             }
63             // Geser ke bawah
64             for (int step = 1; piece.y + piece.length - 1 + step < state.board.length; step++) {
65                 int checkX = piece.x;
66                 int checkY = piece.y + piece.length - 1 + step;
67                 if (checkY >= state.board.length) break;
68                 if (!(checkX == state.exitX && checkY == state.exitY) && state.board[checkY][checkX] != '.') break;
69                 Board newBoard = state.cloneBoard();
70                 newBoard.movePiece(id, step);
71                 String desc = "Geser " + id + " ke bawah " + step;
72                 Node nextNode = new Node(newBoard, currentNode, desc, currentNode.g + 1, 0);
73                 nextNodes.add(nextNode);
74             }
75         }
76     }
77     return nextNodes;
78 }

```

Gambar 4.1.2.2 Metode Kelas Node

### 4.1.3 Piece

Untuk basis piece dibuat dengan struktur data yang terdiri dari character-nya, nilai koordinat dari head, nilai panjang piecennya, boolean untuk bentuk horizontal dan boolean untuk memastikan suatu piece merupakan primary atau bukan; piece tentunya memerlukan suatu konstruktor.

```
● ● ●  
1  public class Piece {  
2      public char id;  
3      public int x, y; // Ini koordinat dari head  
4      public int length;  
5      public boolean isHorizontal;  
6      public boolean isPrimary;  
7  
8      // Ctor  
9      public Piece(char id, int x, int y, int length, boolean isHorizontal, boolean isPrimary) {  
10         this.id = id;  
11         this.x = x;  
12         this.y = y;  
13         this.length = length;  
14         this.isHorizontal = isHorizontal;  
15         this.isPrimary = isPrimary;  
16     }  
}
```

Gambar 4.1.3 Struktur Data & Konstruktor Piece

## 4.2 Algoritma

Secara garis besar, masing-masing kelas algoritma yang diimplementasikan berisi segala fungsi pembantu yang akan dipanggil bersamaan dengan metode-metode kelas model sebelumnya dalam fungsi utama solve yang akan melakukan penyelesaian dari setiap papan dalam test case; untuk beberapa kasus seperti GBFS, A\*, dan IDA\* yang memiliki heuristik maka akan ditambahkan parameter pilihan heuristik untuk menentukan proses yang akan dilakukan.

### 4.2.1 Uniform Cost Search (UCS)

UCS tidak memiliki heuristik, sehingga langsung berfokus pada implementasi fungsi solve.

```

● ○ ●
1  public static List<Node> solve(Board initialBoard) {
2      PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(n -> n.g));
3      Set<String> visited = new HashSet<>();
4      long startTime = System.currentTimeMillis();
5
6      Node startNode = new Node(initialBoard, null, null, 0, 0);
7      openSet.add(startNode);
8
9      while (!openSet.isEmpty()) {
10         Node current = openSet.poll();
11
12         if (current.board.isGoal()) {
13             long endTime = System.currentTimeMillis();
14             System.out.println("Solusi ditemukan dalam " + current.g + " langkah");
15             System.out.println("Waktu pencarian: " + (endTime - startTime) + " ms");
16             List<Node> solution = Node.reconstructPath(current);
17             return solution;
18         }
19
20         String key = current.board.getBoardKey();
21         if (visited.contains(key)) continue;
22         visited.add(key);
23
24         List<Node> nextNodes = Node.generateNextNodes(current);
25         for (Node nextNode : nextNodes) {
26             String nextKey = nextNode.board.getBoardKey();
27             if (!visited.contains(nextKey)) {
28                 openSet.add(nextNode);
29             }
30         }
31     }
32
33     System.out.println("Tidak ada solusi");
34     long endTime = System.currentTimeMillis();
35     System.out.println("Waktu pencarian: " + (endTime - startTime) + " ms");
36     return null;
37 }

```

Gambar 4.2.1. Implementasi Fungsi solve

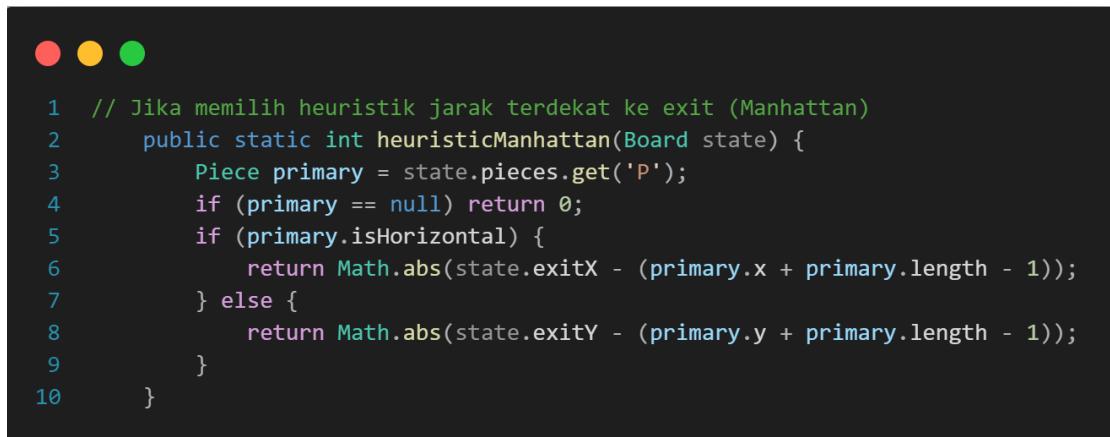
Uniform Cost Search (UCS) diimplementasikan dalam fungsi solve yang memiliki parameter sebuah objek Board sebagai keadaan awal, lalu menggunakan sebuah priority queue untuk menyimpan node-node yang akan dieksplorasi, diurutkan berdasarkan biaya kumulatif (g) dari node tersebut. Setiap node merepresentasikan sebuah keadaan papan, langkah sebelumnya, dan total biaya langkah dari awal. Selama proses pencarian, node dengan biaya terendah akan selalu diambil terlebih dahulu dari antrian. Jika node yang diambil sudah mencapai kondisi tujuan, maka solusi ditemukan dan jalur solusi direkonstruksi. Jika belum, semua kemungkinan langkah berikutnya dihasilkan , dan node-node baru yang belum pernah dikunjungi dilacak dengan visited akan dimasukkan ke

dalam antrian. Proses ini berulang hingga solusi ditemukan atau semua kemungkinan habis.

Kompleksitas waktu dari UCS pada kasus terburuk adalah  $O(b^d)$ , di mana b adalah branching factor (jumlah kemungkinan langkah dari satu keadaan) dan d adalah kedalaman solusi terpendek. Hal ini karena UCS pada dasarnya melakukan pencarian menyeluruh (breadth-first) dengan prioritas pada biaya terendah, sehingga dalam kasus terburuk harus mengeksplorasi hampir semua kemungkinan keadaan hingga menemukan solusi.

#### 4.2.2 GBFS

Secara garis besar untuk algoritma yang membutuhkan heuristik, default heuristik-nya adalah Manhattan Distance yang merupakan jarak terdekat dari primary piece (P) menuju exit (K) dalam fungsi heuristicManhattan, lalu juga diimplementasikan bonus berupa heuristik Blockers atau Obstacle Counter yang merupakan jumlah halangan (kendaraan) yang menghalangi primary piece (P) dalam fungsi heuristicBlockers. Untuk pemanggilan antara keduanya diatasi dengan fungsi calculateHeuristic yang nantinya akan dipanggil dalam fungsi solve.



The screenshot shows a code editor window with three status icons at the top: a red circle, a yellow circle, and a green circle. The main area contains the following Java code:

```
1 // Jika memilih heuristik jarak terdekat ke exit (Manhattan)
2     public static int heuristicManhattan(Board state) {
3         Piece primary = state.pieces.get('P');
4         if (primary == null) return 0;
5         if (primary.isHorizontal) {
6             return Math.abs(state.exitX - (primary.x + primary.length - 1));
7         } else {
8             return Math.abs(state.exitY - (primary.y + primary.length - 1));
9         }
10    }
```

Gambar 4.2.2.1 Fungsi heuristicManhattan

```
1 // Jika memilih heuristic banyak pieces yang menghalangi
2 public static int heuristicBlockers(Board state) {
3     Piece primary = state.pieces.get('P');
4     if (primary == null) return 0;
5
6     int blockers = 0;
7     if (primary.isHorizontal) {
8         int y = primary.y;
9         for (int x = primary.x + primary.length; x <= state.exitX; x++) {
10             char cell = state.board[y][x];
11             if (cell != '.' && cell != primary.id) {
12                 blockers++;
13             }
14         }
15     } else {
16         int x = primary.x;
17         for (int y = primary.y + primary.length; y <= state.exitY; y++) {
18             char cell = state.board[y][x];
19             if (cell != '.' && cell != primary.id) {
20                 blockers++;
21             }
22         }
23     }
24     return blockers;
25 }
```

Gambar 4.2.2.2 Fungsi heuristicBlockers

```
1 // Menghitung nilai heuristic
2 public static int calculateHeuristic(Board state, String type) {
3     if ("manhattan".equalsIgnoreCase(type)) {
4         return heuristicManhattan(state);
5     } else if ("blockingcars".equalsIgnoreCase(type)) {
6         return heuristicBlockers(state);
7     } else {
8         throw new IllegalArgumentException("Unknown heuristic type: " + type);
9     }
10 }
```

Gambar 4.2.2.3 Fungsi calculateHeuristic



```

1  public static List<Node> solve(Board initialBoard, String heuristicType) {
2      PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(n -> n.h));
3      Set<String> visited = new HashSet<>();
4      long startTime = System.currentTimeMillis();
5
6      int h0 = calculateHeuristic(initialBoard, heuristicType);
7      Node startNode = new Node(initialBoard, null, null, 0, h0);
8      openSet.add(startNode);
9
10     while (!openSet.isEmpty()) {
11         Node current = openSet.poll();
12
13         if (current.board.isGoal()) {
14             long endTime = System.currentTimeMillis();
15             System.out.println("Solusi ditemukan dalam " + current.g + " langkah");
16             System.out.println("Waktu pencarian: " + (endTime - startTime) + " ms");
17             List<Node> solution = Node.reconstructPath(current);
18             return solution;
19         }
20
21         String key = current.board.getBoardKey();
22         if (visited.contains(key)) continue;
23         visited.add(key);
24
25         List<Node> nextNodes = Node.generateNextNodes(current);
26         for (Node nextNode : nextNodes) {
27             String nextKey = nextNode.board.getBoardKey();
28             if (!visited.contains(nextKey)) {
29                 int hNext = calculateHeuristic(nextNode.board, heuristicType);
30                 nextNode.h = hNext;
31                 nextNode.g = current.g + 1; // g untuk langkah aja bukan sebagai perhitungan
32                 openSet.add(nextNode);
33             }
34         }
35     }
36
37     System.out.println("Tidak ada solusi");
38     long endTime = System.currentTimeMillis();
39     System.out.println("Waktu pencarian: " + (endTime - startTime) + " ms");
40     return null;
41 }

```

Gambar 4.2.2.4 Implementasi Fungsi solve

Greedy Best-First Search (GBFS) diimplementasikan dalam fungsi solve menerima sebuah objek Board sebagai keadaan awal dan tipe heuristik yang akan digunakan. Proses pencarian menggunakan sebuah priority queue yang mengurutkan node berdasarkan nilai heuristik ( $h$ ) terkecil, tanpa memperhatikan biaya langkah ( $g$ ). Node awal dihitung nilai heuristiknya, lalu dimasukkan ke antrian. Selama pencarian, node dengan nilai heuristik terendah diambil dari antrian. Jika node tersebut sudah mencapai tujuan, maka solusi direkonstruksi dan dikembalikan. Jika belum, semua kemungkinan langkah berikutnya dihasilkan, nilai heuristik untuk setiap node baru dihitung, dan node-node yang belum pernah

dikunjungi dimasukkan ke antrian. Proses ini berulang hingga solusi ditemukan atau semua kemungkinan habis.

Kompleksitas waktu dari GBFS pada kasus terburuk adalah  $O(b^d)$ , di mana b adalah branching factor (jumlah kemungkinan langkah dari satu keadaan) dan d adalah kedalaman solusi. Hal ini karena GBFS dapat mengeksplorasi hampir semua node pada level tertentu jika heuristik yang digunakan tidak cukup baik untuk langsung mengarahkan pencarian ke solusi.

#### 4.2.3 A\*

Sama seperti GBFS, A\* memerlukan heuristik yang secara default adalah Manhattan Distance dalam fungsi calculateManhattanHeuristic, dengan alternatif heuristik Obstacle dalam fungsi calculateObstacleHeuristic dan tambahan heuristik kombinasi dari keduanya dalam fungsi calculateCombinedHeuristic.

```
● ● ●
1 // Hitung jumlah penghalang untuk Heuristik Obstacle
2 public static int calculateObstacleHeuristic(Board board, Piece primaryPiece) {
3     int obstacles = 0;
4     int targetX = board.exitX;
5     int primaryX = primaryPiece.x;
6     int primaryY = primaryPiece.y;
7     for (Piece piece : board.pieces.values()) {
8         if (primary != primaryPiece && piece.isHorizontal && piece.y == primaryY) {
9             if ((primaryY < targetX && piece.x > primaryY && piece.x < targetX) ||
10                 (primaryY > targetX && piece.x < primaryY && piece.x + piece.length > targetX)) {
11                 obstacles++;
12             }
13         } else if (piece != primaryPiece && !piece.isHorizontal) {
14             if (piece.x >= Math.min(primaryX, targetX) && piece.x <= Math.max(primaryX, targetX) && piece.y <= primaryY && piece.y + piece.length > primaryY) {
15                 obstacles++;
16             }
17         }
18     }
19     return obstacles;
20 }
21
22 // Hitung jarak Manhattan untuk Heuristik Manhattan Distance
23 public static int calculateManhattanHeuristic(Board board, Piece primaryPiece) {
24     int horizontalDistance = Math.abs(primaryPiece.x - board.exitX);
25     int verticalDistance = Math.abs(primaryPiece.y - board.exitY) / 2;
26     return horizontalDistance + verticalDistance;
27 }
28
29 // Hitung kombinasi dari Heuristik Obstacle dan Manhattan Distance
30 public static int calculateCombinedHeuristic(Board board, Piece primaryPiece) {
31     int obstacleCount = calculateObstacleHeuristic(board, primaryPiece);
32     int manhattanDistance = calculateManhattanHeuristic(board, primaryPiece);
33     return obstacleCount + (manhattanDistance * 2);
34 }
35
```

Gambar 4.2.3.1 Fungsi Kalkulasi Heuristik

```
● ● ●

1  public static List<Node> solve(Board initialBoard, String heuristicType) {
2      PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(n -> n.g + n.h));
3      Set<String> visited = new HashSet<>();
4      long startTime = System.currentTimeMillis();
5
6      Piece primaryPiece = initialBoard.pieces.get('P');
7      int h0 = 0;
8      h0 = switch (heuristicType.toLowerCase()) {
9          case "obstacle" -> calculateObstacleHeuristic(initialBoard, primaryPiece);
10         case "combined" -> calculateCombinedHeuristic(initialBoard, primaryPiece);
11         default -> calculateManhattanHeuristic(initialBoard, primaryPiece);
12     };
13
14     Node startNode = new Node(initialBoard, null, null, 0, h0);
15     openSet.add(startNode);
16
17     while (!openSet.isEmpty()) {
18         Node current = openSet.poll();
19         if (current.board.isGoal()) {
20             long endTime = System.currentTimeMillis();
21             List<Node> solution = Node.reconstructPath(current);
22             System.out.println("Solusi ditemukan dalam " + current.g + " langkah");
23             System.out.println("Waktu pencarian: " + (endTime - startTime) + " ms");
24             return solution;
25         }
26         String key = current.board.getBoardKey();
27         if (visited.contains(key)) continue;
28         visited.add(key);
29
30         List<Node> nextNodes = Node.generateNextNodes(current);
31         for (Node nextNode : nextNodes) {
32             String nextKey = nextNode.board.getBoardKey();
33             if (!visited.contains(nextKey)) {
34                 Piece nextPrimary = nextNode.board.pieces.get('P');
35                 int hNext = 0;
36                 hNext = switch (heuristicType.toLowerCase()) {
37                     case "obstacle" -> calculateObstacleHeuristic(nextNode.board, nextPrimary);
38                     case "combined" -> calculateCombinedHeuristic(nextNode.board, nextPrimary);
39                     default -> calculateManhattanHeuristic(nextNode.board, nextPrimary);
40                 };
41                 nextNode.h = hNext;
42                 openSet.add(nextNode);
43             }
44         }
45     }
46     System.out.println("Tidak ditemukan solusi");
47     long endTime = System.currentTimeMillis();
48     System.out.println("Waktu pencarian: " + (endTime - startTime) + " ms");
49     return null;
50 }
```

Gambar 4.2.3.2 Implementasi Fungsi solve

A\* diimplementasikan dalam fungsi solve pada kelas A. Fungsi ini menerima objek Board sebagai keadaan awal dan tipe heuristik yang akan digunakan. Proses pencarian menggunakan priority queue yang mengurutkan node berdasarkan nilai

$g + h$ , di mana  $g$  adalah biaya dari awal ke node saat ini (jumlah langkah), dan  $h$  adalah estimasi biaya ke tujuan berdasarkan heuristik yang dipilih. Node awal dihitung nilai heuristiknya, lalu dimasukkan ke antrian. Selama pencarian, node dengan nilai  $g + h$  terkecil diambil dari antrian. Jika node tersebut sudah mencapai tujuan, maka solusi direkonstruksi dan dikembalikan. Jika belum, semua kemungkinan langkah berikutnya dihasilkan, nilai heuristik untuk setiap node baru dihitung, dan node-node yang belum pernah dikunjungi dimasukkan ke antrian. Proses ini berulang hingga solusi ditemukan atau semua kemungkinan habis.

Kompleksitas waktu untuk A\* pada kasus terburuk adalah  $O(b^d)$ , di mana  $b$  adalah branching factor (jumlah kemungkinan langkah dari satu keadaan) dan  $d$  adalah kedalaman solusi. Namun, jika heuristik yang digunakan admissible dan konsisten, A\* biasanya jauh lebih efisien daripada pencarian buta karena dapat memangkas banyak node yang tidak relevan.

#### 4.2.4 IDA\*

```

1 package algoritma;
2
3 import java.util.*;
4 import model.*;
5
6 public class IDA {
7     public static List<Node> solve(Board initialBoard, String heuristicType) {
8         long startTime = System.currentTimeMillis();
9
10        Piece primaryPiece = initialBoard.pieces.get("P");
11        int h0 = getHeuristic(initialBoard, primaryPiece, heuristicType);
12        Node startNode = new Node(initialBoard, parent=null, moveDesc=null, E:0, h0);
13        int threshold = h0;
14
15        while (true) {
16            // System.out.println("Mulai iterasi dengan threshold: " + threshold); // debugging
17
18            Set<String> visited = new HashSet<>();
19            Map<String, Integer> gScoreMap = new HashMap<>();
20
21            Result result = search(startNode, threshold, heuristicType, visited, gScoreMap);
22
23            if (result.found) {
24                long endTime = System.currentTimeMillis();
25                List<Node> solution = Node.reconstructPath(result.goalNode);
26                System.out.println("Solusi ditemukan dalam " + result.goalNode.g + " langkah");
27                System.out.println("Waktu pencarian: " + (endTime - startTime) + " ms");
28                return solution;
29            }
29
30            if (result.minThreshold == Integer.MAX_VALUE) {
31                long endTime = System.currentTimeMillis();
32                System.out.println("Tidak ditemukan solusi!");
33                System.out.println("Waktu pencarian: " + (endTime - startTime) + " ms");
34                return null;
35            }
36
37            threshold = result.minThreshold;
38        }
39    }
40
41
42    private static Result search(Node node, int threshold, String heuristicType, Set<String> visited, Map<String, Integer> gScoreMap) {
43        int f = node.g + node.h;
44        String key = node.board.getBoardKey();
45
46        if (gScoreMap.containsKey(key) && gScoreMap.get(key) <= node.g) {
47            return new Result(false, goalNode=null, Integer.MAX_VALUE);
48        }
49        gScoreMap.put(key, node.g);
50
51        if (f > threshold) {
52            return new Result(false, goalNode=null, f);
53        }
54
55        if (node.board.isGoal()) {
56            return new Result(true, node, f);
57        }
58
59        visited.add(key);
60        int min = Integer.MAX_VALUE;
61
62        List<Node> children = Node.generateNextNodes(node);
63
64        for (Node child : children) {
65            child.g = node.g + 1;
66            Piece nextPrimary = child.board.pieces.get("P");
67            child.h = getHeuristic(child.board, nextPrimary, heuristicType);
68
69            if (child.h < min) {
70                min = child.h;
71            }
72        }
73
74        if (min > threshold) {
75            return new Result(false, goalNode=null, min);
76        }
77    }
78
79    private static int getHeuristic(Board board, Piece piece, String heuristicType) {
80        if (heuristicType.equals("E"))
81            return 0;
82        else if (heuristicType.equals("H"))
83            return calculateH(board, piece);
84        else if (heuristicType.equals("G"))
85            return calculateG(board, piece);
86        else
87            throw new IllegalArgumentException("Heuristic type not supported");
88    }
89
90    private static int calculateH(Board board, Piece piece) {
91        int h = 0;
92
93        for (Piece p : board.pieces.values()) {
94            if (p.equals(piece))
95                continue;
96
97            if (p.x == piece.x && p.y == piece.y)
98                h++;
99            else if (Math.abs(p.x - piece.x) + Math.abs(p.y - piece.y) == 1)
100                h++;
101        }
102
103        return h;
104    }
105
106    private static int calculateG(Board board, Piece piece) {
107        int g = 0;
108
109        for (Piece p : board.pieces.values()) {
110            if (p.equals(piece))
111                continue;
112
113            if (p.x == piece.x && p.y == piece.y)
114                g++;
115            else if (Math.abs(p.x - piece.x) + Math.abs(p.y - piece.y) == 1)
116                g++;
117        }
118
119        return g;
120    }
121}
```

```

67     child.h = getHeuristic(child.board, nextPrimary, heuristicType);
68
69     Piece currentPrimary = node.board.pieces.get('P');
70     String moveDirection = "";
71     if (nextPrimary.x > currentPrimary.x) {
72         moveDirection = "ke kanan " + (nextPrimary.x - currentPrimary.x);
73     } else if (nextPrimary.x < currentPrimary.x) {
74         moveDirection = "ke kiri " + (currentPrimary.x - nextPrimary.x);
75     } else if (nextPrimary.y > currentPrimary.y) {
76         moveDirection = "ke bawah " + (nextPrimary.y - currentPrimary.y);
77     } else if (nextPrimary.y < currentPrimary.y) {
78         moveDirection = "ke atas " + (currentPrimary.y - nextPrimary.y);
79     }
80     child.moveDesc = "Geser P " + moveDirection;
81 }
82
83 children.sort(Comparator.comparingInt(c -> c.g + c.h));
84
85 for (Node child : children) {
86     String childKey = child.board.getBoardKey();
87     if (visited.contains(childKey)) continue;
88
89     Result result = search(child, threshold, heuristicType, visited, gScoreMap);
90     if (result.found) {
91         return result;
92     }
93
94     if (result.minThreshold < min) {
95         min = result.minThreshold;
96     }
97
98     visited.remove(childKey);
99 }
100
101 visited.remove(key);
102 return new Result(found=false, goalNode=null, min);
103 }
104
105 private static int getHeuristic(Board board, Piece primaryPiece, String heuristicType) {
106     if (primaryPiece == null) return 0;
107     return switch (heuristicType.toLowerCase()) {
108         case "obstacle" -> A.calculateObstacleHeuristic(board, primaryPiece);
109         case "combined" -> A.calculateCombinedHeuristic(board, primaryPiece);
110         default -> A.calculateManhattanHeuristic(board, primaryPiece);
111     };
112 }
113
114 private static class Result {
115     boolean found;
116     Node goalNode;
117     int minThreshold;
118
119     Result(boolean found, Node goalNode, int minThreshold) {
120         this.found = found;
121         this.goalNode = goalNode;
122         this.minThreshold = minThreshold;
123     }
124 }
125 }
```

IDA\* diimplementasikan melalui fungsi solve yang akan menghitung nilai heuristik awal dari posisi saat ini menggunakan metode getHeuristic, lalu membuat simpul awal (startNode) dengan level awal ( $g = 0$ ) dan dengan nilai heuristik yang didapat. Nilai heuristik ini akan digunakan sebagai threshold untuk eksplorasi pencarian. Selanjutnya, masuk ke loop utama yang secara bertahap meningkatkan threshold. Di setiap iterasi, algoritma akan memanggil fungsi search untuk mencoba menemukan solusi dalam batas atas threshold saat ini. Fungsi search menggunakan pendekatan DFS yang dikombinasikan dengan heuristik ( $f = g + h$ ) serta menyimpan informasi visited. Selain itu disimpan juga, gScoreMap untuk memastikan bahwa hanya jalur dengan *cost* terendah saja yang akan diproses.

Kompleksitas waktu dari IDA\* awalnya adalah  $O(b^d * d)$  dengan  $b$  adalah branching factor dan  $d$  adalah kedalaman solusi optimal. Dengan optimasi menggunakan gScoreMap dan sorting children berdasarkan  $f$ , kompleksitasnya mendekati A\*, yaitu  $O(b^d)$ .

## BAB V

# PENGUJIAN

Untuk mempermudah pengujian, kami sebenarnya membuat banyak test case yang akhirnya kami pilih 4 yang menggambarkan kondisi berbeda-beda yaitu:

Test Case 1	<b>6 6</b> 12 .ABBCC .AD... PPDEFGK HIIIEFG HJJEFKL MMM..L
Test Case 2	<b>3 4</b> 2 ...P .A.P .ABB   K
Test Case 3	<b>6 6</b> 13   K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM

Test Case 4

```
6 6
9
..AABB
.CCD..
.E.DFF
K.E.PPG
.E..HG
.IIIHG
```

### 5.1. UCS

Hasil Test Case 1

Solusi ditemukan dalam 26 langkah  
Waktu pencarian: 141 ms

Langkah 26: Geser P ke kanan 3  
HBBCCG  
HA...G  
.A...PP  
IIIDFL  
JJDEFL  
MMMEF.

Hasil Test Case 2

Solusi ditemukan dalam 3 langkah  
Waktu pencarian: 10 ms

Langkah 3: Geser P ke bawah 2  
.A.  
.A.  
BB.P  
P

Hasil Test Case 3

Solusi ditemukan dalam 26 langkah  
Waktu pencarian: 141 ms

	<p>Langkah 22: Geser P ke atas 2</p>
Hasil Test Case 4	<p>Solusi ditemukan dalam 24 langkah Waktu pencarian: 240 ms</p> <p>Langkah 24: Geser P ke kiri 3</p>

## 5.2. GBFS

### 5.2.1. Manhattan Distance Heuristic

Hasil Test Case 1	<p>Solusi ditemukan dalam 278 langkah Waktu pencarian: 116 ms</p> <p>Langkah 278: Geser P ke kanan 2</p>
Hasil Test Case 2	<p>Solusi ditemukan dalam 3 langkah Waktu pencarian: 15 ms</p>

	<p>Langkah 3: Geser P ke bawah 2</p> <pre> .A..  .A..  .BB  .P </pre>
Hasil Test Case 3	<p>Solusi ditemukan dalam 176 langkah Waktu pencarian: 98 ms</p> <p>Langkah 176: Geser P ke atas 2</p> <pre> P  BBPLCC  AE.LFF  AE..HH  GEIIJJ  GNNN.D  ..MM.D </pre>
Hasil Test Case 4	<p>Solusi ditemukan dalam 278 langkah Waktu pencarian: 198 ms</p> <p>Langkah 278: Geser P ke kiri 3</p> <pre> .EAABB  .ECC..  .E.FFG  PP....G  ...DHG  IIDH. </pre>

### 5.2.2. Obstacle Heuristic

Hasil Test Case 1	<p>Solusi ditemukan dalam 61 langkah Waktu pencarian: 96 ms</p>
-------------------	---

	<p>Langkah 61: Geser P ke kanan 2</p> <p>.BBCCG</p> <p>HA...G</p> <p>HAD...PP</p> <p>IIDEF.</p> <p>JJ.EFL</p> <p>MMMEFL</p>
Hasil Test Case 2	<p>Solusi ditemukan dalam 3 langkah</p> <p>Waktu pencarian: 16 ms</p> <p>Langkah 3: Geser P ke bawah 2</p> <p>.A..</p> <p>.A..</p> <p>.BBP</p> <p>P</p>
Hasil Test Case 3	<p>Solusi ditemukan dalam 334 langkah</p> <p>Waktu pencarian: 116 ms</p> <p>Langkah 334: Geser P ke atas 2</p> <p>P</p> <p>BBPLCC</p> <p>AE.LFF</p> <p>AE.HH.</p> <p>.EIIJJ</p> <p>GNNN.D</p> <p>GMM..D</p>
Hasil Test Case 4	<p>Solusi ditemukan dalam 976 langkah</p> <p>Waktu pencarian: 187 ms</p> <p>Langkah 976: Geser P ke kiri 3</p> <p>.EAABB</p> <p>.ECC..</p> <p>.EFF.G</p> <p>PP...HG</p> <p>...DHG</p> <p>IID..</p>

## 5.3 A\*

### 5.3.1 Manhattan Distance Heuristic

Hasil Test Case 1	<p>Solusi ditemukan dalam 27 langkah Waktu pencarian: 156 ms</p> <p>Langkah 27: Geser P ke kanan 2</p> <p>HBBCCG HA...G .A...PP IIIDEFL JJDEFL MMMEF.</p>
Hasil Test Case 2	<p>Solusi ditemukan dalam 3 langkah Waktu pencarian: 15 ms</p> <p>Langkah 3: Geser P ke bawah 2</p> <p>.A.. .A.. .BBP P</p>
Hasil Test Case 3	<p>Solusi ditemukan dalam 22 langkah Waktu pencarian: 111 ms</p> <p>Langkah 22: Geser P ke atas 2</p> <p>P BBP.CC AE.LFF AE.LHH GEIIJJ GNNN.D .MM..D</p>
Hasil Test Case 4	<p>Solusi ditemukan dalam 24 langkah Waktu pencarian: 217 ms</p>

	<p>Langkah 24: Geser P ke kiri 3</p> <p>.EAABB .ECCHG .EFFHG <b>PP</b>...G ...D.. IID..</p>
--	---

### 5.3.2 Obstacle Heuristic

Hasil Test Case 1	<p>Solusi ditemukan dalam 26 langkah Waktu pencarian: 187 ms</p> <p>Langkah 26: Geser P ke kanan 3</p> <p>HBBCCG HA...G .A...<b>PP</b> IIDEFL JJDEFL MMMEF.</p>
Hasil Test Case 2	<p>Solusi ditemukan dalam 3 langkah Waktu pencarian: 11 ms</p> <p>Langkah 3: Geser P ke bawah 2</p> <p>.A. .A.. BB.<b>P</b> <b>P</b></p>
Hasil Test Case 3	<p>Solusi ditemukan dalam 22 langkah Waktu pencarian: 131 ms</p> <p>Langkah 22: Geser P ke atas 2</p> <p><b>P</b> BB<b>P</b>.CC AE.LFF AE.LHH GEIIJJ GNNN.D .MM..D</p>

Hasil Test Case 4

Solusi ditemukan dalam 24 langkah  
Waktu pencarian: 254 ms

Langkah 24: Geser P ke kiri 4

.EAABB  
.ECCH.  
.EFFH.  
**PP**...**G**  
...D.G  
IID.G

### 5.3.3 Combined Heuristic

Hasil Test Case 1

Solusi ditemukan dalam 28 langkah  
Waktu pencarian: 101 ms

Langkah 28: Geser P ke kanan 2  
HBBCCG  
HA...G  
.A...**PP**  
IIIDEFL  
JJDEFL  
MMMEF.

Hasil Test Case 2

Solusi ditemukan dalam 3 langkah  
Waktu pencarian: 15 ms

Langkah 3: Geser P ke bawah 2  
.A..  
.A..  
.BBP  
**P**

Hasil Test Case 3

Solusi ditemukan dalam 22 langkah  
Waktu pencarian: 110 ms

	<p>Langkah 22: Geser P ke atas 2</p>
Hasil Test Case 4	<p>Solusi ditemukan dalam 28 langkah Waktu pencarian: 101 ms</p> <p>Langkah 24: Geser P ke kiri 3</p>

## 5.4 IDA\*

### 5.4.1 Manhattan Distance Heuristic

Hasil Test Case 1	<p>Solusi ditemukan dalam 26 langkah Waktu pencarian: 1873 ms</p> <p>Langkah 26: Geser P ke kanan 2</p>
Hasil Test Case 2	<p>Solusi ditemukan dalam 3 langkah Waktu pencarian: 28 ms</p> <p>Langkah 3: Geser P ke bawah 2</p>
Hasil Test Case 3	<p>Solusi ditemukan dalam 22 langkah Waktu pencarian: 279 ms</p>

	<p>Langkah 22: Geser P ke atas 2</p> <pre> A E L F F G G E I I J J G N N N D . . M M . D .     </pre>
Hasil Test Case 4	<p>Solusi ditemukan dalam 24 langkah Waktu pencarian: 2766 ms</p> <p>Langkah 24: Geser P ke kiri 3</p> <pre> . E A A B B . E C C H G . E F F H G p p . . G .     </pre> <p>...D..</p> <p>IIID..</p>

#### 5.4.2 Obstacle Heuristic

Hasil Test Case 1	<p>Solusi ditemukan dalam 26 langkah Waktu pencarian: 1356 ms</p> <p>Langkah 26: Geser P ke kanan 3</p> <pre> H B B C C G H A . . G . . A . . p p I I D E F L J J D E F L M M M E F .     </pre>
Hasil Test Case 2	<p>Solusi ditemukan dalam 3 langkah Waktu pencarian: 62 ms</p> <p>Langkah 3: Geser P ke bawah 2</p> <pre> . A . . . . A . . . . B B . . . . P . .     </pre>
Hasil Test Case 3	<p>Solusi ditemukan dalam 22 langkah Waktu pencarian: 359 ms</p> <p>Langkah 22: Geser P ke atas 2</p> <pre> A E L F F G G E I I J J G N N N D . . M M . D .     </pre>
Hasil Test Case 4	<p>Solusi ditemukan dalam 24 langkah Waktu pencarian: 3227 ms</p>

	<p>Langkah 24: Geser P ke kiri 4</p> <p>.EAAAB .ECCHG .EFFHG <b>P</b>...<b>G</b> ...D.. IIID..</p>
--	--

### 5.4.3 Combined Heuristic

Hasil Test Case 1	<p>Solusi ditemukan dalam 27 langkah Waktu pencarian: 579 ms</p> <p>Langkah 27: Geser P ke kanan 2</p> <p>HBBCCG HA...G .A...<b>PP</b> IIIDFL JJDEFL MMMEF.</p>
Hasil Test Case 2	<p>Solusi ditemukan dalam 3 langkah Waktu pencarian: 26 ms</p> <p>Langkah 3: Geser P ke bawah 2</p> <p>.A. .A. .BB. <b>P</b></p>
Hasil Test Case 3	<p>Solusi ditemukan dalam 23 langkah Waktu pencarian: 401 ms</p> <p>Langkah 23: Geser P ke atas 1</p> <p><b>P</b> BBP.CC AE<b>LFF</b> AE.LHH GEIIIJ GNND.D .MM..D</p>
Hasil Test Case 4	<p>Solusi ditemukan dalam 24 langkah Waktu pencarian: 3142 ms</p> <p>Langkah 24: Geser P ke kiri 3</p> <p>.EAAAB .ECCHG .EFFHG <b>PP</b>...<b>G</b> ...D.. IIID..</p>

## **BAB VI**

## **KESIMPULAN & SARAN**

### **6.1 Kesimpulan**

Dalam penyelesaian permainan Rush Hour, algoritma A\* terbukti sebagai pendekatan yang paling efisien jika dilihat dari jumlah node yang dieksplorasi serta jumlah langkah (moves) dalam solusi yang relatif minimal, setelah Uniform Cost Search (UCS). Meskipun UCS menghasilkan solusi dengan langkah paling sedikit, metode ini mengeksplorasi node dalam jumlah sangat besar, sehingga memerlukan waktu eksekusi paling lama dibandingkan metode lainnya. Selain itu, Greedy Best First Search (GBFS) mampu menemukan solusi dengan sangat cepat, namun solusi yang dihasilkannya cenderung tidak optimal karena jumlah langkahnya lebih banyak dibandingkan A\* maupun UCS.

### **6.2 Saran**

Untuk seterusnya, sebaiknya implementasi bisa mengeksplorasi mengenai memori lagi agar bisa meningkatkan efisiensi dalam optimasi penggunaan memori bagi UCS. Selain itu, diperlukan percobaan untuk skala board yang lebih besar untuk menguji algoritma yang diimplementasikan dengan skala yang lebih besar. Selain itu, bisa mencoba alternatif algoritma lain seperti Fringe Search yang menjadi alternatif bagi A\*, atau sebagainya.

## LAMPIRAN

Github Repository: [https://github.com/Inforable/Tucil3\\_13523156\\_13523163](https://github.com/Inforable/Tucil3_13523156_13523163)

Tabel berikut

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

## **DAFTAR PUSTAKA**

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf)

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf)

<https://www.baeldung.com/cs/find-path-uniform-cost-search>

<https://ojs.aaai.org/index.php/SOCS/article/view/18425/18216>

<https://www.datacamp.com/tutorial/a-star-algorithm>

<https://www.sciencedirect.com/science/article/pii/0004370285900840>