

Tugas Kecil 3 IF2211 Strategi Algoritma
Semester II tahun 2024/2025
Penyelesaian Puzzle Rush Hour dengan Algoritma Pathfinding



Oleh:
Muhammad Fithra Rizki - 13523049
Ahmad Wicaksono - 13523121

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

DAFTAR ISI

BAB I

PENJELASAN ALGORITMA

1.1 Algoritma *Greedy Breadth First Search*

Algoritma Greedy Best-First Search (Greedy BFS) adalah algoritma pencarian yang menggunakan fungsi heuristik $h(n)$ untuk memperkirakan jarak dari suatu node ke node tujuan. Algoritma ini bersifat "serakah" karena selalu memilih node yang tampaknya paling dekat dengan tujuan berdasarkan nilai heuristik terkecil, tanpa memperhitungkan biaya yang telah dikeluarkan sebelumnya. Karena hanya mempertimbangkan estimasi ke depan, algoritma ini dapat bergerak cepat menuju tujuan, namun cenderung mengabaikan jalur yang lebih baik secara keseluruhan. Akibatnya, solusi yang ditemukan tidak dijamin optimal, dan kelengkapan algoritma juga tidak selalu terjamin, terutama jika tidak ada mekanisme pencatatan node yang telah dikunjungi. Dalam praktiknya, algoritma ini cocok digunakan ketika waktu pencarian lebih penting daripada optimalitas solusi, seperti dalam aplikasi game atau sistem navigasi real-time dengan keterbatasan waktu.

Berikut adalah langkah-langkah dalam menggunakan algoritma Greedy Breadth First Search:

1. Inisialisasi *open list* dengan node awal.
2. Selama *open list* tidak kosong:
 - a. Ambil node dengan nilai heuristik $h(n)$ terkecil dari *open list*.
 - b. Jika node tersebut adalah tujuan, kembalikan jalur sebagai solusi.
 - c. Tandai node sebagai *visited*.
 - d. Tambahkan semua tetangganya yang belum dikunjungi ke *open list*, bersama nilai heuristiknya.
3. Jika *open list* kosong dan tujuan belum ditemukan, pencarian gagal.

1.2 Algoritma *Uniform Cost Search*

Algoritma Uniform Cost Search (UCS) merupakan strategi pencarian yang menjamin solusi optimal dengan memilih jalur berdasarkan biaya aktual terendah dari titik awal hingga ke node saat ini, dilambangkan sebagai $g(n)$. UCS tidak menggunakan heuristik apa pun, sehingga eksplorasinya tidak diarahkan secara khusus ke tujuan, melainkan meluas berdasarkan akumulasi biaya. Algoritma ini sangat mirip dengan algoritma Dijkstra dan dijamin lengkap serta optimal.

selama semua biaya antar node bernilai non-negatif. Meskipun efektif untuk menemukan solusi terbaik, UCS dapat menjadi lambat atau boros memori pada graf atau ruang pencarian yang besar karena eksplorasi dilakukan secara menyeluruh, bahkan pada jalur yang tidak menjanjikan arah ke tujuan.

Berikut adalah langkah-langkah dalam menggunakan algoritma Uniform Cost Search:

1. Inisialisasi *priority queue* dengan node awal dan biaya 0.
2. Selama *priority queue* tidak kosong:
 - a. Ambil node dengan total biaya $g(n)$ terendah dari antrian.
 - b. Jika node adalah tujuan, kembalikan jalurnya.
 - c. Tandai node sebagai *visited*.
 - d. Tambahkan tetangga-tetangga node ke antrian dengan biaya total yang diperbarui.
3. Jika antrian kosong dan tujuan belum ditemukan, pencarian gagal.

1.3 Algoritma A^*

A^* adalah algoritma pencarian yang sangat efisien dan optimal jika fungsi heuristik $h(n)$ yang digunakan bersifat tidak melebihi-lebihkan biaya ke tujuan. A^* menggabungkan keunggulan UCS dan Greedy BFS dengan menggunakan fungsi evaluasi $f(n)=g(n)+h(n)$, yaitu kombinasi dari biaya aktual dari awal ke node saat ini dan estimasi biaya ke tujuan. Dengan pendekatan ini, A^* dapat secara efektif menyeimbangkan eksplorasi ke arah tujuan dan pertimbangan terhadap biaya yang telah dikeluarkan. A^* bersifat lengkap dan optimal jika syarat heuristik terpenuhi. Namun, karena A^* menyimpan banyak informasi (semua node terbuka dan tertutup), penggunaan memori menjadi kelemahan utama, terutama pada graf besar atau pencarian dengan banyak kemungkinan jalur.

Berikut adalah langkah-langkah dalam menggunakan algoritma A^* :

1. Inisialisasi *open list* dengan node awal dan $f(n)=g(n)+h(n)$.
2. Selama *open list* tidak kosong:
 - a. Ambil node dengan nilai $f(n)$ terendah.
 - b. Jika node tersebut adalah tujuan, kembalikan jalur.
 - c. Pindahkan node ke *closed list*.
 - d. Untuk setiap tetangga:
 - Hitung $g(n)$, $h(n)$, dan $f(n)$.
 - Jika node belum ada di *open/closed list*, tambahkan.

- Jika sudah ada dan nilai baru lebih baik, perbarui.
3. Jika tidak ada solusi dan open list kosong, pencarian gagal.

1.4 Algoritma *Iterative Deepening A (IDA*)**

Algoritma Iterative Deepening A* (IDA*) adalah algoritma pencarian yang menggabungkan kelebihan dari A* dalam menemukan solusi optimal dengan efisiensi memori dari pencarian Depth-First Search (DFS). Alih-alih menyimpan semua node dalam memori seperti A*, IDA* melakukan pencarian berulang kali menggunakan DFS yang dibatasi oleh nilai ambang $f(n)=g(n)+h(n)$, yaitu jumlah biaya aktual dari awal hingga node n dan estimasi biaya ke tujuan. Setiap iterasi dilakukan dengan batas nilai f tertentu, dan jika solusi belum ditemukan, batas ini akan diperbarui dengan nilai f terkecil yang melebihi batas sebelumnya. Pendekatan ini memungkinkan pencarian yang efisien secara memori, walaupun dapat menyebabkan beberapa node dieksplorasi lebih dari sekali. IDA* bersifat lengkap dan optimal jika heuristik yang digunakan bersifat admissible. Algoritma ini sangat efektif digunakan dalam ruang pencarian yang besar seperti pada puzzle atau game karena penggunaan memorinya yang rendah dibanding A*.

Berikut adalah langkah-langkah dalam menggunakan algoritma Iterative Deepening A*:

1. Hitung nilai $f(\text{start}) = g(\text{start}) + h(\text{start})$, dan tetapkan ini sebagai batas awal (threshold).
2. Lakukan pencarian DFS terbatas (depth-limited) dengan syarat:
 - Jika $f(n) > \text{threshold}$, simpan nilai f tersebut sebagai kandidat batas berikutnya.
 - Jika node tujuan ditemukan, hentikan dan kembalikan jalur sebagai solusi.
3. Jika DFS selesai tanpa menemukan tujuan, perbarui threshold menjadi nilai f terkecil yang melebihi threshold sebelumnya.
4. Ulangi langkah DFS dengan threshold baru.
5. Teruskan proses hingga node tujuan ditemukan atau semua kemungkinan habis.

BAB II

ANALISIS ALGORITMA

2.1 Definisi $f(n)$ dan $g(n)$ Setiap Algoritma

Dalam konteks algoritma pencarian jalur, fungsi $f(n)$ dan $g(n)$ memiliki definisi yang berbeda tergantung pada jenis algoritma yang digunakan. Pada algoritma Uniform Cost Search (UCS), fungsi $f(n)=g(n)$, di mana $g(n)$ adalah total biaya kumulatif dari node awal ke node n . UCS tidak menggunakan heuristik dan hanya mempertimbangkan biaya aktual. Pada algoritma Greedy Best-First Search (GBFS), $f(n)=h(n)$, yaitu hanya menggunakan fungsi heuristik untuk memperkirakan jarak dari node saat ini ke node tujuan tanpa mempertimbangkan biaya sebelumnya. Sebaliknya, algoritma A^* menggunakan pendekatan gabungan dengan $f(n)=g(n)+h(n)$, yang menjumlahkan biaya aktual dari awal ke node saat ini ($g(n)$) dengan estimasi biaya ke tujuan ($h(n)$). Pendekatan ini memungkinkan A^* menyeimbangkan eksplorasi jalur berdasarkan pengalaman masa lalu dan prediksi masa depan.

Sementara itu, algoritma Iterative Deepening A^* (IDA*), yang merupakan pengembangan dari A^* , juga menggunakan fungsi evaluasi $f(n)=g(n)+h(n)$. Namun, tidak seperti A^* yang menggunakan antrian prioritas, IDA* menggabungkan pendekatan DFS dengan batas ambang $f(n)$ yang terus ditingkatkan secara iteratif. Pada setiap iterasi, hanya node-node dengan $f(n) \leq \text{threshold}$ yang akan dijelajahi. Dengan demikian, IDA* tetap mengandalkan definisi yang sama terhadap $g(n)$ dan $h(n)$, namun mengeksplorasinya secara lebih hemat memori dibanding A^* , karena tidak menyimpan semua node dalam antrian seperti A^* .

2.2 Admissible Heuristic pada Algoritma A^*

Heuristik dikatakan admissible jika tidak pernah melebih-lebihkan (overestimate) biaya minimum yang dibutuhkan untuk mencapai tujuan dari suatu node. Dalam definisi formal, sebuah heuristik $h(n)$ disebut admissible jika untuk semua node n , berlaku $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah biaya sebenarnya dari node n ke tujuan. Algoritma A^* dijamin akan menemukan solusi optimal jika heuristik yang digunakan bersifat admissible, karena nilai $f(n)=g(n)+h(n)$ tidak akan terlalu kecil atau menyesatkan. Dalam konteks puzzle seperti *Rush Hour*, jika heuristik menghitung jumlah kendaraan yang menghalangi jalan keluar tanpa melebih-lebihkan hambatan sebenarnya, maka heuristik tersebut dapat dikatakan admissible.

2.3 Algoritma UCS dan BFS

Secara struktur, UCS dan Breadth-First Search (BFS) memang terlihat mirip, terutama dalam kasus-kasus di mana semua langkah antar-node memiliki bobot yang sama. Dalam kondisi seperti puzzle *Rush Hour* di mana setiap langkah memindahkan kendaraan hanya dihitung sebagai satu langkah dengan biaya yang seragam, UCS dan BFS akan membangkitkan node dalam urutan yang sama dan menghasilkan jalur solusi yang sama. Perbedaan utama terletak pada prinsip yang digunakan: UCS memilih node dengan biaya total terkecil $g(n)$, sedangkan BFS memilih node berdasarkan kedalaman level pencarian. Namun, ketika semua biaya transisi antar node adalah 1, maka $g(n)$ pada UCS setara dengan kedalaman, sehingga keduanya bertindak identik.

2.4 Perbandingan Efisiensi Algoritma A* dan UCS

Secara teoritis, algoritma A* lebih efisien dibandingkan dengan UCS dalam menyelesaikan puzzle seperti *Rush Hour*, selama heuristik yang digunakan bersifat admissible dan informatif. UCS menjelajahi semua node dengan urutan biaya terendah secara menyeluruh tanpa panduan arah, sehingga sering kali memproses banyak node yang tidak relevan terhadap tujuan. Sementara itu, A* menggunakan heuristik untuk memprioritaskan eksplorasi ke arah tujuan, sehingga dapat memotong sebagian besar ruang pencarian yang tidak diperlukan. Efisiensi ini terutama terlihat ketika ruang pencarian sangat besar atau dalam kasus di mana tujuan berada jauh dari node awal. Namun, jika heuristik yang digunakan buruk atau tidak admissible, A* bisa kehilangan keunggulannya dan menjadi tidak optimal atau bahkan lebih lambat dari UCS.

2.5 Optimalitas Algoritma GBFS

Algoritma Greedy Best-First Search (GBFS) tidak menjamin solusi optimal karena hanya mempertimbangkan estimasi jarak ke tujuan tanpa memperhatikan biaya yang telah dikeluarkan. Hal ini menyebabkan GBFS cenderung memilih jalur yang tampak paling menjanjikan secara lokal, meskipun bisa jadi jalur tersebut secara keseluruhan lebih mahal dibanding jalur lain yang tidak dipertimbangkan. Dalam konteks penyelesaian *Rush Hour*, GBFS dapat menemukan solusi dengan cepat, namun solusi tersebut belum tentu merupakan jalur tercepat atau paling efisien. Karena tidak menghitung $g(n)$, GBFS bisa melewati jalur panjang selama estimasi ke tujuan.

tampak pendek, sehingga solusinya tidak optimal dan kelengkapan pun tidak dijamin jika tidak ada mekanisme pelacakan node yang telah dikunjungi.

BAB III

SOURCE CODE DALAM BAHASA JAVA

3.1 Board.java

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

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

    public Board(File file) throws IOException {
        List<String> lines = new ArrayList<>();
        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
            String line;
            while ((line = br.readLine()) != null) {
                lines.add(line);
            }
        }

        if (lines.size() < 2) {
            throw new IllegalArgumentException("Format file tidak valid.");
        }

        // 1. Ukuran kolom dan baris
        String[] sizeParts = lines.get(0).trim().split(" ");
        if (sizeParts.length != 2) {
            throw new IllegalArgumentException("Format file tidak valid: baris pertama harus berisi '<rows> <cols>'");
        }
        try {
            this.rows = Integer.parseInt(sizeParts[0]);
            this.cols = Integer.parseInt(sizeParts[1]);
        } catch (NumberFormatException e) {
```

```

throw new IllegalArgumentException("Format file tidak valid: ukuran baris/kolom
bukan angka.");
}
if (this.rows <= 0 || this.cols <= 0) {
throw new IllegalArgumentException("Ukuran baris dan kolom harus positif.");
}

int declaredNPiece;
try {
declaredNPiece = Integer.parseInt(lines.get(1).trim()) + 1;
} catch (NumberFormatException e) {
throw new IllegalArgumentException("Format file tidak valid: jumlah bidak di baris
kedua bukan angka.");
}

this.exitRow = -2;
this.exitCol = -2;
List<int[]> kLocationsInFile = new ArrayList<>();

// if (lines.size() > 2 && lines.get(2).length() >= this.cols &&
lines.get(2).contains("K") && lines.indexOf(lines.get(2)) == 2) { // Cek apakah ini
baris untuk K di atas
// for(int j=0; j < lines.get(2).length(); j++){
// if(lines.get(2).charAt(j) == 'K') kLocationsInFile.add(new int[]{2,j});
// }
// }
int gridDataStartLineIndex = 2;

int actualGridContentStartLine = 2;
List<String> potentialGridLines = new ArrayList<>();
if (lines.size() > actualGridContentStartLine) {
boolean firstGridLineIsExit =
lines.get(actualGridContentStartLine).trim().chars().allMatch(c -> c == 'K' || c ==
' ');
if (firstGridLineIsExit && lines.get(actualGridContentStartLine).contains("K")) {
for (int j = 0; j < lines.get(actualGridContentStartLine).length(); j++) {
if (lines.get(actualGridContentStartLine).charAt(j) == 'K') {
kLocationsInFile.add(new int[]{actualGridContentStartLine, j, 1});
}
}
}
actualGridContentStartLine++;

```

```

}
}
for (int i = 0; i < this.rows; i++) {
if (lines.size() <= actualGridContentStartLine + i) break;
String currentLine = lines.get(actualGridContentStartLine + i);
// Kiri
if (currentLine.length() > 0 && currentLine.charAt(0) == 'K') {
kLocationsInFile.add(new int[]{actualGridContentStartLine + i, 0, 2});
}
if (currentLine.length() > this.cols && currentLine.charAt(this.cols) == 'K') {
kLocationsInFile.add(new int[]{actualGridContentStartLine + i, this.cols, 3});
}
}

if (lines.size() > actualGridContentStartLine + this.rows) {
String bottomLine = lines.get(actualGridContentStartLine + this.rows);
if (bottomLine.contains("K")) {
for (int j = 0; j < bottomLine.length(); j++) {
if (bottomLine.charAt(j) == 'K') {
kLocationsInFile.add(new int[]{actualGridContentStartLine + this.rows, j, 4});
}
}
}
}

if (kLocationsInFile.size() != 1) {
throw new IllegalArgumentException("Papan harus memiliki tepat satu pintu keluar (K). Ditemukan: " + kLocationsInFile.size());
}

int[] kPos = kLocationsInFile.get(0);
int kType = kPos[2];
int kFileLine = kPos[0];
int kFileCol = kPos[1];

int rowOffset = 0;
int colOffset = 0;

if (kType == 1) {
this.exitRow = -1;
this.exitCol = kFileCol;
}

```

```

rowOffset = kFileLine + 1 - 2;
if (kFileLine == 2) rowOffset = 1;
} else if (kType == 2) {
this.exitRow = kFileLine - (2 + rowOffset);
this.exitCol = -1;
if (kFileCol == 0) colOffset = 1;
} else if (kType == 3) {
this.exitRow = kFileLine - (2 + rowOffset);
this.exitCol = this.cols;
} else if (kType == 4) {
this.exitRow = this.rows;
this.exitCol = kFileCol - colOffset;
} else {
throw new IllegalArgumentException("Posisi pintu keluar K tidak valid atau tidak di
tepi.");
}

this.grid = new char[this.rows][this.cols];
this.pieces = new ArrayList<>();
Map<Character, List<int[]>> tempPieceCells = new HashMap<>();

for (int i = 0; i < this.rows; i++) {
int currentFileLineIndex = 2 + rowOffset + i;
if (currentFileLineIndex >= lines.size()) {
throw new IllegalArgumentException("Data grid tidak lengkap: baris tidak cukup untuk
ukuran " + this.rows + "x" + this.cols);
}
String gridContentLine = lines.get(currentFileLineIndex);
for (int j = 0; j < this.cols; j++) {
int currentColInFileLine = colOffset + j;
if (currentColInFileLine >= gridContentLine.length()) {
this.grid[i][j] = '.';
continue;
}
char c = gridContentLine.charAt(currentColInFileLine);
if (c == 'K') {
throw new IllegalArgumentException("Karakter 'K' tidak boleh berada di dalam area
grid permainan.");
}
this.grid[i][j] = c;
}

```

```

if (c != '.') {
tempPieceCells.putIfAbsent(c, new ArrayList<>());
tempPieceCells.get(c).add(new int[]{i, j});
}
}
}

// 3. Jumlah primary (P)
if (!tempPieceCells.containsKey('P')) {
throw new IllegalArgumentException("Bidak utama 'P' tidak ditemukan di grid.");
}

for (Map.Entry<Character, List<int[]>> entry : tempPieceCells.entrySet()) {
char pieceName = entry.getKey();
List<int[]> cellLocations = entry.getValue();
Piece currentPieceObject = new Piece(pieceName);

for (int[] cell : cellLocations) {
currentPieceObject.addCell(cell[0], cell[1]);
}
determineAndSetOrientation(currentPieceObject);

// 4. Validasi Bentuk mobil
if (currentPieceObject.getOrientation() == PieceOrientation.SINGLE_BLOCK) {
throw new IllegalArgumentException("Bidak '" + pieceName + "' tidak valid: tidak
boleh berukuran 1x1.");
}
if (currentPieceObject.getOrientation() == PieceOrientation.OTHER) {
throw new IllegalArgumentException("Bidak '" + pieceName + "' tidak valid: harus
berbentuk garis lurus (Nx1 atau 1xN). Ditemukan: " +
currentPieceObject.getOrientation());
}
}

this.pieces.add(currentPieceObject);
if (pieceName == 'P') {
if (this.primaryPiece != null) {
// Seharusnya tidak terjadi jika P dikelompokkan oleh tempPieceCells
throw new IllegalArgumentException("Lebih dari satu definisi bidak utama 'P'
ditemukan.");
}
this.primaryPiece = currentPieceObject;
}
}

```

```

}

// 5. Validasi Jumlah pieces
if (this.pieces.size() != declaredNPiece) {
throw new IllegalArgumentException("Jumlah bidak yang dideklarasikan (" +
declaredNPiece +
") tidak sesuai dengan jumlah bidak unik yang ditemukan di grid (" +
this.pieces.size() + ").");
}

// 6. Validasi Primary harus sejajar dengan pintu keluar
if (this.primaryPiece == null) {
throw new IllegalArgumentException("Bidak utama 'P' diperlukan untuk validasi
keselarasan dengan pintu keluar.");
}
if (this.exitRow == -2 || this.exitCol == -2) {
throw new IllegalArgumentException("Posisi Pintu Keluar 'K' tidak berhasil
ditentukan secara valid untuk pemeriksaan keselarasan.");
}

PieceOrientation pOrientation = this.primaryPiece.getOrientation();
boolean aligned = false;
if (pOrientation == PieceOrientation.HORIZONTAL) {
int pRow = this.primaryPiece.cells.get(0)[0];
if (this.exitRow == pRow) {
aligned = true;
}
} else if (pOrientation == PieceOrientation.VERTICAL) {
int pCol = this.primaryPiece.cells.get(0)[1];
if (this.exitCol == pCol) {
aligned = true;
}
}

if (!aligned) {
String pPosInfo = (pOrientation == PieceOrientation.HORIZONTAL) ?
"baris " + this.primaryPiece.cells.get(0)[0] :
"kolom " + this.primaryPiece.cells.get(0)[1];
String kPosInfo = String.format("baris=%d, kolom=%d", this.exitRow, this.exitCol);
String expectedKPos = (pOrientation == PieceOrientation.HORIZONTAL) ?

```

```

"pada baris " + this.primaryPiece.cells.get(0)[0] :
"pada kolom " + this.primaryPiece.cells.get(0)[1];

throw new IllegalArgumentException(
String.format("Bidak utama 'P' (orientasi: %s, posisi: %s) tidak sejajar dengan
pintu keluar 'K' (posisi: %s). 'K' seharusnya berada %s.",
pOrientation, pPosInfo, kPosInfo, expectedKPos
));
}
}

// Konstruktor private untuk clone
private Board() {}

private void determineAndSetOrientation(Piece piece) {
if (piece.cells.isEmpty()) {
piece.setOrientation(PieceOrientation.OTHER);
return;
}
if (piece.cells.size() == 1) {
piece.setOrientation(PieceOrientation.SINGLE_BLOCK);
return;
}

Set<Integer> uniqueRows = new HashSet<>();
Set<Integer> uniqueCols = new HashSet<>();
int minRow = Integer.MAX_VALUE, maxRow = Integer.MIN_VALUE;
int minCol = Integer.MAX_VALUE, maxCol = Integer.MIN_VALUE;

for (int[] cell : piece.cells) {
uniqueRows.add(cell[0]);
uniqueCols.add(cell[1]);
minRow = Math.min(minRow, cell[0]);
maxRow = Math.max(maxRow, cell[0]);
minCol = Math.min(minCol, cell[1]);
maxCol = Math.max(maxCol, cell[1]);
}

if (uniqueRows.size() == 1) {
if ((maxCol - minCol + 1) == piece.cells.size() && uniqueCols.size() ==
piece.cells.size()) {
piece.setOrientation(PieceOrientation.HORIZONTAL);

```

```

    } else {
piece.setOrientation(PieceOrientation.OTHER);
    }
    } else if (uniqueCols.size() == 1) {
if ((maxRow - minRow + 1) == piece.cells.size() && uniqueRows.size() ==
piece.cells.size()) {
piece.setOrientation(PieceOrientation.VERTICAL);
    } else {
piece.setOrientation(PieceOrientation.OTHER);
    }
    } else {
piece.setOrientation(PieceOrientation.OTHER);
    }
    }

public void print() {
if (this.exitRow == -1) {
for (int j = 0; j < this.cols; j++) {
System.out.print(j == this.exitCol ? 'K' : ' ');
}
System.out.println();
}

for (int i = 0; i < rows; i++) {
if (this.exitCol == -1 && this.exitRow == i) {
System.out.print('K');
    } else if (this.exitCol == -1) {
System.out.print(' ');
    }

for (int j = 0; j < cols; j++) {
System.out.print(grid[i][j]);
}

if (this.exitCol == this.cols && this.exitRow == i) {
System.out.print('K');
    }
System.out.println();
}

if (this.exitRow == this.rows) {
for (int j = 0; j < this.cols; j++) {

```



```

System.out.print(j == this.exitCol ? 'K' : ' ');
}
System.out.println();
}

if (primaryPiece != null && primaryPiece.cells != null &&
!primaryPiece.cells.isEmpty()) {
System.out.println("Primary piece (P) at: " +
Arrays.deepToString(primaryPiece.cells.toArray()) +
" Orientation: " + primaryPiece.getOrientation());
} else {
System.out.println("Primary piece (P) not found or has no cells.");
}
System.out.println("Exit at: (row=" + exitRow + ", col=" + exitCol + ") relative to
grid (0-indexed, -1 or size means border)");
}

@Override
public Board clone() {
Board copy = new Board();
copy.rows = this.rows;
copy.cols = this.cols;
copy.grid = new char[rows][cols];
for (int i = 0; i < rows; i++) {
copy.grid[i] = Arrays.copyOf(this.grid[i], cols);
}

copy.pieces = new ArrayList<>();
for (Piece p : this.pieces) {
Piece clonedPiece = p.clone();
copy.pieces.add(clonedPiece);
if (clonedPiece.name == 'P') {
copy.primaryPiece = clonedPiece;
}
}

if (this.primaryPiece != null && copy.primaryPiece == null) {
for(Piece p : copy.pieces) {
if (p.name == this.primaryPiece.name) { // Seharusnya 'P'
copy.primaryPiece = p;
break;
}
}
}

```

```
}  
}  
  
copy.exitRow = this.exitRow;  
copy.exitCol = this.exitCol;  
return copy;  
}  
}
```

3.2 Move.java

```
public class Move {  
    public char piece;  
    public char direction;  
    public Board board;  
  
    public Move(char piece, char direction, Board board) {  
        this.piece = piece;  
        this.direction = direction;  
        this.board = board;  
    }  
}
```

3.3 Piece.java

```
import java.util.ArrayList;  
import java.util.List;  
  
public class Piece {  
    public char name;  
    public List<int[]> cells;  
    public PieceOrientation orientation; // Field untuk menyimpan orientasi  
  
    public Piece(char name) {  
        this.name = name;
```

```

this.cells = new ArrayList<>();
// Inisialisasi default, akan di-set kemudian di Board.java
this.orientation = PieceOrientation.OTHER;
}

public void addCell(int row, int col) {
cells.add(new int[]{row, col});
}

// Setter untuk orientasi
public void setOrientation(PieceOrientation orientation) {
this.orientation = orientation;
}

// Getter untuk orientasi
public PieceOrientation getOrientation() {
return this.orientation;
}

@Override
public Piece clone() {
Piece copy = new Piece(this.name);
for (int[] cell : this.cells) {
copy.addCell(cell[0], cell[1]);
}
copy.setOrientation(this.orientation); // Salin orientasi saat cloning
return copy;
}
}

```

3.4 PieceOrientation.java

```

public enum PieceOrientation {
HORIZONTAL,
VERTICAL,
SINGLE_BLOCK,
OTHER
}

```

3.5 Solver.java

```
import java.util.List;

public interface Solver {
    void solve(Board start);

    public List<Board> solveAndReturnPath(Board start);

    int getLastSummarizedStepCount();
    int getNodesExplored();
}
```

3.6 Main.java

```
import java.io.File;
import java.io.IOException;
import java.util.InputMismatchException;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            // Input
            System.out.print("Masukkan nama file (test/input/): ");
            String filePath = "test/input" + scanner.nextLine();

            if (args.length > 0) {
                filePath = args[0];
            }

            File file = new File(filePath);
            Board board = new Board(file);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InputMismatchException e) {
            e.printStackTrace();
        }
    }
}
```

```

System.out.println("Board awal:");
board.print();
System.out.println();

int algoChoice;
while (true) {
    System.out.println("Pilih algoritma:");
    System.out.println("1. Greedy Best First Search (GBFS)");
    System.out.println("2. Uniform Cost Search (UCS)");
    System.out.println("3. A* Search");
    System.out.println("4. IDA* Search");
    System.out.print("Pilihan Anda (1-4): ");
    try {
        algoChoice = scanner.nextInt();
        if (algoChoice >= 1 && algoChoice <= 4) {
            break;
        } else {
            System.out.println("Pilihan tidak valid. Masukkan angka antara 1 dan 4.");
        }
    }
    catch (InputMismatchException e) {
        System.out.println("Input tidak valid. Harap masukkan angka.");
        scanner.next();
    }
    System.out.println();
}

Solver solver = null;
int heuristicChoice = -1;
String algorithmName = "";

if (algoChoice == 2) {
    algorithmName = "Uniform Cost Search (UCS)";
    solver = new UCS();
}
else {
    while (true) {
        System.out.println("\nPilih heuristik:");
        System.out.println("1. Manhattan Distance");
        System.out.println("2. Euclidean Distance");
        System.out.println("3. Obstacle-aware Distance");
    }
}

```

```

System.out.print("Pilihan Anda (1-3): ");
try {
    heuristicChoice = scanner.nextInt();
    if (heuristicChoice >= 1 && heuristicChoice <= 3) {
        break;
    } else {
        System.out.println("Pilihan tidak valid. Masukkan angka antara 1 dan 3.");
    }
} catch (InputMismatchException e) {
    System.out.println("Input tidak valid. Harap masukkan angka.");
    scanner.next();
}
System.out.println();
}

int heuristicIndex = heuristicChoice - 1;

switch (algoChoice) {
    case 1:
        algorithmName = "Greedy Best First Search (GBFS)";
        solver = new GBFS(heuristicIndex);
        break;
    case 3:
        algorithmName = "A* Search";
        solver = new AStar(heuristicIndex);
        break;
    case 4:
        algorithmName = "IDA* Search";
        solver = new IDAStar(heuristicIndex);
        break;
}

if (solver == null) {
    System.err.println("Error: Solver tidak terinisialisasi dengan benar. Program akan berhenti.");
    return;
}

System.out.println("\nMencari solusi...");
long startTime = System.currentTimeMillis();

```

```

solver.solve(board);
long endTime = System.currentTimeMillis();
System.out.println("Waktu eksekusi: " + (endTime - startTime) + " ms");
System.out.println("Algoritma: " + algorithmName);
}

catch (IOException e) {
System.err.println("Error membaca file: " + e.getMessage());
}

catch (Exception e) {
System.err.println("error: " + e.getMessage());
e.printStackTrace();
}

finally {
if (scanner != null) {
scanner.close(); //tutup scanner
}
}
}
}
}

```

3.7 AStar.java

```

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.*;

public class AStar implements Solver {
private final char[] priorityDirs = {'U', 'D', 'L', 'R'};
private int heuristicType;
public int nodesExpanded = 0;
public int lastSummarizedStepCount = 0;

```

```

private static class SummarizedStep {
    char piece;
    char direction;
    int moveCount;
    Board boardState;
    int g;
    int h;
    // f = g + h

    public SummarizedStep(char piece, char direction, int moveCount, Board boardState,
        int g, int h) {
        this.piece = piece;
        this.direction = direction;
        this.moveCount = moveCount;
        this.boardState = boardState;
        this.g = g;
        this.h = h;
    }

    public String getDisplay(String dirName) {
        return piece + "-" + dirName + (moveCount > 1 ? " " + moveCount + " kali" : "") +
            " (g=" + g + ", h=" + h + ", f=" + (g + h) + ")";
    }
}

public AStar(int heuristicType) {
    this.heuristicType = heuristicType;
}

private String getAlgorithmName() {
    return "A* Search";
}

private boolean needsHeuristic() {
    return true;
}

private void ensureTestDirectoryExists() {
    File directory = new File("test");
    if (!directory.exists()) {
        directory.mkdirs();
    }
}

```



```

}
}

@Override
public void solve(Board start) {
    nodesExpanded = 0;
    lastSummarizedStepCount = 0;

    PriorityQueue<Node> openSet = new PriorityQueue<>();
    Set<String> closedSet = new HashSet<>();
    Map<String, Integer> bestCost = new HashMap<>();

    int h = Heuristic.calculate(start, heuristicType);
    Node startNode = new Node(start, null, '\0', '\0', 0, h);
    openSet.add(startNode);
    String startKey = getBoardKey(start);
    bestCost.put(startKey, 0);

    System.out.println(getAlgorithmName() + " dengan heuristik " +
        Heuristic.getName(heuristicType));

    Node solutionNode = null;

    while (!openSet.isEmpty()) {
        Node current = openSet.poll();
        nodesExpanded++;

        if (isGoalState(current.board)) {
            solutionNode = current;
            break;
        }

        String boardKey = getBoardKey(current.board);

        if (closedSet.contains(boardKey) && current.g >= bestCost.getOrDefault(boardKey,
            Integer.MAX_VALUE)) {
            continue;
        }

        if (current.g > bestCost.getOrDefault(boardKey, Integer.MAX_VALUE)) {
            continue;
        }
    }
}

```

```

closedSet.add(boardKey);
for (Piece piece : current.board.pieces) {
    for (char dir : priorityDirs) {
        if (canMove(current.board, piece.name, dir)) {
            Board newBoard = move(current.board, piece.name, dir);
            String newBoardKey = getBoardKey(newBoard);
            int newG = current.g + 1;

            if (newG < bestCost.getOrDefault(newBoardKey, Integer.MAX_VALUE)) {
                bestCost.put(newBoardKey, newG);
                int newH = Heuristic.calculate(newBoard, heuristicType);
                Node neighbor = new Node(newBoard, current, piece.name, dir, newG, newH);
                openSet.add(neighbor);
            }
        }
    }
}

if (solutionNode != null) {
    List<SummarizedStep> summarizedPath = getSummarizedPath(solutionNode);
    this.lastSummarizedStepCount = summarizedPath.size();
    printSummarizedSolution(summarizedPath);
} else {
    System.out.println("Tidak ditemukan solusi!");
    ensureTestDirectoryExists();
    try (PrintWriter writer = new PrintWriter(new FileWriter("test/output/output.txt",
        false))) {
        writer.println(getAlgorithmName() + " dengan heuristik " +
            Heuristic.getName(heuristicType));
        writer.println("Tidak ditemukan solusi!");
        writer.println("Node yang dieksplorasi: " + nodesExpanded);
    } catch (IOException e) {
        System.err.println("Gagal menulis ke file output.txt (solusi tidak ditemukan): " +
            e.getMessage());
    }
}

@Override
public List<Board> solveAndReturnPath(Board start) {

```

```

nodesExpanded = 0;
lastSummarizedStepCount = 0;
PriorityQueue<Node> openSet = new PriorityQueue<>();
Set<String> closedSet = new HashSet<>();
Map<String, Integer> bestCost = new HashMap<>();
int hVal = Heuristic.calculate(start, heuristicType);
Node startNode = new Node(start, null, '\\0', '\\0', 0, hVal);
openSet.add(startNode);
String startKey = getBoardKey(start);
bestCost.put(startKey, 0);
System.out.println(getAlgorithmName() + " dengan heuristik " +
Heuristic.getName(heuristicType) + " (mencari path list)");
Node solutionNode = null;
while (!openSet.isEmpty()) {
Node current = openSet.poll();
nodesExpanded++;
if (isGoalState(current.board)) {
solutionNode = current;
break;
}
String boardKey = getBoardKey(current.board);
if (closedSet.contains(boardKey) && current.g >= bestCost.getOrDefault(boardKey,
Integer.MAX_VALUE)) {
continue;
}
if (current.g > bestCost.getOrDefault(boardKey, Integer.MAX_VALUE)) {
continue;
}
closedSet.add(boardKey);
for (Piece piece : current.board.pieces) {
for (char dir : priorityDirs) {
if (canMove(current.board, piece.name, dir)) {
Board newBoard = move(current.board, piece.name, dir);
String newBoardKey = getBoardKey(newBoard);
int newG = current.g + 1;
if (newG < bestCost.getOrDefault(newBoardKey, Integer.MAX_VALUE)) {
bestCost.put(newBoardKey, newG);
int newH = Heuristic.calculate(newBoard, heuristicType);
Node neighbor = new Node(newBoard, current, piece.name, dir, newG, newH);
openSet.add(neighbor);
}
}
}
}

```

```

    }
    }
    }
    }

    if (solutionNode == null) {
        System.out.println("Tidak ditemukan solusi!");
        ensureTestDirectoryExists();
        try (PrintWriter writer = new PrintWriter(new FileWriter("test/output/output.txt",
            false))) {
            writer.println(getAlgorithmName() + " dengan heuristik " +
                Heuristic.getName(this.heuristicType));
            writer.println("Tidak ditemukan solusi!");
            writer.println("Node yang dieksplorasi: " + nodesExpanded);
        } catch (IOException e) {
            System.err.println("Gagal menulis ke file output.txt (solusi tidak ditemukan): " +
                e.getMessage());
        }
        return new ArrayList<>();
    }

    List<Board> boardPath = new ArrayList<>();
    Node tempNode = solutionNode;
    while (tempNode != null) {
        boardPath.add(tempNode.board);
        tempNode = tempNode.parent;
    }
    Collections.reverse(boardPath);
    List<SummarizedStep> summarizedPath = getSummarizedPath(solutionNode);
    this.lastSummarizedStepCount = summarizedPath.size();
    printSummarizedSolution(summarizedPath); // Cetak dan simpan solusi
    return boardPath;
}

@Override
public int getLastSummarizedStepCount() {
    return lastSummarizedStepCount;
}

@Override
public int getNodesExplored() {
    return nodesExpanded;
}

```

```

private List<SummarizedStep> getSummarizedPath(Node solutionNode) {
    List<SummarizedStep> summarizedSteps = new ArrayList<>();
    if (solutionNode == null) {
        return summarizedSteps;
    }

    List<Node> rawPathNodes = new ArrayList<>();
    Node current = solutionNode;
    while (current != null && current.parent != null) {
        rawPathNodes.add(current);
        current = current.parent;
    }
    Collections.reverse(rawPathNodes);

    if (rawPathNodes.isEmpty()) {
        return summarizedSteps;
    }

    char currentPiece = rawPathNodes.get(0).piece;
    char currentDirection = rawPathNodes.get(0).direction;
    int moveCount = 0;
    Board lastBoardInSequence = rawPathNodes.get(0).board;
    int gVal = rawPathNodes.get(0).g;
    int hVal = rawPathNodes.get(0).h;

    for (Node node : rawPathNodes) {
        if (node.piece == currentPiece && node.direction == currentDirection) {
            moveCount++;
            lastBoardInSequence = node.board;
            gVal = node.g;
            hVal = node.h;
        } else {
            summarizedSteps.add(new SummarizedStep(currentPiece, currentDirection, moveCount,
                lastBoardInSequence, gVal, hVal));
            currentPiece = node.piece;
            currentDirection = node.direction;
            moveCount = 1;
            lastBoardInSequence = node.board;
            gVal = node.g;
            hVal = node.h;
        }
    }
}

```

```

}
if (moveCount > 0) {
    summarizedSteps.add(new SummarizedStep(currentPiece, currentDirection, moveCount,
    lastBoardInSequence, gVal, hVal));
}
return summarizedSteps;
}

private void printSummarizedSolution(List<SummarizedStep> summarizedPath) {
    ensureTestDirectoryExists();
    try (PrintWriter writer = new PrintWriter(new
    FileWriter("test/output/outputAStar.txt", false))) {
        String algoHeader = getAlgorithmName() + " dengan heuristik " +
        Heuristic.getName(heuristicType);
        System.out.println("\n" + algoHeader); // Already printed by
        solve/solveAndReturnPath, but good for consistency if called standalone
        writer.println(algoHeader);
        writer.println("-----");

        int stepNumber = 1;
        for (SummarizedStep step : summarizedPath) {
            String stepDisplay = "Langkah " + stepNumber + ": " +
            step.getDisplay(getDirName(step.direction));
            System.out.println(stepDisplay);
            writer.println(stepDisplay);

            step.boardState.print(); // Prints to console
            // Write board state to file
            for (int r = 0; r < step.boardState.rows; r++) {
                for (int c = 0; c < step.boardState.cols; c++) {
                    writer.print(step.boardState.grid[r][c] + (c == step.boardState.cols - 1 ? " " : "
                    "));
                }
                writer.println();
            }
            System.out.println();
            writer.println();
            stepNumber++;
        }

        String summary = "Solusi ditemukan dalam " + summarizedPath.size() + " langkah
        (ringkas).";
    }
}

```

```

System.out.println(summary);
writer.println(summary);

String nodesExploredStr = "Node yang dieksplorasi: " + nodesExpanded;
System.out.println(nodesExploredStr);
writer.println(nodesExploredStr);

} catch (IOException e) {
System.err.println("Gagal menulis langkah solusi ke file outputAStar.txt: " +
e.getMessage());
}
}

private boolean isGoalState(Board board) {
if (board.primaryPiece == null || (board.exitRow == -1 && board.exitCol == -1))
return false;
for (int[] cell : board.primaryPiece.cells) {
int r = cell[0], c = cell[1];
// Exit di atas grid
if (board.exitRow == -1 && board.exitCol == c && r == 0) return true;
// Exit di bawah grid
if (board.exitRow == board.rows && board.exitCol == c && r == board.rows - 1) return
true;
// Exit di kiri grid
if (board.exitCol == -1 && board.exitRow == r && c == 0) return true;
// Exit di kanan grid
if (board.exitCol == board.cols && board.exitRow == r && c == board.cols - 1) return
true;
// Exit di dalam grid (sampling P)
if ((r == board.exitRow && Math.abs(c - board.exitCol) == 1) ||
(c == board.exitCol && Math.abs(r - board.exitRow) == 1)) {
return true;
}
}
return false;
}

private String getBoardKey(Board board) {
StringBuilder sb = new StringBuilder();
for (int i = 0; i < board.rows; i++) {
for (int j = 0; j < board.cols; j++) {

```

```
sb.append(board.grid[i][j]);
}
}
return sb.toString();
}

private String getDirName(char d) {
return switch (d) {
case 'L' -> "kiri";
case 'R' -> "kanan";
case 'U' -> "atas";
case 'D' -> "bawah";
default -> "?";
};
}

private boolean canMove(Board board, char pieceName, char dir) {
Piece pieceToMove = null;
for (Piece p : board.pieces) {
if (p.name == pieceName) {
pieceToMove = p;
break;
}
}

if (pieceToMove == null || pieceToMove.cells.isEmpty()) {
return false;
}

PieceOrientation orientation = pieceToMove.getOrientation();

if (orientation == PieceOrientation.HORIZONTAL) {
if (dir == 'U' || dir == 'D') {
return false;
}
} else if (orientation == PieceOrientation.VERTICAL) {
if (dir == 'L' || dir == 'R') {
return false;
}
}
}
```



```

for (int[] cell : pieceToMove.cells) {
    int currentRow = cell[0];
    int currentCol = cell[1];
    int newRow = currentRow;
    int newCol = currentCol;

    switch (dir) {
        case 'L' -> newCol--;
        case 'R' -> newCol++;
        case 'U' -> newRow--;
        case 'D' -> newRow++;
    }

    if (newRow < 0 || newRow >= board.rows || newCol < 0 || newCol >= board.cols) {
        return false;
    }

    char destinationCellContent = board.grid[newRow][newCol];
    boolean partOfItself = false;
    for(int[] ownCell : pieceToMove.cells){
        if(ownCell[0] == newRow && ownCell[1] == newCol){
            partOfItself = true;
            break;
        }
    }

    if (destinationCellContent != '.' && destinationCellContent != 'K' && !partOfItself)
    {
        return false;
    }
}
return true;
}

private Board move(Board board, char pieceName, char dir) {
    Board newBoard = board.clone();
    Piece targetPieceInNewBoard = null;
    for(Piece p : newBoard.pieces){
        if(p.name == pieceName){
            targetPieceInNewBoard = p;
            break;

```

```

}
}

if (targetPieceInNewBoard == null) return newBoard;

for (int[] cell : targetPieceInNewBoard.cells) {
newBoard.grid[cell[0]][cell[1]] = '.';
}

for (int[] cell : targetPieceInNewBoard.cells) {
switch (dir) {
case 'L' -> cell[1]--;
case 'R' -> cell[1]++;
case 'U' -> cell[0]--;
case 'D' -> cell[0]++;
}
}

for (int[] cell : targetPieceInNewBoard.cells) {
if (cell[0] >= 0 && cell[0] < newBoard.rows && cell[1] >=0 && cell[1] <
newBoard.cols) {
newBoard.grid[cell[0]][cell[1]] = pieceName;
} else {
System.err.println("Error critical (AStar): Piece " + pieceName + " moved out of
bounds. Row: " + cell[0] + ", Col: " + cell[1]);
return board;
}
}

char KODE_BIDAK_UTAMA = 'P';
if (pieceName == KODE_BIDAK_UTAMA) {
newBoard.primaryPiece = targetPieceInNewBoard;
}

return newBoard;
}

private static class Node implements Comparable<Node> {
Board board;
Node parent;
char piece;

```

```

char direction;
int g;
int h;

public Node(Board board, Node parent, char piece, char direction, int g, int h) {
    this.board = board;
    this.parent = parent;
    this.piece = piece;
    this.direction = direction;
    this.g = g;
    this.h = h;
}

@Override
public int compareTo(Node other) {
    int fThis = this.g + this.h;
    int fOther = other.g + other.h;
    if (fThis == fOther) { // Tie-breaking: prefer smaller h
        return Integer.compare(this.h, other.h);
    }
    return Integer.compare(fThis, fOther);
}
}
}
}

```

3.8 GBFS.java

```

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.*;

public class GBFS implements Solver {
    private final char[] priorityDirs = {'U', 'D', 'L', 'R'};
    private int heuristicType;
    public int nodesExpanded = 0;
    public int lastSummarizedStepCount = 0;
}

```

```

private static class SummarizedStep {
    char piece;
    char direction;
    int moveCount;
    Board boardState;
    int h;

    public SummarizedStep(char piece, char direction, int moveCount, Board boardState,
        int h) {
        this.piece = piece;
        this.direction = direction;
        this.moveCount = moveCount;
        this.boardState = boardState;
        this.h = h;
    }

    public String getDisplay(String dirName) {
        return piece + "-" + dirName + (moveCount > 1 ? " " + moveCount + " kali" : "") +
            " (h=" + h + ")";
    }
}

public GBFS() {
    this.heuristicType = 0; // Default Manhattan
}

public GBFS(int heuristicType) {
    this.heuristicType = heuristicType;
}

private String getAlgorithmName() {
    return "Greedy Best First Search (GBFS)";
}

private boolean needsHeuristic() {
    return true;
}

private void ensureTestDirectoryExists() {
    File directory = new File("test");
    if (!directory.exists()) {

```

```

directory.mkdirs();
}
}

@Override
public void solve(Board start) {
    nodesExpanded = 0;
    lastSummarizedStepCount = 0;

    PriorityQueue<Node> openSet = new PriorityQueue<>();
    Set<String> closedSet = new HashSet<>();

    int h = Heuristic.calculate(start, heuristicType);
    Node startNode = new Node(start, null, '\0', '\0', h);
    openSet.add(startNode);

    System.out.println(getAlgorithmName() + " dengan heuristik " +
        Heuristic.getName(heuristicType));

    Node solutionNode = null;

    while (!openSet.isEmpty()) {
        Node current = openSet.poll();
        nodesExpanded++;

        if (isGoalState(current.board)) {
            solutionNode = current;
            break;
        }

        String boardKey = getBoardKey(current.board);

        if (closedSet.contains(boardKey)) {
            continue;
        }
        closedSet.add(boardKey);

        for (Piece piece : current.board.pieces) {
            for (char dir : priorityDirs) {
                if (canMove(current.board, piece.name, dir)) {
                    Board newBoard = move(current.board, piece.name, dir);

```

```

String newBoardKey = getBoardKey(newBoard);

if (!closedSet.contains(newBoardKey)) {
    int newH = Heuristic.calculate(newBoard, heuristicType);
    Node neighbor = new Node(newBoard, current, piece.name, dir, newH);
    openSet.add(neighbor);
}
}
}
}
}

if (solutionNode != null) {
    List<SummarizedStep> summarizedPath = getSummarizedPath(solutionNode);
    this.lastSummarizedStepCount = summarizedPath.size();
    printSummarizedSolution(summarizedPath);
} else {
    System.out.println("Tidak ditemukan solusi!");
    ensureTestDirectoryExists();
    try (PrintWriter writer = new PrintWriter(new FileWriter("test/output/output.txt",
        false))) {
        writer.println(getAlgorithmName() + " dengan heuristik " +
            Heuristic.getName(heuristicType));
        writer.println("Tidak ditemukan solusi!");
        writer.println("Node yang dieksplorasi: " + nodesExpanded);
    } catch (IOException e) {
        System.err.println("Gagal menulis ke file output.txt (solusi tidak ditemukan): " +
            e.getMessage());
    }
}
}

@Override
public List<Board> solveAndReturnPath(Board start) {
    nodesExpanded = 0;
    lastSummarizedStepCount = 0;

    PriorityQueue<Node> openSet = new PriorityQueue<>();
    Set<String> closedSet = new HashSet<>();

    int hVal = Heuristic.calculate(start, heuristicType);

```

```

Node startNode = new Node(start, null, '\0', '\0', hVal);
openSet.add(startNode);

System.out.println(getAlgorithmName() + " dengan heuristik " +
Heuristic.getName(heuristicType) + " (mencari path list)");

Node solutionNode = null;

while (!openSet.isEmpty()) {
Node current = openSet.poll();
nodesExpanded++;

if (isGoalState(current.board)) {
solutionNode = current;
break;
}

String boardKey = getBoardKey(current.board);

if (closedSet.contains(boardKey)) {
continue;
}
closedSet.add(boardKey);

for (Piece piece : current.board.pieces) {
for (char dir : priorityDirs) {
if (canMove(current.board, piece.name, dir)) {
Board newBoard = move(current.board, piece.name, dir);
String newBoardKey = getBoardKey(newBoard);

if (!closedSet.contains(newBoardKey)) {
int newH = Heuristic.calculate(newBoard, heuristicType);
Node neighbor = new Node(newBoard, current, piece.name, dir, newH);
openSet.add(neighbor);
}
}
}
}

if (solutionNode == null) {

```

```

System.out.println("Tidak ditemukan solusi!");
ensureTestDirectoryExists();
try (PrintWriter writer = new PrintWriter(new FileWriter("test/output/outputB.txt",
false))) {
writer.println(getAlgorithmName() + " dengan heuristik " +
Heuristic.getName(this.heuristicType));
writer.println("Tidak ditemukan solusi!");
writer.println("Node yang dieksplorasi: " + nodesExpanded);
} catch (IOException e) {
System.err.println("Gagal menulis ke file outputB.txt (solusi tidak ditemukan): " +
e.getMessage());
}
return new ArrayList<>();
}

List<Board> boardPath = new ArrayList<>();
Node tempNode = solutionNode;
while (tempNode != null) {
boardPath.add(tempNode.board);
tempNode = tempNode.parent;
}
Collections.reverse(boardPath);

List<SummarizedStep> summarizedPath = getSummarizedPath(solutionNode);
this.lastSummarizedStepCount = summarizedPath.size();
printSummarizedSolution(summarizedPath); // Cetak dan simpan solusi
return boardPath;
}

@Override
public int getLastSummarizedStepCount() {
return lastSummarizedStepCount;
}

@Override
public int getNodesExplored() {
return nodesExpanded;
}

private List<SummarizedStep> getSummarizedPath(Node solutionNode) {
List<SummarizedStep> summarizedSteps = new ArrayList<>();

```



```

if (solutionNode == null) return summarizedSteps;

List<Node> rawPathNodes = new ArrayList<>();
Node current = solutionNode;
while (current != null && current.parent != null) {
    rawPathNodes.add(current);
    current = current.parent;
}
Collections.reverse(rawPathNodes);

if (rawPathNodes.isEmpty()) return summarizedSteps;

char currentPiece = rawPathNodes.get(0).piece;
char currentDirection = rawPathNodes.get(0).direction;
int moveCount = 0;
Board lastBoardInSequence = rawPathNodes.get(0).board;
int hVal = rawPathNodes.get(0).h;

for (Node node : rawPathNodes) {
    if (node.piece == currentPiece && node.direction == currentDirection) {
        moveCount++;
        lastBoardInSequence = node.board;
        hVal = node.h;
    } else {
        summarizedSteps.add(new SummarizedStep(currentPiece, currentDirection, moveCount,
            lastBoardInSequence, hVal));
        currentPiece = node.piece;
        currentDirection = node.direction;
        moveCount = 1;
        lastBoardInSequence = node.board;
        hVal = node.h;
    }
}

if (moveCount > 0) {
    summarizedSteps.add(new SummarizedStep(currentPiece, currentDirection, moveCount,
        lastBoardInSequence, hVal));
}
return summarizedSteps;
}

private void printSummarizedSolution(List<SummarizedStep> summarizedPath) {

```

```

ensureTestDirectoryExists();
try (PrintWriter writer = new PrintWriter(new
FileWriter("test/output/outputGBFS.txt", false))) {
String algoHeader = getAlgorithmName() + " dengan heuristik " +
Heuristic.getName(heuristicType);
System.out.println("\n" + algoHeader);
writer.println(algoHeader);
writer.println("-----");

int stepNumber = 1;
for (SummarizedStep step : summarizedPath) {
String stepDisplay = "Langkah " + stepNumber + ": " +
step.getDisplay(getDirName(step.direction));
System.out.println(stepDisplay);
writer.println(stepDisplay);

step.boardState.print(); // Prints to console
for (int r = 0; r < step.boardState.rows; r++) {
for (int c = 0; c < step.boardState.cols; c++) {
writer.print(step.boardState.grid[r][c] + (c == step.boardState.cols - 1 ? " " : "
"));
}
writer.println();
}
System.out.println();
writer.println();
stepNumber++;
}

String summary = "Solusi ditemukan dalam " + summarizedPath.size() + " langkah
(ringkas).";
System.out.println(summary);
writer.println(summary);

String nodesExploredStr = "Node yang dieksplorasi: " + nodesExpanded;
System.out.println(nodesExploredStr);
writer.println(nodesExploredStr);

} catch (IOException e) {
System.err.println("Gagal menulis langkah solusi ke file outputGBFS.txt: " +
e.getMessage());
}

```

```

}

private boolean isGoalState(Board board) {
    if (board.primaryPiece == null || (board.exitRow == -1 && board.exitCol == -1))
        return false;
    for (int[] cell : board.primaryPiece.cells) {
        int r = cell[0], c = cell[1];
        // Exit di atas grid
        if (board.exitRow == -1 && board.exitCol == c && r == 0) return true;
        // Exit di bawah grid
        if (board.exitRow == board.rows && board.exitCol == c && r == board.rows - 1) return
true;
        // Exit di kiri grid
        if (board.exitCol == -1 && board.exitRow == r && c == 0) return true;
        // Exit di kanan grid
        if (board.exitCol == board.cols && board.exitRow == r && c == board.cols - 1) return
true;
        // Exit di dalam grid (samping P)
        if ((r == board.exitRow && Math.abs(c - board.exitCol) == 1) ||
(c == board.exitCol && Math.abs(r - board.exitRow) == 1)) {
            return true;
        }
    }
    return false;
}

private String getBoardKey(Board board) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < board.rows; i++) {
        for (int j = 0; j < board.cols; j++) {
            sb.append(board.grid[i][j]);
        }
    }
    return sb.toString();
}

private String getDirName(char d) {
    return switch (d) {
        case 'L' -> "kiri";
        case 'R' -> "kanan";
        case 'U' -> "atas";
        case 'D' -> "bawah";
    }
}

```

```
default -> "?";
};
}

private boolean canMove(Board board, char pieceName, char dir) {
    Piece pieceToMove = null;
    for (Piece p : board.pieces) {
        if (p.name == pieceName) {
            pieceToMove = p;
            break;
        }
    }

    if (pieceToMove == null || pieceToMove.cells.isEmpty()) {
        return false;
    }

    PieceOrientation orientation = pieceToMove.getOrientation();

    if (orientation == PieceOrientation.HORIZONTAL) {
        if (dir == 'U' || dir == 'D') {
            return false;
        }
    } else if (orientation == PieceOrientation.VERTICAL) {
        if (dir == 'L' || dir == 'R') {
            return false;
        }
    }

    for (int[] cell : pieceToMove.cells) {
        int currentRow = cell[0];
        int currentCol = cell[1];
        int newRow = currentRow;
        int newCol = currentCol;

        switch (dir) {
            case 'L' -> newCol--;
            case 'R' -> newCol++;
            case 'U' -> newRow--;
            case 'D' -> newRow++;
        }
    }
}
```

```

if (newRow < 0 || newRow >= board.rows || newCol < 0 || newCol >= board.cols) {
return false;
}

char destinationCellContent = board.grid[newRow][newCol];
boolean partOfItself = false;
for(int[] ownCell : pieceToMove.cells){
if(ownCell[0] == newRow && ownCell[1] == newCol){
partOfItself = true;
break;
}
}

if (destinationCellContent != '.' && destinationCellContent != 'K' && !partOfItself)
{
return false;
}
}
return true;
}

private Board move(Board board, char pieceName, char dir) {
Board newBoard = board.clone();
Piece targetPieceInNewBoard = null;
for(Piece p : newBoard.pieces){
if(p.name == pieceName){
targetPieceInNewBoard = p;
break;
}
}

if (targetPieceInNewBoard == null) return newBoard; // Should not happen if canMove
is true

for (int[] cell : targetPieceInNewBoard.cells) {
newBoard.grid[cell[0]][cell[1]] = '.';
}

for (int[] cell : targetPieceInNewBoard.cells) {
switch (dir) {

```

```

case 'L' -> cell[1]--;
case 'R' -> cell[1]++;
case 'U' -> cell[0]--;
case 'D' -> cell[0]++;
}
}

for (int[] cell : targetPieceInNewBoard.cells) {
if (cell[0] >= 0 && cell[0] < newBoard.rows && cell[1] >=0 && cell[1] <
newBoard.cols) {
newBoard.grid[cell[0]][cell[1]] = pieceName;
} else {
System.err.println("Error critical (GBFS): Piece " + pieceName + " moved out of
bounds. Row: " + cell[0] + ", Col: " + cell[1]);
return board;
}
}

char KODE_BIDAK_UTAMA = 'P';
if (pieceName == KODE_BIDAK_UTAMA) {
newBoard.primaryPiece = targetPieceInNewBoard;
}
return newBoard;
}

private static class Node implements Comparable<Node> {
Board board;
Node parent;
char piece;
char direction;
int h;

public Node(Board board, Node parent, char piece, char direction, int h) {
this.board = board;
this.parent = parent;
this.piece = piece;
this.direction = direction;
this.h = h;
}

@Override
public int compareTo(Node other) {

```

```
return Integer.compare(this.h, other.h);
}
}
}
```

3.9 Heuristic.java

```
public class Heuristic {
    // Tipe heuristik
    public static final int MANHATTAN = 0;
    public static final int EUCLIDEAN = 1;
    public static final int OBSTACLE_AWARE = 2;
    public static int calculate(Board board, int type) {
        switch (type) {
            case MANHATTAN:
                return calculateManhattan(board);
            case EUCLIDEAN:
                return calculateEuclidean(board);
            case OBSTACLE_AWARE:
                return calculateObstacleAware(board);
            default:
                return calculateManhattan(board);
        }
    }

    public static String getName(int type) {
        return switch (type) {
            case MANHATTAN -> "Manhattan Distance";
            case EUCLIDEAN -> "Euclidean Distance";
            case OBSTACLE_AWARE -> "Obstacle-aware Distance";
            default -> "Unknown";
        };
    }

    private static int calculateManhattan(Board board) {
        int minDistance = Integer.MAX_VALUE;
        for (int[] cell : board.primaryPiece.cells) {
            int distance = Math.abs(cell[0] - board.exitRow) + Math.abs(cell[1] -
```

```

board.exitCol);
minDistance = Math.min(minDistance, distance);
}
return minDistance;
}

private static int calculateEuclidean(Board board) {
double minDistance = Double.MAX_VALUE;
for (int[] cell : board.primaryPiece.cells) {
double dx = cell[0] - board.exitRow;
double dy = cell[1] - board.exitCol;
double distance = Math.sqrt(dx * dx + dy * dy);
minDistance = Math.min(minDistance, distance);
}
return (int) Math.ceil(minDistance);
}

private static int calculateObstacleAware(Board board) {
int baseDistance = calculateManhattan(board);
int obstacles = 0;
int[] closestCell = null;
int minDistance = Integer.MAX_VALUE;
for (int[] cell : board.primaryPiece.cells) {
int distance = Math.abs(cell[0] - board.exitRow) + Math.abs(cell[1] -
board.exitCol);
if (distance < minDistance) {
minDistance = distance;
closestCell = cell;
}
}
if (closestCell == null) return baseDistance;
int row = closestCell[0];
int col = closestCell[1];
int exRow = board.exitRow;
int exCol = board.exitCol;
// Vertical path
if (col == exCol) {
int start = Math.min(row, exRow);
int end = Math.max(row, exRow);
for (int i = start; i <= end; i++) {
if (i >= 0 && i < board.rows && col >= 0 && col < board.cols) {

```



```

char ch = board.grid[i][col];
if (ch != '.' && ch != 'P' && ch != 'K') obstacles++;
}
}
}

// Horizontal path
else if (row == exRow) {
int start = Math.min(col, exCol);
int end = Math.max(col, exCol);
for (int j = start; j <= end; j++) {
if (row >= 0 && row < board.rows && j >= 0 && j < board.cols) {
char ch = board.grid[row][j];
if (ch != '.' && ch != 'P' && ch != 'K') obstacles++;
}
}
}

// Diagonal/umum (cek kotak dari primary ke exit)
else {
int minRow = Math.min(row, exRow);
int maxRow = Math.max(row, exRow);
int minCol = Math.min(col, exCol);
int maxCol = Math.max(col, exCol);
for (int i = minRow; i <= maxRow; i++) {
for (int j = minCol; j <= maxCol; j++) {
if (i >= 0 && i < board.rows && j >= 0 && j < board.cols) {
char ch = board.grid[i][j];
if (ch != '.' && ch != 'P' && ch != 'K') obstacles++;
}
}
}
}
return baseDistance + obstacles * 2;
}
}

```

3.10 UCS.java

```

import java.io.File;

```

```

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.*;

public class UCS implements Solver {
    private final char[] priorityDirs = {'U', 'D', 'L', 'R'};
    public int nodesExpanded = 0;
    public int lastSummarizedStepCount = 0;

    private static class SummarizedStep {
        char piece;
        char direction;
        int moveCount;
        Board boardState;
        int g; // Cost
        public SummarizedStep(char piece, char direction, int moveCount, Board boardState,
            int g) {
            this.piece = piece;
            this.direction = direction;
            this.moveCount = moveCount;
            this.boardState = boardState;
            this.g = g;
        }

        public String getDisplay(String dirName) {
            return piece + "-" + dirName + (moveCount > 1 ? " " + moveCount + " kali" : "") +
                " (cost=" + g + ")";
        }
    }

    private String getAlgorithmName() {
        return "Uniform Cost Search (UCS)";
    }

    private boolean needsHeuristic() {
        return false;
    }

    private void ensureTestDirectoryExists() {
        File directory = new File("test");
        if (!directory.exists()) {

```

```

directory.mkdirs();
}
}

@Override
public void solve(Board start) {
    nodesExpanded = 0;
    lastSummarizedStepCount = 0;

    PriorityQueue<Node> openSet = new PriorityQueue<>();
    Map<String, Integer> bestCost = new HashMap<>();

    Node startNode = new Node(start, null, '\\0', '\\0', 0);
    openSet.add(startNode);
    String startKey = getBoardKey(start);
    bestCost.put(startKey, 0);

    System.out.println(getAlgorithmName());

    Node solutionNode = null;

    while (!openSet.isEmpty()) {
        Node current = openSet.poll();
        nodesExpanded++;

        if (isGoalState(current.board)) {
            solutionNode = current;
            break;
        }

        String boardKey = getBoardKey(current.board);
        if (current.g > bestCost.getOrDefault(boardKey, Integer.MAX_VALUE)) {
            continue;
        }

        for (Piece piece : current.board.pieces) {
            for (char dir : priorityDirs) {
                if (canMove(current.board, piece.name, dir)) {
                    Board newBoard = move(current.board, piece.name, dir);
                    String newBoardKey = getBoardKey(newBoard);
                    int newG = current.g + 1;

```

```

if (newG < bestCost.getOrDefault(newBoardKey, Integer.MAX_VALUE)) {
    bestCost.put(newBoardKey, newG);
    Node neighbor = new Node(newBoard, current, piece.name, dir, newG);
    openSet.add(neighbor);
}
}
}
}
}

if (solutionNode != null) {
    List<SummarizedStep> summarizedPath = getSummarizedPath(solutionNode);
    this.lastSummarizedStepCount = summarizedPath.size();
    printSummarizedSolution(summarizedPath);
} else {
    System.out.println("Tidak ditemukan solusi!");
    ensureTestDirectoryExists();
    try (PrintWriter writer = new PrintWriter(new FileWriter("test/output/output.txt",
        false))) {
        writer.println(getAlgorithmName());
        writer.println("Tidak ditemukan solusi!");
        writer.println("Node yang dieksplorasi: " + nodesExpanded);
    } catch (IOException e) {
        System.err.println("Gagal menulis ke file output.txt (solusi tidak ditemukan): " +
            e.getMessage());
    }
}

@Override
public List<Board> solveAndReturnPath(Board start) {
    nodesExpanded = 0;
    lastSummarizedStepCount = 0;

    PriorityQueue<Node> openSet = new PriorityQueue<>();
    Map<String, Integer> bestCost = new HashMap<>();

    Node startNode = new Node(start, null, '\0', '\0', 0);
    openSet.add(startNode);
    String startKey = getBoardKey(start);
    bestCost.put(startKey, 0);

```

```

System.out.println(getAlgorithmName() + " (mencari path list)");

Node solutionNode = null;

while (!openSet.isEmpty()) {
Node current = openSet.poll();
nodesExpanded++;

if (isGoalState(current.board)) {
solutionNode = current;
break;
}

String boardKey = getBoardKey(current.board);
if (current.g > bestCost.getOrDefault(boardKey, Integer.MAX_VALUE)) {
continue;
}

for (Piece piece : current.board.pieces) {
for (char dir : priorityDirs) {
if (canMove(current.board, piece.name, dir)) {
Board newBoard = move(current.board, piece.name, dir);
String newBoardKey = getBoardKey(newBoard);
int newG = current.g + 1;

if (newG < bestCost.getOrDefault(newBoardKey, Integer.MAX_VALUE)) {
bestCost.put(newBoardKey, newG);
Node neighbor = new Node(newBoard, current, piece.name, dir, newG);
openSet.add(neighbor);
}
}
}
}
}

if (solutionNode == null) {
System.out.println("Tidak ditemukan solusi!");
ensureTestDirectoryExists();
try (PrintWriter writer = new PrintWriter(new FileWriter("test/output/output.txt",
false))) {

```

```

writer.println(getAlgorithmName());
writer.println("Tidak ditemukan solusi!");
writer.println("Node yang dieksplorasi: " + nodesExpanded);
} catch (IOException e) {
System.err.println("Gagal menulis ke file output.txt (solusi tidak ditemukan): " +
e.getMessage());
}
return new ArrayList<>();
}

List<Board> boardPath = new ArrayList<>();
Node tempNode = solutionNode;
while (tempNode != null) {
boardPath.add(tempNode.board);
tempNode = tempNode.parent;
}
Collections.reverse(boardPath);

List<SummarizedStep> summarizedPath = getSummarizedPath(solutionNode);
this.lastSummarizedStepCount = summarizedPath.size();
printSummarizedSolution(summarizedPath); // Cetak dan simpan solusi
return boardPath;
}

@Override
public int getLastSummarizedStepCount() {
return lastSummarizedStepCount;
}

@Override
public int getNodesExplored() {
return nodesExpanded;
}

private List<SummarizedStep> getSummarizedPath(Node solutionNode) {
List<SummarizedStep> summarizedSteps = new ArrayList<>();
if (solutionNode == null) return summarizedSteps;

List<Node> rawPathNodes = new ArrayList<>();
Node current = solutionNode;
while (current != null && current.parent != null) {

```

```

rawPathNodes.add(current);
current = current.parent;
}
Collections.reverse(rawPathNodes);

if (rawPathNodes.isEmpty()) return summarizedSteps;

char currentPiece = rawPathNodes.get(0).piece;
char currentDirection = rawPathNodes.get(0).direction;
int moveCount = 0;
Board lastBoardInSequence = rawPathNodes.get(0).board;
int gVal = rawPathNodes.get(0).g;

for (Node node : rawPathNodes) {
    if (node.piece == currentPiece && node.direction == currentDirection) {
        moveCount++;
        lastBoardInSequence = node.board;
        gVal = node.g;
    } else {
        summarizedSteps.add(new SummarizedStep(currentPiece, currentDirection, moveCount,
            lastBoardInSequence, gVal));
        currentPiece = node.piece;
        currentDirection = node.direction;
        moveCount = 1;
        lastBoardInSequence = node.board;
        gVal = node.g;
    }
}

if (moveCount > 0) {
    summarizedSteps.add(new SummarizedStep(currentPiece, currentDirection, moveCount,
        lastBoardInSequence, gVal));
}

return summarizedSteps;
}

private void printSummarizedSolution(List<SummarizedStep> summarizedPath) {
    ensureTestDirectoryExists();
    try (PrintWriter writer = new PrintWriter(new
        FileWriter("test/output/outputUCS.txt", false))) {
        String algoHeader = getAlgorithmName();
        System.out.println("\n" + algoHeader);
    }
}

```

```

writer.println(algoHeader);
writer.println("-----");

int stepNumber = 1;
for (SummarizedStep step : summarizedPath) {
    String stepDisplay = "Langkah " + stepNumber + ": " +
        step.getDisplay(getDirName(step.direction));
    System.out.println(stepDisplay);
    writer.println(stepDisplay);

    step.boardState.print();
    for (int r = 0; r < step.boardState.rows; r++) {
        for (int c = 0; c < step.boardState.cols; c++) {
            writer.print(step.boardState.grid[r][c] + (c == step.boardState.cols - 1 ? " " : "
"));
        }
        writer.println();
    }
    System.out.println();
    writer.println();
    stepNumber++;
}

String summary = "Solusi ditemukan dalam " + summarizedPath.size() + " langkah
    (ringkas).";
System.out.println(summary);
writer.println(summary);

String nodesExploredStr = "Node yang dieksplorasi: " + nodesExpanded;
System.out.println(nodesExploredStr);
writer.println(nodesExploredStr);

} catch (IOException e) {
    System.err.println("Gagal menulis langkah solusi ke file outputUCS.txt: " +
        e.getMessage());
}
}

private boolean isGoalState(Board board) {
    if (board.primaryPiece == null || (board.exitRow == -1 && board.exitCol == -1))
        return false;
    for (int[] cell : board.primaryPiece.cells) {
        int r = cell[0], c = cell[1];

```



```

// Exit di atas grid
if (board.exitRow == -1 && board.exitCol == c && r == 0) return true;
// Exit di bawah grid
if (board.exitRow == board.rows && board.exitCol == c && r == board.rows - 1) return
true;
// Exit di kiri grid
if (board.exitCol == -1 && board.exitRow == r && c == 0) return true;
// Exit di kanan grid
if (board.exitCol == board.cols && board.exitRow == r && c == board.cols - 1) return
true;
// Exit di dalam grid (samping P)
if ((r == board.exitRow && Math.abs(c - board.exitCol) == 1) ||
(c == board.exitCol && Math.abs(r - board.exitRow) == 1)) {
return true;
}
}
return false;
}

private String getBoardKey(Board board) {
StringBuilder sb = new StringBuilder();
for (int i = 0; i < board.rows; i++) {
for (int j = 0; j < board.cols; j++) {
sb.append(board.grid[i][j]);
}
}
return sb.toString();
}

private String getDirName(char d) {
return switch (d) {
case 'L' -> "kiri";
case 'R' -> "kanan";
case 'U' -> "atas";
case 'D' -> "bawah";
default -> "?";
};
}

private boolean canMove(Board board, char pieceName, char dir) {
Piece pieceToMove = null;

```

```
for (Piece p : board.pieces) {
    if (p.name == pieceName) {
        pieceToMove = p;
        break;
    }
}

if (pieceToMove == null || pieceToMove.cells.isEmpty()) {
    return false;
}

PieceOrientation orientation = pieceToMove.getOrientation();

if (orientation == PieceOrientation.HORIZONTAL) {
    if (dir == 'U' || dir == 'D') {
        return false;
    }
} else if (orientation == PieceOrientation.VERTICAL) {
    if (dir == 'L' || dir == 'R') {
        return false;
    }
}

for (int[] cell : pieceToMove.cells) {
    int currentRow = cell[0];
    int currentCol = cell[1];
    int newRow = currentRow;
    int newCol = currentCol;

    switch (dir) {
        case 'L' -> newCol--;
        case 'R' -> newCol++;
        case 'U' -> newRow--;
        case 'D' -> newRow++;
    }

    if (newRow < 0 || newRow >= board.rows || newCol < 0 || newCol >= board.cols) {
        return false;
    }

    char destinationCellContent = board.grid[newRow][newCol];
```

```

boolean partOfItself = false;
for(int[] ownCell : pieceToMove.cells){
    if(ownCell[0] == newRow && ownCell[1] == newCol){
        partOfItself = true;
        break;
    }
}

if (destinationCellContent != '.' && destinationCellContent != 'K' && !partOfItself)
{
    return false;
}
return true;
}

private Board move(Board board, char pieceName, char dir) {
    Board newBoard = board.clone();
    Piece targetPieceInNewBoard = null;
    for(Piece p : newBoard.pieces){
        if(p.name == pieceName){
            targetPieceInNewBoard = p;
            break;
        }
    }

    if (targetPieceInNewBoard == null) return newBoard;

    for (int[] cell : targetPieceInNewBoard.cells) {
        newBoard.grid[cell[0]][cell[1]] = '.';
    }

    for (int[] cell : targetPieceInNewBoard.cells) {
        switch (dir) {
            case 'L' -> cell[1]--;
            case 'R' -> cell[1]++;
            case 'U' -> cell[0]--;
            case 'D' -> cell[0]++;
        }
    }
}

```

```

for (int[] cell : targetPieceInNewBoard.cells) {
if (cell[0] >= 0 && cell[0] < newBoard.rows && cell[1] >=0 && cell[1] <
newBoard.cols) {
newBoard.grid[cell[0]][cell[1]] = pieceName;
} else {
System.err.println("Error critical (UCS): Piece " + pieceName + " moved out of
bounds. Row: " + cell[0] + ", Col: " + cell[1]);
return board;
}
}

char KODE_BIDAK_UTAMA = 'P';
if (pieceName == KODE_BIDAK_UTAMA) {
newBoard.primaryPiece = targetPieceInNewBoard;
}
return newBoard;
}

private static class Node implements Comparable<Node> {
Board board;
Node parent;
char piece;
char direction;
int g;

public Node(Board board, Node parent, char piece, char direction, int g) {
this.board = board;
this.parent = parent;
this.piece = piece;
this.direction = direction;
this.g = g;
}

@Override
public int compareTo(Node other) {
return Integer.compare(this.g, other.g);
}
}
}

```

3.11 IDAStar.java

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.*;

public class IDAStar implements Solver {
    private final char[] priorityDirs = {'U', 'D', 'L', 'R'};
    private int heuristicType;
    private int nodesExpandedThisIteration;
    private int totalNodesExpanded;
    private int lastSummarizedStepCount = 0;

    private static class SummarizedStep {
        char piece;
        char direction;
        int moveCount;
        Board boardState;
        int g;
        int h;

        public SummarizedStep(char piece, char direction, int moveCount, Board boardState,
            int g, int h) {
            this.piece = piece;
            this.direction = direction;
            this.moveCount = moveCount;
            this.boardState = boardState;
            this.g = g;
            this.h = h;
        }

        public String getDisplay(String dirName) {
            return piece + "-" + dirName + (moveCount > 1 ? " " + moveCount + " kali" : "") +
                " (g=" + g + ", h=" + h + ", f=" + (g + h) + ")";
        }
    }

    public IDAStar(int heuristicType) {
        this.heuristicType = heuristicType;
    }
}
```

```

}

private String getAlgorithmName() {
return "IDA* Search";
}

private boolean needsHeuristic() {
return true;
}

private void ensureTestDirectoryExists() {
File directory = new File("test");
if (!directory.exists()) {
directory.mkdirs();
}
}

@Override
public void solve(Board start) {
totalNodesExpanded = 0;
lastSummarizedStepCount = 0;
int bound = Heuristic.calculate(start, heuristicType);
Path solutionPathNode = null;

System.out.println(getAlgorithmName() + " dengan heuristik " +
getHeuristicName(heuristicType));

while (true) {
System.out.println("Menjelajah dengan batas f-cost: " + bound);
nodesExpandedThisIteration = 0;
SearchResult result = search(new Path(start, null, '\0', '\0', 0,
Heuristic.calculate(start, heuristicType)), bound, new HashSet<>());
totalNodesExpanded += nodesExpandedThisIteration;

if (result.isGoal) {
solutionPathNode = result.path;
break;
}

if (result.nextBound == Integer.MAX_VALUE) {
System.out.println("Tidak ada batas berikutnya, solusi tidak ditemukan.");
break;
}
}
}

```

```

if (result.nextBound <= bound) {
    System.out.println("Peringatan: Batas f-cost tidak meningkat (lama: " + bound + ",
baru: " + result.nextBound + "). Menaikkan batas minimal.");
    bound++;
} else {
    bound = result.nextBound;
}

if (bound > 1000 && solutionPathNode==null) { // Increased limit for potentially
harder puzzles
    System.out.println("IDA* melebihi batas iterasi/f-cost maksimum (" + bound + "),
menghentikan.");
    break;
}
}

if (solutionPathNode != null) {
    List<SummarizedStep> summarizedPath = getSummarizedPath(solutionPathNode);
    this.lastSummarizedStepCount = summarizedPath.size();
    printSummarizedSolution(summarizedPath);
} else {
    System.out.println("Tidak ditemukan solusi!");
    System.out.println("Total Node yang dieksplorasi (hingga pencarian terakhir): " +
totalNodesExpanded);
    ensureTestDirectoryExists();
    try (PrintWriter writer = new PrintWriter(new FileWriter("test/output/output.txt",
false))) {
        writer.println(getAlgorithmName() + " dengan heuristik " +
Heuristic.getName(heuristicType));
        writer.println("Tidak ditemukan solusi!");
        writer.println("Total Node yang dieksplorasi: " + totalNodesExpanded);
    } catch (IOException e) {
        System.err.println("Gagal menulis ke file output.txt (solusi tidak ditemukan): " +
e.getMessage());
    }
}
}

@Override
public List<Board> solveAndReturnPath(Board start) {
    totalNodesExpanded = 0;
    lastSummarizedStepCount = 0;

```

```

int bound = Heuristic.calculate(start, heuristicType);
Path solutionPathNode = null;

System.out.println(getAlgorithmName() + " dengan heuristik " +
getHeuristicName(heuristicType) + " (mencari path list)");

while (true) {
System.out.println("Menjelajah dengan batas f-cost: " + bound);
nodesExpandedThisIteration = 0;
SearchResult result = search(new Path(start, null, '\0', '\0', 0,
Heuristic.calculate(start, heuristicType)), bound, new HashSet<>());
totalNodesExpanded += nodesExpandedThisIteration;

if (result.isGoal) {
solutionPathNode = result.path;
break;
}

if (result.nextBound == Integer.MAX_VALUE) {
System.out.println("Tidak ada batas berikutnya, solusi tidak ditemukan.");
break;
}

if (result.nextBound <= bound) {
System.out.println("Peringatan: Batas f-cost tidak meningkat (lama: " + bound + ",
baru: " + result.nextBound + "). Menaikkan batas minimal.");
bound++;
} else {
bound = result.nextBound;
}

if (bound > 1000 && solutionPathNode==null) { // Increased limit
System.out.println("IDA* melebihi batas iterasi/f-cost maksimum (" + bound + "),
menghentikan.");
break;
}
}

if (solutionPathNode == null) {
System.out.println("Tidak ditemukan solusi!");
System.out.println("Total Node yang dieksplorasi (hingga pencarian terakhir): " +
totalNodesExpanded);
ensureTestDirectoryExists();
try (PrintWriter writer = new PrintWriter(new FileWriter("test/output/output.txt",

```



```

false))) {
writer.println(getAlgorithmName() + " dengan heuristik " +
Heuristic.getName(this.heuristicType));
writer.println("Tidak ditemukan solusi!");
writer.println("Total Node yang dieksplorasi: " + totalNodesExpanded);
} catch (IOException e) {
System.err.println("Gagal menulis ke file output.txt (solusi tidak ditemukan): " +
e.getMessage());
}
return new ArrayList<>();
}

List<Board> boardPath = new ArrayList<>();
Path tempPath = solutionPathNode;
while (tempPath != null) {
boardPath.add(tempPath.board);
tempPath = tempPath.parent;
}
Collections.reverse(boardPath);

List<SummarizedStep> summarizedPath = getSummarizedPath(solutionPathNode);
this.lastSummarizedStepCount = summarizedPath.size();
printSummarizedSolution(summarizedPath);
return boardPath;
}

@Override
public int getLastSummarizedStepCount() {
return lastSummarizedStepCount;
}

@Override
public int getNodesExplored() {
return totalNodesExpanded;
}

private SearchResult search(Path currentPath, int bound, Set<String> pathStates) {
nodesExpandedThisIteration++;
Board currentBoard = currentPath.board;
int gCost = currentPath.g;
int hCost = currentPath.h;

```

```

int fCost = gCost + hCost;

if (fCost > bound) {
return new SearchResult(false, null, fCost);
}

if (isGoalState(currentBoard)) {
return new SearchResult(true, currentPath, fCost);
}

String boardKey = getBoardKey(currentBoard);
if (pathStates.contains(boardKey)) {
return new SearchResult(false, null, Integer.MAX_VALUE);
}
pathStates.add(boardKey);

int minNextBound = Integer.MAX_VALUE;

for (Piece piece : currentBoard.pieces) {
for (char dir : priorityDirs) {
if (canMove(currentBoard, piece.name, dir)) {
Board newBoard = move(currentBoard, piece.name, dir);
int newGCost = gCost + 1;
int newHCost = Heuristic.calculate(newBoard, heuristicType);
Path newPath = new Path(newBoard, currentPath, piece.name, dir, newGCost, newHCost);

SearchResult recursiveResult = search(newPath, bound, pathStates);

if (recursiveResult.isGoal) {
pathStates.remove(boardKey);
return recursiveResult;
}
minNextBound = Math.min(minNextBound, recursiveResult.nextBound);
}
}
}

pathStates.remove(boardKey);
return new SearchResult(false, null, minNextBound);
}

```

```

private List<SummarizedStep> getSummarizedPath(Path solutionPathNode) {
    List<SummarizedStep> summarizedSteps = new ArrayList<>();
    if (solutionPathNode == null) return summarizedSteps;

    List<Path> rawPathNodes = new ArrayList<>();
    Path current = solutionPathNode;
    while (current != null && current.parent != null) {
        rawPathNodes.add(current);
        current = current.parent;
    }
    Collections.reverse(rawPathNodes);

    if (rawPathNodes.isEmpty()) return summarizedSteps;

    char currentPiece = rawPathNodes.get(0).piece;
    char currentDirection = rawPathNodes.get(0).direction;
    int moveCount = 0;
    Board lastBoardInSequence = rawPathNodes.get(0).board;
    int gVal = rawPathNodes.get(0).g;
    int hVal = rawPathNodes.get(0).h;

    for (Path pathNode : rawPathNodes) {
        if (pathNode.piece == currentPiece && pathNode.direction == currentDirection) {
            moveCount++;
            lastBoardInSequence = pathNode.board;
            gVal = pathNode.g;
            hVal = pathNode.h;
        } else {
            summarizedSteps.add(new SummarizedStep(currentPiece, currentDirection, moveCount,
                lastBoardInSequence, gVal, hVal));
            currentPiece = pathNode.piece;
            currentDirection = pathNode.direction;
            moveCount = 1;
            lastBoardInSequence = pathNode.board;
            gVal = pathNode.g;
            hVal = pathNode.h;
        }
    }

    if (moveCount > 0) {
        summarizedSteps.add(new SummarizedStep(currentPiece, currentDirection, moveCount,
            lastBoardInSequence, gVal, hVal));
    }
}

```

```

}
return summarizedSteps;
}

private void printSummarizedSolution(List<SummarizedStep> summarizedPath) {
    ensureTestDirectoryExists();
    try (PrintWriter writer = new PrintWriter(new
        FileWriter("test/output/outputIDAStar.txt", false))) {
        String algoHeader = getAlgorithmName() + " dengan heuristik " +
            getHeuristicName(heuristicType);
        System.out.println("\n" + algoHeader);
        writer.println(algoHeader);
        writer.println("-----");

        int stepNumber = 1;
        for (SummarizedStep step : summarizedPath) {
            String stepDisplay = "Langkah " + stepNumber + ": " +
                step.getDisplay(getDirName(step.direction));
            System.out.println(stepDisplay);
            writer.println(stepDisplay);

            step.boardState.print();
            for (int r = 0; r < step.boardState.rows; r++) {
                for (int c = 0; c < step.boardState.cols; c++) {
                    writer.print(step.boardState.grid[r][c] + (c == step.boardState.cols - 1 ? "" : "
"));
                }
                writer.println();
            }
            System.out.println();
            writer.println();
            stepNumber++;
        }

        String summary = "Solusi ditemukan dalam " + summarizedPath.size() + " langkah
            (ringkas).";
        System.out.println(summary);
        writer.println(summary);

        String nodesExploredStr = "Total Node yang dieksplorasi: " + totalNodesExpanded;
        System.out.println(nodesExploredStr);
        writer.println(nodesExploredStr);
    }
}

```

```

} catch (IOException e) {
System.err.println("Gagal menulis langkah solusi ke file outputIDAStar.txt: " +
e.getMessage());
}
}

private boolean isGoalState(Board board) {
if (board.primaryPiece == null || (board.exitRow == -1 && board.exitCol == -1))
return false;
for (int[] cell : board.primaryPiece.cells) {
int r = cell[0], c = cell[1];
// Exit di atas grid
if (board.exitRow == -1 && board.exitCol == c && r == 0) return true;
// Exit di bawah grid
if (board.exitRow == board.rows && board.exitCol == c && r == board.rows - 1) return
true;
// Exit di kiri grid
if (board.exitCol == -1 && board.exitRow == r && c == 0) return true;
// Exit di kanan grid
if (board.exitCol == board.cols && board.exitRow == r && c == board.cols - 1) return
true;
// Exit di dalam grid (samping P)
if ((r == board.exitRow && Math.abs(c - board.exitCol) == 1) ||
(c == board.exitCol && Math.abs(r - board.exitRow) == 1)) {
return true;
}
}
return false;
}

private String getBoardKey(Board board) {
StringBuilder sb = new StringBuilder();
for (int i = 0; i < board.rows; i++) {
for (int j = 0; j < board.cols; j++) {
sb.append(board.grid[i][j]);
}
}
return sb.toString();
}

private String getDirName(char d) {

```

```

return switch (d) {
case 'L' -> "kiri";
case 'R' -> "kanan";
case 'U' -> "atas";
case 'D' -> "bawah";
default -> "?";
};
}

private String getHeuristicName(int type) { // Already exists
return Heuristic.getName(type);
}

private boolean canMove(Board board, char pieceName, char dir) {
Piece pieceToMove = null;
for (Piece p : board.pieces) {
if (p.name == pieceName) {
pieceToMove = p;
break;
}
}

if (pieceToMove == null || pieceToMove.cells.isEmpty()) {
return false;
}

PieceOrientation orientation = pieceToMove.getOrientation();

if (orientation == PieceOrientation.HORIZONTAL) {
if (dir == 'U' || dir == 'D') {
return false;
}
} else if (orientation == PieceOrientation.VERTICAL) {
if (dir == 'L' || dir == 'R') {
return false;
}
}

for (int[] cell : pieceToMove.cells) {
int currentRow = cell[0];
int currentCol = cell[1];
int newRow = currentRow;

```

```

int newCol = currentCol;

switch (dir) {
case 'L' -> newCol--;
case 'R' -> newCol++;
case 'U' -> newRow--;
case 'D' -> newRow++;
}

if (newRow < 0 || newRow >= board.rows || newCol < 0 || newCol >= board.cols) {
return false;
}

char destinationCellContent = board.grid[newRow][newCol];
boolean partOfItself = false;
for(int[] ownCell : pieceToMove.cells){
if(ownCell[0] == newRow && ownCell[1] == newCol){
partOfItself = true;
break;
}
}

if (destinationCellContent != '.' && destinationCellContent != 'K' && !partOfItself)
{
return false;
}

return true;
}

private Board move(Board board, char pieceName, char dir) {
Board newBoard = board.clone();
Piece targetPieceInNewBoard = null;
for(Piece p : newBoard.pieces){
if(p.name == pieceName){
targetPieceInNewBoard = p;
break;
}
}

if (targetPieceInNewBoard == null) return newBoard;

```

```

for (int[] cell : targetPieceInNewBoard.cells) {
newBoard.grid[cell[0]][cell[1]] = '.';
}

for (int[] cell : targetPieceInNewBoard.cells) {
switch (dir) {
case 'L' -> cell[1]--;
case 'R' -> cell[1]++;
case 'U' -> cell[0]--;
case 'D' -> cell[0]++;
}
}

for (int[] cell : targetPieceInNewBoard.cells) {
if (cell[0] >= 0 && cell[0] < newBoard.rows && cell[1] >=0 && cell[1] <
newBoard.cols) {
newBoard.grid[cell[0]][cell[1]] = pieceName;
} else {
System.err.println("Error critical (IDA*): Piece " + pieceName + " moved out of
bounds. Row: " + cell[0] + ", Col: " + cell[1]);
return board;
}
}

char KODE_BIDAK_UTAMA = 'P';
if (pieceName == KODE_BIDAK_UTAMA) {
newBoard.primaryPiece = targetPieceInNewBoard;
}

return newBoard;
}

private static class SearchResult {
boolean isGoal;
Path path;
int nextBound;

public SearchResult(boolean isGoal, Path path, int nextBound) {
this.isGoal = isGoal;
this.path = path;
this.nextBound = nextBound;
}
}

```



```

}

private static class Path {
    Board board;
    Path parent;
    char piece;
    char direction;
    int g;
    int h;

    public Path(Board board, Path parent, char piece, char direction, int g, int h) {
        this.board = board;
        this.parent = parent;
        this.piece = piece;
        this.direction = direction;
        this.g = g;
        this.h = h;
    }
}
}
}

```

3.12 GUI

3.12.1 MainGUI.java

```

public class MainGUI {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(() -> {
            new GUIFrame();
        });
    }
}

```

3.12.2 AnimationManager.java

```

import java.util.List;
import javax.swing.*;

```

```

public class AnimationManager {
    private final List<Board> steps;
    private final BoardGUI panel;
    private final JSlider speedSlider;
    private SwingWorker<Void, Board> worker;
    private boolean stopped = false;
    private Runnable onFinish;

    public AnimationManager(List<Board> steps, BoardGUI panel, JSlider speedSlider) {
        this(steps, panel, speedSlider, null);
    }

    public AnimationManager(List<Board> steps, BoardGUI panel, JSlider speedSlider,
        Runnable onFinish) {
        this.steps = steps;
        this.panel = panel;
        this.speedSlider = speedSlider;
        this.onFinish = onFinish;
    }

    public void start() {
        stopped = false;
        worker = new SwingWorker<>() {
            @Override
            protected Void doInBackground() throws Exception {
                for (int i = 0; i < steps.size(); i++) {
                    if (stopped) break;

                    // Kirim board ke publish
                    publish(steps.get(i));

                    // Dapatkan delay dari slider
                    int delay = speedSlider.getValue();
                    Thread.sleep(delay);
                }
                return null;
            }

            @Override
            protected void process(List<Board> chunks) {

```

```

Board latest = chunks.get(chunks.size() - 1);
panel.setBoard(latest);
}

@Override
protected void done() {
    if (onFinish != null) onFinish.run();
}
};
worker.execute();
}

public void stop() {
    stopped = true;
    if (worker != null && !worker.isDone()) {
        worker.cancel(true);
    }
}

public List<Board> getSteps() {
    return steps;
}
}

```

3.12.3 BoardGUI.java

```

import java.awt.*;
import javax.swing.*;

public class BoardGUI extends JPanel {
    private Board board;

    public void setBoard(Board board) {
        this.board = board;
        repaint();
    }

    @Override
    protected void paintComponent(Graphics g) {

```

```

super.paintComponent(g);
if (board == null) return;
int cellSize = 80;
int totalCols = board.cols;
int totalRows = board.rows;
int boardWidth = totalCols * cellSize;
int boardHeight = totalRows * cellSize;
int offsetX = (getWidth() - boardWidth) / 2;
int offsetY = (getHeight() - boardHeight) / 2;
for (int i = 0; i < totalRows; i++) {
    for (int j = 0; j < totalCols; j++) {
        char c = board.grid[i][j];
        Color fillColor = switch (c) {
            case '.' -> Color.WHITE;
            case 'P' -> Color.RED;
            default -> Color.LIGHT_GRAY;
        };
        int x = offsetX + j * cellSize;
        int y = offsetY + i * cellSize;
        g.setColor(fillColor);
        g.fillRect(x, y, cellSize, cellSize);
        g.setColor(Color.BLACK);
        g.drawRect(x, y, cellSize, cellSize);
        if (c != '.' && c != 'K') {
            g.setColor(Color.BLACK);
            g.drawString(String.valueOf(c), x + 35, y + 45);
        }
    }
}

if (board.exitCol <= 0 && board.exitRow >= 0 && board.exitRow < board.rows) {
    // K di kiri luar grid
    int x = offsetX - cellSize / 2;
    int y = offsetY + board.exitRow * cellSize;
    g.setColor(Color.GREEN);
    g.fillRect(x, y, cellSize / 2, cellSize);
    g.setColor(Color.BLACK);
    g.drawRect(x, y, cellSize / 2, cellSize);
} else if (board.exitCol >= board.cols && board.exitRow >= 0 && board.exitRow <
board.rows) {
    // K di kanan luar grid

```

```

int x = offsetX + board.cols * cellSize;
int y = offsetY + board.exitRow * cellSize;
g.setColor(Color.GREEN);
g.fillRect(x, y, cellSize / 2, cellSize);
g.setColor(Color.BLACK);
g.drawRect(x, y, cellSize / 2, cellSize);
} else if (board.exitRow <= 0 && board.exitCol >= 0 && board.exitCol < board.cols) {
// K di atas luar grid
int x = offsetX + board.exitCol * cellSize;
int y = offsetY - cellSize / 2;
g.setColor(Color.GREEN);
g.fillRect(x, y, cellSize, cellSize / 2);
g.setColor(Color.BLACK);
g.drawRect(x, y, cellSize, cellSize / 2);
} else if (board.exitRow >= board.rows && board.exitCol >= 0 && board.exitCol <
board.cols) {
// K di bawah luar grid
int x = offsetX + board.exitCol * cellSize;
int y = offsetY + board.rows * cellSize;
g.setColor(Color.GREEN);
g.fillRect(x, y, cellSize, cellSize / 2);
g.setColor(Color.BLACK);
g.drawRect(x, y, cellSize, cellSize / 2);
}
}
}
}

```

3.12.4 GUIFrame.java

```

import java.awt.*;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.util.List;
import javax.swing.*;

public class GUIFrame extends JFrame {
private JComboBox<String> algoCombo;
private JComboBox<String> heuristicCombo;

```

```

private JLabel statusLabel;
private BoardGUI boardPanel;
private File selectedFile;
private Board currentBoard;
private boolean isRunning = false;
private AnimationManager animationManager;
private JSlider speedSlider;

public GUIFrame() {
    super("Rush Hour Solver");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(900, 700);
    setLayout(new BorderLayout());

    initTopPanel();
    initBoardPanel();
    initBottomPanel();

    int initialIdx = algoCombo.getSelectedIndex();
    heuristicCombo.setEnabled(initialIdx != 1);

    setVisible(true);
}

private void initTopPanel() {
    JPanel topPanel = new JPanel(new FlowLayout());

    JButton fileButton = new JButton("Pilih File");
    algoCombo = new JComboBox<>(new String[]{"Greedy Best First Search", "Uniform Cost Search", "A*", "IDA*"});
    heuristicCombo = new JComboBox<>(new String[]{"Manhattan", "Euclidean", "Obstacle-aware"});
    JButton runButton = new JButton("Jalankan");

    heuristicCombo.setEnabled(false);

    algoCombo.addActionListener(e -> {
        int idx = algoCombo.getSelectedIndex();
        heuristicCombo.setEnabled(idx != 1);
    });
}

```

```

fileButton.addActionListener(e -> {
    JFileChooser fc = new JFileChooser("test/input/");
    if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
        selectedFile = fc.getSelectedFile();
        try {
            currentBoard = new Board(selectedFile);
            boardPanel.setBoard(currentBoard);
            statusLabel.setText("File dimuat: " + selectedFile.getName());
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(this, "Gagal membaca file:\n" + ex.getMessage());
        }
    }
});

runButton.addActionListener(e -> {
    if (isRunning) {
        if (animationManager != null) {
            animationManager.stop();
        }
        boardPanel.setBoard(currentBoard);
        runButton.setText("Jalankan");
        isRunning = false;
        statusLabel.setText("Dihentikan. Board dikembalikan.");
        return;
    }

    // Jika belum ada board
    if (currentBoard == null) {
        JOptionPane.showMessageDialog(this, "Pilih file dulu.");
        return;
    }

    // Jalankan solver
    runButton.setText("Hentikan");
    isRunning = true;
    int algoIdx = algoCombo.getSelectedIndex();
    int heurIdx = heuristicCombo.getSelectedIndex();
    Solver solver = switch (algoIdx) {
        case 0 -> new GBFS(heurIdx);
        case 1 -> new UCS();
        case 2 -> new AStar(heurIdx);
        case 3 -> new IDAStar(heurIdx);
        default -> null;
    };
});

```

```

};

if (solver == null) {
JOptionPane.showMessageDialog(this, "Solver belum tersedia.");
runButton.setText("Jalankan");
isRunning = false;
return;
}

Board boardCopy = currentBoard.clone();
long start = System.currentTimeMillis();
List<Board> path = solver.solveAndReturnPath(boardCopy);
long end = System.currentTimeMillis();
if (path == null || path.isEmpty()) {
JOptionPane.showMessageDialog(this, "Tidak ditemukan solusi.");
statusLabel.setText("Solusi tidak ditemukan.");
runButton.setText("Jalankan");
isRunning = false;
showSavePrompt();
return;
}

statusLabel.setText("Selesai dalam " + (end - start) + " ms | langkah: " +
solver.getLastSummarizedStepCount() + "| Node yang dieksplorasi: " +
solver.getNodesExplored());
animationManager = new AnimationManager(path, boardPanel, speedSlider, () -> {
runButton.setText("Jalankan");
isRunning = false;
showSavePrompt();
});
animationManager.start();
});

topPanel.add(fileButton);
topPanel.add(new JLabel("Algoritma:"));
topPanel.add(algoCombo);
topPanel.add(new JLabel("Heuristik:"));
topPanel.add(heuristicCombo);
topPanel.add(runButton);

add(topPanel, BorderLayout.NORTH);
}

private void initBoardPanel() {

```



```

boardPanel = new BoardGUI();
add(boardPanel, BorderLayout.CENTER);
}

private void initBottomPanel() {
JPanel bottom = new JPanel(new BorderLayout());
// --- Panel untuk kecepatan animasi ---
JPanel speedPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
speedSlider = new JSlider(100, 1000, 500);
speedSlider.setInverted(true); // kanan cepat
speedSlider.setMajorTickSpacing(300);
speedSlider.setMinorTickSpacing(100);
JLabel speedLabel = new JLabel("Kecepatan Animasi:");
speedPanel.add(speedLabel);
speedPanel.add(speedSlider);
// --- Status label ---
statusLabel = new JLabel("Pilih file untuk mulai.");
statusLabel.setHorizontalAlignment(SwingConstants.CENTER);
// Tambahkan ke bottom panel
bottom.add(speedPanel, BorderLayout.NORTH);
bottom.add(statusLabel, BorderLayout.CENTER);
add(bottom, BorderLayout.SOUTH);
}

private void showSavePrompt() {
int result = JOptionPane.showConfirmDialog(
this,
"Apakah Anda ingin menyimpan solusi ke file?",
"Simpan Solusi",
JOptionPane.YES_NO_OPTION
);
if (result == JOptionPane.YES_OPTION && selectedFile != null) {
try {
String name = selectedFile.getName();
String outputFileName = name.substring(0, name.lastIndexOf('.'));
String outputPath = "test/output/" + outputFileName + ".txt";
File outDir = new File("test/output");
if (!outDir.exists()) outDir.mkdirs();
PrintWriter writer = new PrintWriter(new FileWriter(outputPath));
if (animationManager == null){

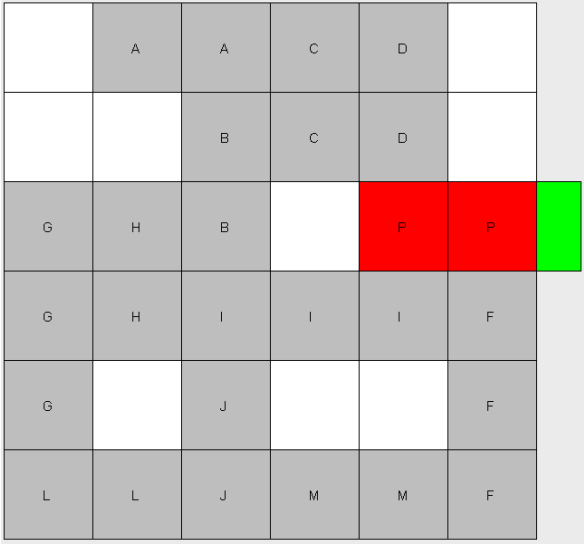
```

```
writer.println("Solusi disimpan dari file input: " + name);
writer.println("Tidak ditemukan solusi!");
writer.close();
} else {
writer.println("Solusi disimpan dari file input: " + name);
writer.println("Langkah solusi (tidak ringkas, hanya board):");
for (Board b : animationManager.getSteps()) {
for (int i = 0; i < b.rows; i++) {
for (int j = 0; j < b.cols; j++) {
writer.print(b.grid[i][j]);
}
writer.println();
}
writer.println();
}
writer.close();
}

JOptionPane.showMessageDialog(this, "Solusi disimpan ke " + outputPath);
} catch (Exception e) {
JOptionPane.showMessageDialog(this, "Gagal menyimpan solusi: " + e.getMessage());
}
}
}
}
```

BAB IV TESTING

4.1 Testing 1 (Test Jalan Keluar Kanan)

Input	Output
6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	<p>Output Gambar Final:</p>  <p>GBFS</p> <p>Manhattan: Selesai dalam 285 ms langkah: 77 Node yang dieksplorasi: 303</p> <p>Euclidean: Selesai dalam 209 ms langkah: 77 Node yang dieksplorasi: 303</p> <p>Obstacle-aware: Selesai dalam 140 ms langkah: 71 Node yang dieksplorasi: 289</p> <p>UCS</p> <p>Selesai dalam 59 ms langkah: 7 Node yang dieksplorasi: 574</p> <p>A*</p>

	Manhattan: Selesai dalam 82 ms langkah: 7 Node yang dieksplorasi: 308
	Euclidean: Selesai dalam 46 ms langkah: 7 Node yang dieksplorasi: 308
	Obstacle-aware: Selesai dalam 45 ms langkah: 6 Node yang dieksplorasi: 115
	IDA*
	Manhattan: Selesai dalam 428 ms langkah: 7 Node yang dieksplorasi: 233737
	Euclidean: Selesai dalam 312 ms langkah: 7 Node yang dieksplorasi: 233737
	Obstacle-aware: Selesai dalam 37 ms langkah: 7 Node yang dieksplorasi: 9594

4.2 Testing 2 (Test Jalan Keluar Bawah)

Input	Output
5 6 6 AA..F. BBP.F. ..P.F. CCDD.. .EE... K	Output Gambar Final: <div> </div>

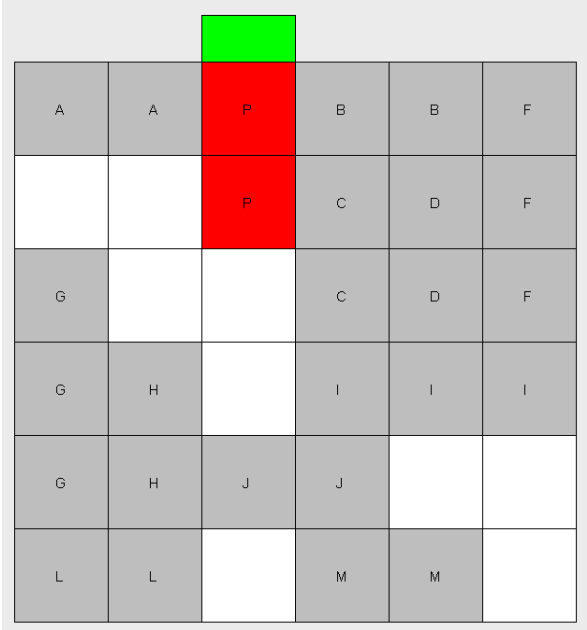
	GBFS
	Manhattan: Selesai dalam 16 ms langkah: 10 Node yang dieksplorasi: 27
	Euclidean: Selesai dalam 0 ms langkah: 10 Node yang dieksplorasi: 27
	Obstacle-aware: Selesai dalam 0 ms langkah: 3 Node yang dieksplorasi: 5
	UCS
	Selesai dalam 15 ms langkah: 4 Node yang dieksplorasi: 98
	A*
	Manhattan: Selesai dalam 16 ms langkah: 4 Node yang dieksplorasi: 13
	Euclidean: Selesai dalam 19 ms langkah: 4 Node yang dieksplorasi: 13
	Obstacle-aware: Selesai dalam 0 ms langkah: 3 Node yang dieksplorasi: 5
	IDA*
	Manhattan: Selesai dalam 15 ms langkah: 4 Node yang dieksplorasi: 131
	Euclidean: Selesai dalam 16 ms langkah: 4 Node yang dieksplorasi: 131
	Obstacle-aware: Selesai dalam 0 ms langkah: 5 Node yang dieksplorasi: 16

4.3 Testing 3 (Test Jalan Keluar Kiri)

Input	Output
6 6 11 AAB..F ..BCDF KGPPCDF GH.III GHJ... LLJMM.	<p>Output Gambar Final:</p>
	<p>GBFS</p> <p>Manhattan: Selesai dalam 63 ms langkah: 14 Node yang dieksplorasi: 31</p> <p>Euclidean: Selesai dalam 51 ms langkah: 14 Node yang dieksplorasi: 31</p> <p>Obstacle-aware: Selesai dalam 34 ms langkah: 9 Node yang dieksplorasi: 23</p>
	<p>UCS</p> <p>Selesai dalam 29 ms langkah: 4 Node yang dieksplorasi: 100</p>
	<p>A*</p> <p>Manhattan: Selesai dalam 0 ms langkah: 4 Node yang dieksplorasi: 43</p> <p>Euclidean:</p>

	<p>Selesai dalam 9 ms langkah: 4 Node yang dieksplorasi: 43</p> <p>Obstacle-aware:</p> <p>Selesai dalam 16 ms langkah: 4 Node yang dieksplorasi: 11</p>
	IDA*
	<p>Manhattan:</p> <p>Selesai dalam 0 ms langkah: 4 Node yang dieksplorasi: 1305</p>
	<p>Euclidean:</p> <p>Selesai dalam 11 ms langkah: 4 Node yang dieksplorasi: 1305</p>
	<p>Obstacle-aware:</p> <p>Selesai dalam 0 ms langkah: 4 Node yang dieksplorasi: 246</p>

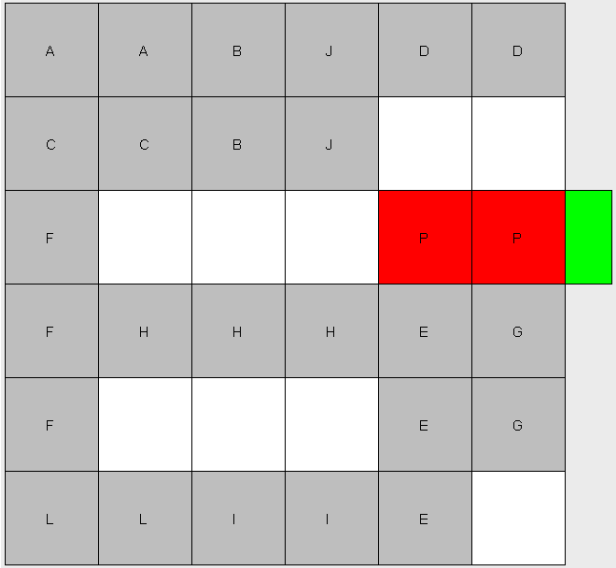
4.4 Testing 4 (Test Jalan Keluar Atas)

Input	Output
6 6 11 K AABB.F ...CDF G.PCDF GHPIII GHJJ.. LL.MM.	<div> Output Gambar Final:  </div>
	GBFS

	Manhattan: Selesai dalam 32 ms langkah: 3 Node yang dieksplorasi: 4
	Euclidean: Selesai dalam 11 ms langkah: 3 Node yang dieksplorasi: 4
	Obstacle-aware: Selesai dalam 8 ms langkah: 2 Node yang dieksplorasi: 4
	UCS Selesai dalam 16 ms langkah: 3 Node yang dieksplorasi: 112
	A* Manhattan: Selesai dalam 16 ms langkah: 3 Node yang dieksplorasi: 4
	Euclidean: Selesai dalam 0 ms langkah: 3 Node yang dieksplorasi: 4
	Obstacle-aware: Selesai dalam 0 ms langkah: 2 Node yang dieksplorasi: 4
	IDA* Manhattan: Selesai dalam 15 ms langkah: 3 Node yang dieksplorasi: 25
	Euclidean: Selesai dalam 16 ms langkah: 3 Node yang dieksplorasi: 25
	Obstacle-aware: Selesai dalam 15 ms langkah: 3 Node yang dieksplorasi: 5

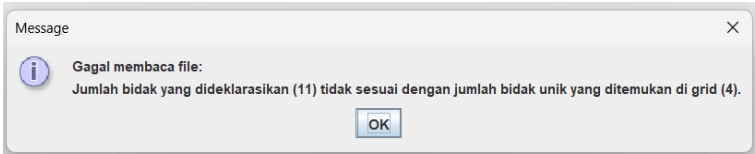
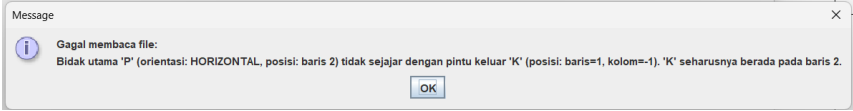
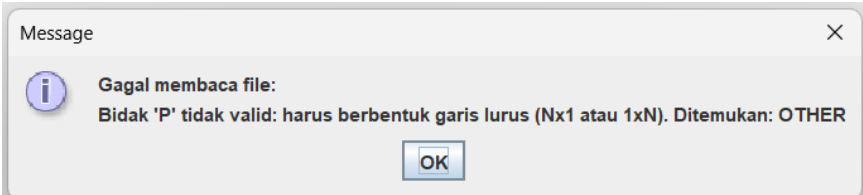
4.5 Testing 5 (Test Rush Hour® Deluxe Edition - Grandmaster Level 60)

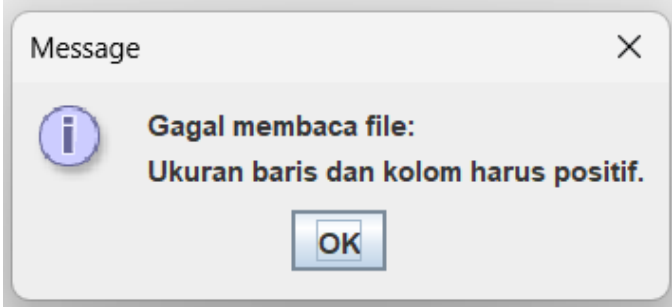
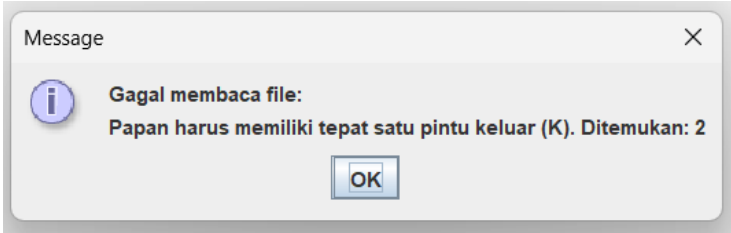
Source: <https://youtu.be/xKDE1E3Q4XI?si=Viz3gnwvIwQeobEN>

Input	Output
6 6 11 AAB.DD CCB.E. F.PPE.K FHHHEG F..J.G LL.JII	<p>Output Gambar Final:</p>  <p>GBFS</p> <p>Manhattan: Selesai dalam 1065 ms langkah: 533 Node yang dieksplorasi: 8248</p> <p>Euclidean: Selesai dalam 919 ms langkah: 533 Node yang dieksplorasi: 8248</p> <p>Obstacle-aware: Selesai dalam 161 ms langkah: 76 Node yang dieksplorasi: 8157</p> <p>UCS</p> <p>Selesai dalam 159 ms langkah: 53 Node yang dieksplorasi: 2958</p> <p>A*</p> <p>Manhattan: Selesai dalam 163 ms langkah: 51 Node yang dieksplorasi: 2950</p>

	<p>Euclidean:</p> <p>Selesai dalam 192 ms langkah: 51 Node yang dieksplorasi: 2950</p> <p>Obstacle-aware:</p> <p>Selesai dalam 140 ms langkah: 54 Node yang dieksplorasi: 3867</p>
--	--

4.6 Testing 6 (Testing Validasi)

Input	Output
6 6 10HH ...PPHKH AAAAAA BBBBBB	
6 6 11 AAB..F K..BCDF GPPCDF GH.III GHJ... LLJMM.	
6 6 11 PPB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	

-5 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	
6 6 11 AAB..F ..BCDF GPPCDFK GH.IIIK GHJ... LLJMM.	

BAB V

ANALISIS HASIL

Setelah seluruh algoritma pathfinding diimplementasikan, dilakukan serangkaian eksperimen untuk mengamati perilaku dan performa masing-masing metode dalam menyelesaikan puzzle Rush Hour. Analisis dilakukan terhadap empat algoritma: Greedy Best First Search (GBFS), Uniform Cost Search (UCS), A*, dan Iterative Deepening A* (IDA*), dengan mempertimbangkan efisiensi waktu pencarian solusi, jumlah simpul yang dieksplorasi, dan efektivitas dalam menemukan solusi terpendek.

Secara umum, algoritma GBFS menunjukkan kecepatan eksekusi yang tinggi karena hanya mempertimbangkan nilai heuristik ($h(n)$) tanpa memperhatikan biaya sebenarnya ($g(n)$). Hal ini membuatnya sangat cepat pada puzzle sederhana, tetapi seringkali terjebak dalam jalur

sub-optimal karena tidak mengevaluasi total biaya lintasan. Kompleksitas waktu GBFS dapat dikatakan $O(b^d)$ dalam kasus terburuk, di mana b adalah branching factor dan d adalah kedalaman solusi, namun jumlah node yang dikunjungi bisa lebih kecil dibanding algoritma lain karena penelusurannya sangat mengarah (*greedy*).

UCS menjamin solusi optimal karena selalu memilih node dengan biaya total terendah sejauh ini ($g(n)$). Namun, karena UCS tidak menggunakan heuristik, algoritma ini bisa mengeksplorasi banyak simpul tak relevan, terutama pada puzzle dengan jalur solusi yang panjang. Kompleksitas waktunya juga $O(b^d)$, namun dalam praktik UCS sering lebih lambat dibanding A^* dan IDA^* , karena tidak memiliki panduan arah menuju goal selain memperluas semua kemungkinan biaya rendah.

A^* adalah algoritma yang paling seimbang, menggabungkan $g(n)$ dan $h(n)$ untuk memilih node yang paling menjanjikan dengan pendekatan *best-first*. Dengan heuristik yang akurat seperti Obstacle-Aware, A^* dapat menemukan solusi optimal dengan eksplorasi yang efisien. Kompleksitas waktu A^* adalah $O(b^d)$ dalam kasus terburuk, tetapi sangat bergantung pada kualitas heuristik. Dengan heuristik admissible dan consistent, A^* tetap menjamin solusi optimal. Dalam pengujian kami, A^* seringkali menghasilkan solusi lebih cepat dari UCS dan lebih akurat dari GBFS.

IDA^* adalah varian A^* yang menggunakan DFS dengan batasan nilai $f(n)$. Algoritma ini sangat efisien dalam hal penggunaan memori ($O(d)$), karena hanya menyimpan satu jalur saat berjalan. Namun, ia dapat mengulang banyak node karena sifatnya yang *iterative deepening*. Kompleksitas waktunya adalah $O(b^d)$ juga, tetapi konstanta tersembunyi sering lebih besar karena pengulangan threshold yang terus meningkat. Meskipun demikian, IDA^* sangat cocok digunakan untuk puzzle yang kompleks dan dalam ketika memori terbatas.

Dari eksperimen visual menggunakan GUI, juga diamati bahwa kecepatan animasi dapat mempengaruhi persepsi terhadap efisiensi solver. Oleh karena itu, ditambahkan slider kecepatan agar pengguna dapat menyesuaikan sesuai preferensi. Selain itu, fitur penyimpanan solusi memungkinkan analisis lebih lanjut terhadap langkah-langkah penyelesaian.

Secara keseluruhan, pemilihan algoritma terbaik sangat bergantung pada konteks: A^* memberikan performa terbaik untuk keseimbangan kecepatan dan optimalitas, IDA^* cocok untuk efisiensi memori, UCS untuk solusi pasti (meski lambat), dan GBFS untuk solusi cepat tanpa jaminan optimal.

BAB VI

IMPLEMENTASI BONUS

5.1 GUI

GUI pengguna dirancang menggunakan library Java Swing, yang memungkinkan pengguna menjalankan solver melalui tampilan grafis yang intuitif. Implementasi GUI dilakukan dalam kelas GUIFrame dan BoardGUI. Di dalam GUIFrame, kami menyusun layout utama menggunakan BorderLayout dengan tiga bagian utama: panel atas untuk kontrol input, panel tengah untuk visualisasi papan, dan panel bawah untuk informasi status serta pengaturan kecepatan animasi. Panel atas menyediakan komponen JComboBox untuk memilih algoritma dan heuristik, JButton untuk memuat file input dan menjalankan solver, serta listener untuk mengatur heuristik berdasarkan algoritma terpilih. Setelah file dipilih, objek Board akan diinisialisasi dan divisualisasikan melalui BoardGUI, yang mewarisi JPanel dan melakukan override paintComponent untuk menggambar kotak-kotak papan, bidak-bidak, serta pintu keluar (exit). Untuk animasi solusi, kami membuat kelas AnimationManager yang menggunakan javax.swing.Timer untuk memperbarui tampilan papan berdasarkan path yang dihasilkan solver. Slider (JSlider) ditambahkan di bagian bawah GUI untuk memungkinkan pengguna mengatur kecepatan animasi secara dinamis. Slider ini di-*bind* dengan AnimationManager agar delay antar langkah dapat berubah saat animasi berjalan. Setelah solusi selesai dianimasikan, sebuah dialog konfirmasi (JOptionPane) akan muncul untuk menawarkan penyimpanan solusi ke dalam file teks, menggunakan nama file input yang sama.

5.2 Obstacle Aware Heuristic

Heuristik ini dikembangkan untuk memberikan estimasi yang lebih realistis terhadap jarak antara bidak utama ('P') dan titik keluar ('K'), dengan mempertimbangkan keberadaan rintangan di jalur tersebut. Implementasinya dilakukan dalam kelas Heuristic, sebagai tambahan dari heuristik *Manhattan* dan *Euclidean*. Dalam ObstacleAwareHeuristic, kami melakukan traversal dari posisi bidak utama ke arah exit dan mencatat berapa banyak bidak yang menghalangi jalur langsung tersebut. Untuk setiap rintangan, ditambahkan penalti ke nilai heuristik akhir. Jika rintangan tidak hanya satu tetapi berlapis atau saling bertumpukan, maka penalti akan bertambah, merepresentasikan kesulitan tambahan untuk menggeser semua bidak

tersebut. Dengan demikian, algoritma seperti *A* dan *GBFS* yang menggunakan heuristik ini akan memiliki panduan yang lebih tepat untuk memilih jalur solusi yang paling menjanjikan.

5.3 Iterative Deepening A^* (IDA*)

IDA* merupakan variasi dari A^* yang menggunakan strategi pencarian berbasis kedalaman (depth-first) dengan batasan $f = g + h$ yang diiterasi secara bertahap. Implementasi algoritma ini dilakukan dalam kelas IDAStar, yang menginisialisasi pencarian dengan threshold awal berdasarkan nilai heuristik node awal. Fungsi pencarian DFS dimodifikasi agar hanya mengeksplorasi node yang f -nya tidak melebihi threshold. Jika seluruh jalur gagal ditemukan dalam iterasi tersebut, threshold dinaikkan ke nilai f minimum yang melebihi batas sebelumnya. Proses ini diulang hingga solusi ditemukan. Salah satu tantangan implementasi IDA* adalah menghindari eksplorasi redundan tanpa menggunakan struktur data besar seperti priority queue. Oleh karena itu, kami juga menjaga jejak state menggunakan struktur Set untuk mencegah pengulangan. Kelebihan utama dari IDA* dibanding A^* adalah penggunaan memori yang jauh lebih hemat, karena tidak menyimpan semua node terbuka secara simultan. Hal ini membuat IDA* lebih cocok untuk masalah dengan state space besar.

LAMPIRAN

Repository Program

Repository program dapat diakses melalui tautan GitHub berikut :

https://github.com/sonix03/Tucil3_13523049_13523121

Implementasi Spesifikasi Program

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/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://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/>