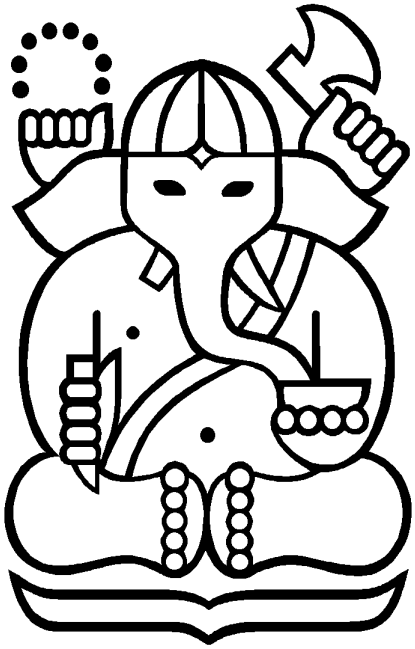
LAPORAN TUGAS KECIL 3

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Strategi Algoritma (IF-2211 2024/2025)



Disusun oleh :

Ranashahira Reztaputri – 13523007 – K01

Noumisyifa Nabila Nareswari – 13523058 – K01

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

BAB I

DASAR TEORI

1.1. Penjelasan Algoritma

1.1.1. Uniform Cost Search (UCS)

Uniform Cost Search atau UCS merupakan salah satu algoritma pathfinding yang merupakan *uninformed search* atau *blind search*, yaitu pencarian tanpa adanya informasi tambahan seperti heuristik. Uniform Cost Search (UCS) adalah algoritma penelusuran graf yang menemukan jalur dengan biaya kumulatif paling sedikit dari simpul awal ke simpul tujuan. Pada UCS, pencarian jalur memanfaatkan skala prioritas, di mana jalur yang dicari lebih dulu adalah jalur dengan *cost* terkecil. Fungsi evaluasi pada UCS adalah f(n) = g(n), dengan g(n) adalah *cost* jalur dari root ke node n. UCS seringkali dikatakan sebagai spesialisasi dari algoritma A* dengan fungsi heuristik h(n) bernilai nol.

```
public class UCS {
    private Utils utils;
    private int exploredNodes;
    private Scanner scanner;
    private Node solution;
    private Set<Board> visitedBoards;

    public UCS(Board initialBoard) {
        utils = new Utils();
        exploredNodes = 0;
        scanner = new Scanner(System.in);
        visitedBoards = new HashSet<Board>();
        solution = null;
        solve(initialBoard);
    }
```

Pada algoritma UCS untuk penyelesaian puzzle Rush Hour ini, terdapat atribut exploredNodes yang menghitung jumlah node yang telah ditelusuri, solution yang berisi node yang merupakan solusi, dan visitedBoards yang menyimpan data Board yang pernah dikunjungi sebelumnya untuk mencegah terbentuknya siklus. Berikut merupakan potongan kode dari fungsi solve pada kelas UCS:

```java
private void solve(Board initialBoard) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    pq.add(new Node(initialBoard, cost:0, parent:null));

    while(!pq.isEmpty()){
        Node current = pq.poll();
        Board currentBoard = current.getState();

        if(visitedBoards.contains(currentBoard)){
            continue;
        }

        exploredNodes++;
        visitedBoards.add(currentBoard);

        if(currentBoard.isFinished()){
            solution = current;
            break;
        }

        List<Board> nextBoards = utils.generateAllPossibleMoves(currentBoard, currentBoard.getLastMoves(),
        currentBoard.getLastDist(), currentBoard.getLastPiece());
```

Algoritma ini memanfaatkan kelas Queue, atau lebih spesifiknya PriorityQueue. Setiap node yang ditelusuri dimasukkan ke dalam Queue. PriorityQueue digunakan untuk memastikan node yang akan dikunjungi terurut dari *cost* terkecil terlebih dahulu. Dengan begitu, node yang ditelusuri adalah node dengan *cost* terkecil sehingga dapat menemukan solusi dengan *cost* minimum. Tiap node yang berada di dalam PriorityQueue ditelusuri sampai menemukan node tujuan (primary piece berhasil keluar) atau PriorityQueue kosong.

1.1.2. Greedy Best First Search (GBFS)

Greedy Best First Search atau GBFS merupakan algoritma penelusuran graf yang berusaha menemukan jalur paling menjanjikan dari titik awal hingga menuju titik tujuan. Algoritma ini bekerja dengan mengevaluasi biaya dari setiap jalur yang mungkin dan kemudian memperluas jalur dengan biaya terendah. Algoritma ini termasuk ke dalam *informed search* karena memanfaatkan heuristik sebagai informasi tambahan.GBFS memiliki fungsi heuristik f(n) = h(n), di mana h(n) merupakan fungsi heuristik yang menghitung estimasi *cost* dari node n ke node tujuan. Berikut adalah langkah pergerakan algoritma yang diimplementasikan di program

```java
public class GBFS {
    private List<Board> solutionPath;
    private Board currentBoard;
    private int nodeCount; // untuk menghitung jumlah node yang diperiksa
    private Set<Board> visitedBoards; // untuk menghindari siklus - langsung menyimpan objek Board
    private Scanner scanner;

    public GBFS(){
        solutionPath = new ArrayList<Board>();
        nodeCount = 0;
        visitedBoards = new HashSet<Board>();
        scanner = new Scanner(System.in);
    }
```

Ini adalah atribut dan constructor dari kelas GBFS, penjelasan atribut tertera pada komentar kode

```java
public void solve(Board initBoard, boolean isTreeHeuristic){
    Utils utils = new Utils();
    solutionPath.add(initBoard);
    currentBoard = new Board(initBoard);
    visitedBoards.add(currentBoard);

    TreeHeuristic th = new TreeHeuristic();
    DistHeuristic dh = new DistHeuristic();

    int heuristicValue;
```

Berikut adalah inisialisasi nilai-nilai sebelum menjalankan algoritma penyelesaian. Objek Utils dibutuhkan untuk *generate* semua kemungkinan papan. Bentuk awal papan ditambahkan ke dalam *path* solusi yang merupakan sebuah List of Boards, currentBoard untuk *tracking* papan dengan posisi apa yang saat ini sedang dieksplorasi, visitedBoards untuk mencegah siklik, th dan dh untuk masing-masing fungsi heuristik yang ingin digunakan,dan heuristicValue untuk menyimpan nilai hasil fungsi heuristik saat ini.

```
while (!currentBoard.isFinished()) {
    List<Board> currentPossibleBoards = new ArrayList<Board>(utils.generateAllPossibleMoves(currentBoard, currentBoard.getLastMoves(), currentBoard.getLastDist(), currentBoard.getLastPiece()));

    Board bestBoard = null;
    int minHeuristicValue = Integer.MAX_VALUE;

    for (Board nextBoard : currentPossibleBoards) {
        nodeCount++;
        
        if (nextBoard.isFinished()) {
            bestBoard = nextBoard;
            break;
        }

        if (isContain(nextBoard)) {
            continue;
        }

        if(isTreeHeuristic) heuristicValue = th.evaluate(nextBoard);
        else heuristicValue = dh.evaluate(nextBoard);

        if (heuristicValue < minHeuristicValue) {
            minHeuristicValue = heuristicValue;
            bestBoard = nextBoard;
        }
    }

    if (bestBoard == null) {
        break;
    }

    currentBoard = bestBoard;
    visitedBoards.add(currentBoard);
    solutionPath.add(currentBoard);

    if (currentBoard.isFinished()) {
        scanner.nextLine();
        break;
    }
}
System.out.println("node: " + this.getNodeCount());
scanner.nextLine();
```

Berikut adalah *loop* utama dari fungsi penyelesaian dan berikut adalah langkah-langkah yang terjadi di tahap ini:

- Generate semua posisi papan yang mungkin dari kombinasi posisi papan saat ini dan simpan ke dalam currentPossibleBoards
- State nilai awal untuk pengecekan (bestBoard = null dan minHeuristicValue = maxInteger)
- Iterasi untuk semua kemungkinan Board yang di-*generate* dan evaluasi nilai hasil heuristiknya lalu menyimpan setiap kali ada susunan papan yang nilai heuristiknya lebih kecil dari nilai heuristik saat ini. Nilai node yang dieksplor jugua ditambahkan setiap iterasi.
- Jika bestBoard, yaitu kombinasi apapun yang terkecil dari list of Board yang di-*generate* bernilai null, maka tidak ada kombinasi papanyang bisa dicek kembali dan keluar dari loop.
- Jika Board saat ini adalah solusi, maka keluar dari loop.
- Pengecekan apakah board sudah *solved* atau belum dilakukan di *display* hasil yang mengecek apakah currentBoard.isFinished() bernilai true atau tidak.

1.1.3. A* Search

A* Search merupakan algoritma pathfinding yang menghindari perluasan jalur yang *cost*-nya sudah bernilai tinggi. A* Search termasuk *informed search* karena memanfaatkan heuristik dalam pencarian jalur. Algoritma ini memiliki cara kerja yang serupa dengan UCS, yaitu memperluas jalur dengan *cost* terkecil, tetapi perbedaanya adalah A* Search menggunakan heuristik. Fungsi evaluasi dari A* Search adalah f(n) = g(n) + h(n), di mana g(n) adalah *cost* aktual dari root ke node n dan h(n) adalah estimasi *cost* dari node n ke node tujuan.

```java
package solver;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
import obj.Board;
import utils.Utils;

public class AStar {
    private Utils utils;
    private int exploredNodes;
    private Scanner scanner;
    private Node solution;
    private Set<Board> visitedBoards;
    private TreeHeuristic th;
    private DistHeuristic dh;
    private int heuristicValue;

    public AStar(Board initialBoard, boolean isTreeHeuristic) {
        utils = new Utils();
        exploredNodes = 0;
        scanner = new Scanner(System.in);
        visitedBoards = new HashSet<Board>();
        solution = null;
        th = new TreeHeuristic();
        dh = new DistHeuristic();
        solve(initialBoard, isTreeHeuristic);

    }
```

Kelas AStar memiliki atribut exploredNodes untuk menghitung jumlah node yang ditelusuri, visitedBoards untuk menampung Board yang sudah pernah dikunjungi, solution untuk menyimpan Board yang merupakan solusi, serta th, dh, dan heuristicValue yang menyimpan nilai heuristik berdasarkan jenis heuristiknya.

```
private void solve(Board initialBoard, boolean isTreeHeuristic) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    if(isTreeHeuristic) heuristicValue = th.evaluate(initialBoard);
    else heuristicValue = dh.evaluate(initialBoard);
    pq.add(new Node(initialBoard, heuristicValue, parent:null));

    while(!pq.isEmpty()){
        Node current = pq.poll();
        Board currentBoard = current.getState();

        if(visitedBoards.contains(currentBoard)){
            continue;
        }
        exploredNodes++;
        visitedBoards.add(currentBoard);
        if(currentBoard.isFinished()){
            solution = current;
            break;
        }

        List<Board> nextBoards = utils.generateAllPossibleMoves(currentBoard, currentBoard.getLastMoves(),
        currentBoard.getLastDist(), currentBoard.getLastPiece());
        for(Board nextBoard : nextBoards){
            if(!visitedBoards.contains(nextBoard)){
                if(isTreeHeuristic) heuristicValue = th.evaluate(nextBoard);
                else heuristicValue = dh.evaluate(nextBoard);
                Node nextNode = new Node(nextBoard, current.getCost()+1+heuristicValue, current);
                pq.add(nextNode);
            }
        }
    }
}
```

Cara kerja dari algoritma A* tidak jauh berbeda dari UCS. Perbedaannya adalah fungsi evaluasi A* juga menggunakan fungsi heuristik h(n) sebagai aspek pertimbangan *cost*. Jadi, secara keseluruhan, algoritma A* adalah memasukkan node awal ke PriorityQueue, lalu memasukkan semua tetangganya ke dalam PriorityQueue. Tetangganya yang memiliki *cost* terkecil akan ditelusuri lebih dulu dan dihapus dari PriorityQueue. Proses ini terus berjalan sampai menemukan node tujuan (*primary piece* berhasil keluar) atau PriorityQueue kosong.

1.1.4. Iterative Deepening Search (IDS)

Iterative Deepening Search atau IDS merupakan algoritma penelusuran graf yang menggabungkan *completeness* dari Breadth-First Search dengan efisiensi memori dari Depth-First Search. Pencarian dengan metode IDS bekerja dengan cara berulang kali menjalankan versi DFS yang dibatasi kedalamannya, dengan batas kedalaman yang terus meningkat hingga tujuan ditemukan. Sama halnya dengan UCS, algoritma pencarian ini termasuk *uninformed search* atau *blind search* karena tidak melibatkan informasi tambahan.

Dalam implementasinya, semua informasi yang diperlukan di dalam satu disimpan di dalam satu objek bertipe IDSNode dengan atribut dan konstruktor sebagai berikut

```java
public class IDSNode {
    private Board currentBoard;
    private List<Board> pathOfBoards;
    private int depth;

    IDSNode(Board inpBoard){
        this.currentBoard = inpBoard;
        this.pathOfBoards = new ArrayList<Board>();
        pathOfBoards.add(inpBoard);
        this.depth = 0;
    }
```

currentBoard berfungsi untuk menyimpan Board yang terakhir dieksplor oleh IDSNode saat ini, pathOfBoards berfungsi menyimpan jalur (*states*) yang telah dilalui jalur ini, dan depth adalah kedalaman saat ini. Ada satu konstruktor cukup penting sebagai berikut.

```java
IDSNode(IDSNode prevNode){
    this.currentBoard = prevNode.currentBoard;
    this.pathOfBoards = new ArrayList<Board>(prevNode.pathOfBoards);
    this.depth = prevNode.depth;
}
```

Konstruktor ini berfungsi untuk menghasilkan IDSNode untuk langkah selanjutnya yang berisikan path sebelumnya ditambah dengan pathOfBoards dari IDSNode sebelumnya. Berikut adalah atribut yang dideklarasikan di kelas IDS.java.

```java
public class IDS {
    private Stack<IDSNode> treeStack;
    private int curMaxDepth;
    private IDSNode solution;
    private int exploredNodes;
    private boolean isFinished;
```

Atribut treeStack adalah atribut yang berfungsi untuk menyimpan Node yang perlu diproses dengan logika FIFO (First In First Out), curMaxDepth menyimpan kedalaman sata ini, solution menyimpan Node solusi jika ditemukan, exploredNodes menyimpan jumlah Node yang telah dieksplorasi, dan isFinished untuk menyimpan status apakah sudah ditemukan solusi atau belum. Berikut adalah awal dari logika penyelesaian dengan IDS yang berisikan nilai awal.

```
public IDS(Board initBoard){
    this.solution = null;          // Menyimpan node solusi jika ditemukan
    this.exploredNodes = 0;        // Menghitung node yang dieksplorasi/digenerate
    this.curMaxDepth = 0;          // Batas kedalaman dimulai dari 0 untuk Iterative Deepening
    this.isFinished = false;

    Utils utils = new Utils();

    IDSNode rootNode = new IDSNode(initBoard); // Membuat node awal (kedalaman 0
```

Berikut adalah awal dari iterasi penyelesaian

```
while (this.solution == null) {
    this.treeStack = new Stack<IDSNode>();
    this.treeStack.push(rootNode);

    while (!this.treeStack.isEmpty()) {
        IDSNode currentNode = this.treeStack.pop();
        Board currentBoardState = currentNode.getCurrentBoard();

        if (currentBoardState.isFinished()) {
            this.solution = currentNode;
            this.isFinished = true;
            break;
        }

        if (currentNode.getDepth() < this.curMaxDepth) {

            List<Board> nextPossibleBoards;
            nextPossibleBoards = utils.generateAllPossibleMoves(currentBoardState,
            currentBoardState.getLastMoves(),
            currentBoardState.getLastDist(), currentBoardState.getLastPiece());


            for (int i = nextPossibleBoards.size() - 1; i >= 0; i--) {
                Board nextBoard = nextPossibleBoards.get(i);
                IDSNode childNode = new IDSNode(nextBoard, currentNode);
                this.treeStack.push(childNode);
                this.exploredNodes++;
            }
        }
    }

    if (this.solution != null) {
        break;
    }

    this.curMaxDepth++;
```

Berikut adalah *loop* utama dari fungsi penyelesaian dan berikut adalah langkah-langkah yang terjadi di tahap ini:

- Untuk setiap iterasi dengan maxDepth yang baru, diinisialisasi nilai awal treeStack, diisi dengan rootNode.
- Memulai iterasi dengan menge-pop Node terluar di stack.

- Cek apakah Node tersebut berisi currentboard yang sudah selesai, jika iya maka keluar dari loop.
- Jika kedalaman Node saat ini masih belum mencapai maksimum, maka *generate* semua kemungkinan kombinasi gerak Board selanjutnya, lalu buat childNode untuk masing-masing Board yang di-*generate* dan push ke dalam stack.
- Keluar dari loop kedua, cek apakah sudah ditemukan solusi dengan maksimal kedalaman saat ini, jika sudah maka keluar dari loop utama.
- Jika belum ditemukan solusi, maka lanjut iterasi dengan maxDepth baru, dan terus hingga ditemukan solusi.

BAB II

ALGORITMA

2.1. Analisis Algoritma

2.1.1. Analisis Fungsi Heuristik Pencarian

Untuk memastikan bahwa suatu *tree-search* optimal, maka heuristik yang digunakan harus *admissible*. Heuristik dikatakan *admissible* jika untuk semua node berlaku: h(n) <= h*(n), di mana h*(n) merupakan *cost* aktual untuk mencapai node tujuan. Hal ini menunjukkan bahwa heuristik yang *admissible* adalah heuristik yang tidak pernah melebih-lebihkan *cost* daripada *cost* yang semestinya.

Pada implementasi solver puzzle Rush Hour ini, kami menggunakan dua jenis heuristik. Berikut merupakan penjabaran dari heuristik yang kami gunakan:

1) Jumlah piece yang menghalangi jalan primary piece menuju exit pada tiga move ke depan

   Heuristik ini merupakan heuristik yang admissible. Heuristik ini merupakan *best case* dari perpindahan *piece*, di mana *piece* yang menghalangi dapat langsung dipindahkan sehingga tidak menghalangi *primary piece* menuju exit. Hal ini menunjukkan bahwa h(n) <= h*(n) pasti memenuhi dan h(n) tidak pernah melebih-lebihkan atau *overestimate cost* pada suatu node. Selain itu, heuristik ini meninjau tiga *move* ke depan sehingga *minimum cost* yang diambil adalah *minimum cost* dari tiga *move* ke depan. Dengan meninjau beberapa *move* kedepan, kita dapat mendeteksi kasus di mana terdapat suatu node dengan *cost* kecil, tetapi *cost move-move* selanjutnya lebih besar dibandingkan hasil perpanjangan dari node lain.

2) Jarak primary piece menuju exit

   Heuristik ini tidak admissible. Heuristik ini hanya admissible pada kasus di mana semua jalur primary piece menuju exit diisi piece yang menghalangi primary piece. Dikatakan admissible pada kasus tersebut karena jarak piece menuju exit menggambarkan best case di mana kita dapat menggeser semua piece penghalang, lalu menggerakkan primary piece keluar.

Dari hasil analisis tersebut, kami menjadikan heuristik pertama sebagai heuristik utama karena admissible dan dapat memperkirakan solusi dengan lebih baik.

2.1.2. Analisis Teoritis Algoritma

Fungsi evaluasi atau f(n) menyatakan cost untuk menuju node n dari node saat ini. Pada *uninformed search*, f(n) dinyatakan juga sebagai g(n). g(n) adalah *cost* aktual dari *root* menuju node n. Pada implementasi ini, g(n) yang digunakan adalah jumlah *move* sehingga *cost* antara satu node ke node lain yang menyatakan state dari board pasti bernilai satu karena tetangga dari node adalah node lain yang bisa dicapai dengan satu *move*.

Penyelesaian puzzle Rush Hour dengan menggunakan UCS, pada kasus ini, sama dengan BFS. Hal ini disebabkan cara kerja algoritma UCS yang pasti mengunjungi node dengan *cost* terendah terlebih dahulu, di mana *cost* pada implementasi ini memiliki nilai yang sama dengan level atau aras sehingga urutan penelusuran UCS serupa dengan BFS yang mengutamakan penelusuran berdasarkan aras terkecil terlebih dahulu.

Secara teoritis, algoritma A* cenderung lebih efisien dibandingkan dengan algoritma Uniform Cost Search (UCS) pada penyelesaian puzzle Rush Hour. Hal ini disebabkan UCS hanya berpatokan pada nilai *cost* murni tanpa adanya informasi tambahan dari heuristik. Dengan adanya penggunaan heuristik pada A*, algoritma tersebut dapat memotong pencarian pada node dengan *cost* lebih besar dengan jumlah yang lebih banyak dibandingkan UCS yang hanya membandingkan *cost* murni.

Mengenai algoritma Greedy Best First Search, karena algoritma ini terus memaksakan jalur dengan Node yang saat ini minimum/maksimum lokal (dalam konteks algoritma kali ini adalah minimum lokal), maka algoritma ini tidak menjamin solusi yang paling optimal bahkan dengan heuristik sebaik apapun itu. Hal ini terjadi karena jalur dengan Node yang secara terus-menerus memiliki nilai minimum lokal belum tentu jalur yang paling optimal/minimum secara global. Bahkan algoritma ini belum tentu menghasilkan suatu solusi karena pada dasarnya kemungkinan jalur yang dieksplorasi hanyalah satu (tidak ada *backtracking*) walaupun sebelum memilih satu Node anak ia algoritma ini mengecek Node anak lainnya terlebih dahulu.

BAB III

IMPLEMENTASI DAN PENGUJIAN

3.1. Source Code Program

3.1.1. Heuristic.java

```java
package solver;

import obj.Board;

abstract public class Heuristic {
    abstract public int evaluate(Board b);
}
```

3.1.2. BaseHeuristic.java

```java
package solver;

import java.util.HashMap;

import obj.Board;
import obj.Piece;

// Heuristic: Jumlah piece yang menghalangi primary piece ke exit
public class BaseHeuristic extends Heuristic {

    @Override
    public int evaluate(Board b){
        HashMap<Character, Boolean> evaluatedPiece = new HashMap<>();

        int count = 0;
        Piece primaryPiece = b.getPiece(pieceChar:'P');
        if(primaryPiece == null) return Integer.MAX_VALUE;
        if(primaryPiece.isHorizontal()){
            if(b.getExitCol() < b.getStartColPiece(primaryPiece)){ // exit is left
                for(int j=1; j<b.getStartColPiece(primaryPiece); j++){
                    if(evaluatedPiece.get(b.getBoardState()[b.getExitRow()][j]) == null){
                        count++;
                        evaluatedPiece.put(b.getBoardState()[b.getExitRow()][j], value:true);
                    }
                }
            }else{ // exit is right
                for(int j=b.getStartColPiece(primaryPiece)+primaryPiece.getLen(); j<b.getExitCol(); j++){
                    if(evaluatedPiece.get(b.getBoardState()[b.getExitRow()][j]) == null){
                        count++;
                        evaluatedPiece.put(b.getBoardState()[b.getExitRow()][j], value:true);
                    }
                }
            }
```

```
        }else{
            if(b.getExitRow() < b.getStartRowPiece(primaryPiece)){ // exit is up
                for(int i=1; i<b.getStartRowPiece(primaryPiece); i++){
                    if(evaluatedPiece.get(b.getBoardState()[i][b.getExitCol()]) == null){
                        count++;
                        evaluatedPiece.put(b.getBoardState()[i][b.getExitCol()], value:true);
                    }
                }
            }else{ // exit is down
                for(int i=b.getStartRowPiece(primaryPiece)+primaryPiece.getLen(); i<b.getExitRow(); i++){
                    if(evaluatedPiece.get(b.getBoardState()[i][b.getExitCol()]) == null){
                        count++;
                        evaluatedPiece.put(b.getBoardState()[i][b.getExitCol()], value:true);
                    }
                }
            }
        }
        return count;
    }
}
```

3.1.3. TreeHeuristic.java

```
package solver;

import java.util.List;
import obj.Board;
import utils.Utils;

public class TreeHeuristic extends Heuristic {
    private final int MAX_DEPTH = 3;
    private final BaseHeuristic baseHeuristic = new BaseHeuristic();

    @Override
    public int evaluate(Board b) {
        return evaluateMinimum(b, depth:0, Integer.MAX_VALUE);
    }

    private int evaluateMinimum(Board board, int depth, int currentBestScore) {
        if (depth >= MAX_DEPTH || board.isFinished()) {
            return baseHeuristic.evaluate(board);
        }

        Utils utils = new Utils();
        List<Board> boards = utils.generateAllPossibleMoves(board);
        if (boards.isEmpty()) {
            return baseHeuristic.evaluate(board);
        }

        int bestScore = currentBestScore;
        for (Board b : boards) {
            if (b.isFinished()) {
                return 0;
            }
            int score = evaluateMinimum(b, depth + 1, bestScore);
            if (score < bestScore) {
                bestScore = score;
            }
        }
        return bestScore;
    }
}
```

3.1.4. UCS.java

```java
package solver;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
import obj.Board;
import utils.Utils;

public class UCS {
    private Utils utils;
    private int exploredNodes;
    private Scanner scanner;
    private Node solution;
    private Set<Board> visitedBoards;

    public UCS(Board initialBoard) {
        utils = new Utils();
        exploredNodes = 0;
        scanner = new Scanner(System.in);
        visitedBoards = new HashSet<Board>();
        solution = null;
        solve(initialBoard);
    }
```

```java
    public class Node implements Comparable<Node> {
        private Board state;
        private int cost;
        private Node parent;

        public Node(Board state, int cost, Node parent) {
            this.state = state;
            this.cost = cost;
            this.parent = parent;
        }

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

        public int getCost() {
            return cost;
        }
```

```java
        public Board getState() {
            return state;
        }

        public Node getParent() {
            return parent;
        }

        public List<Board> getPathOfBoards() {
            List<Board> path = new ArrayList<>();
            Node current = this;

            while(current != null){
                path.add(current.getState());
                current = current.getParent();
            }

            Collections.reverse(path);
            return path;
        }
    }
```

```
private void solve(Board initialBoard) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    pq.add(new Node(initialBoard, cost:0, parent:null));

    while(!pq.isEmpty()){
        Node current = pq.poll();
        Board currentBoard = current.getState();

        if(visitedBoards.contains(currentBoard)){
            continue;
        }

        exploredNodes++;
        visitedBoards.add(currentBoard);

        if(currentBoard.isFinished()){
            solution = current;
            break;
        }

        List<Board> nextBoards = utils.generateAllPossibleMoves(currentBoard, currentBoard.getLastMoves(),
        currentBoard.getLastDist(), currentBoard.getLastPiece());
```

```
        for(Board nextBoard : nextBoards){
            if(!visitedBoards.contains(nextBoard)){
                Node nextNode = new Node(nextBoard, current.getCost() + 1, current);
                pq.add(nextNode);
            }
        }
    }
}

public Node getSolution() {
    return solution;
}

public int getExploredNodesCount() {
    return exploredNodes;
}
```

```
public void printSolutionPath() {
    if(solution != null){
        List<Board> path = solution.getPathOfBoards();
        for(int i = 0; i < path.size(); i++){
            System.out.println("\nLangkah Ke-" + i + " (Papan ke-" + (i+1) + "):");
            path.get(i).printBoardState();
        }
        System.out.println("Kedalaman: " + (path.size() - 1) );
        System.out.println("Jumlah Node: " + exploredNodes );
    }else{
        System.out.println(x:"Tidak ada solusi yang ditemukan");
    }
}

public void writeSolution(String inputFileName) {
    if (solution != null) {
        String outputFileName = inputFileName.substring(beginIndex:0, inputFileName.lastIndexOf(ch:'.')) + "Solution.txt";
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFileName))) {
            writer.write(str:"================================================\n");
            writer.write(str:"         SOLUSI RUSH HOUR MENGGUNAKAN UCS          \n");
            writer.write(str:"================================================\n\n");
            writer.write("Total langkah: " + (solution.getPathOfBoards().size() - 1) + "\n");
            writer.write("Total node dieksplorasi: " + exploredNodes + "\n\n");

            for (int i = 0; i < solution.getPathOfBoards().size(); i++) {
                writer.write("Langkah Ke-" + i + "\n");
                Board board = solution.getPathOfBoards().get(i);
```

```
                char[][] boardState = board.getBoardState();
                int boardRow = board.getBoardRow();
                int boardCol = board.getBoardCol();
                int exitRow = board.getExitRow();
                int exitCol = board.getExitCol();

                if (exitRow == 0) {
                    writer.write(str:" ");
                    for (int j = 0; j < boardCol + 1; j++) {
                        if (j == exitCol) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }
                    writer.write(str:"\n");
                }
```

```
                for (int r = 1; r <= boardRow; r++) {
                    if (r != exitRow) {
                        writer.write(str:" ");
                    } else {
                        if (exitCol == 0) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }

                    for (int c = 1; c <= boardCol; c++) {
                        writer.write(boardState[r][c]);
                    }

                    if (r != exitRow) {
                        writer.write(str:" ");
                    } else {
                        if (exitCol == boardCol + 1) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }
                    writer.write(str:"\n");
                }
```

```
                if (exitRow == boardRow + 1) {
                    writer.write(str:" ");
                    for (int j = 0; j < boardCol + 1; j++) {
                        if (j == exitCol) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }
                    writer.write(str:"\n");
                }

                writer.write(str:"\n");
            }

            writer.write(str:"==================================================\n");
            writer.write(str:"                    SOLUSI SELESAI                \n");
            writer.write(str:"==================================================\n");

            System.out.println("Solusi berhasil disimpan ke file: " + outputFileName);
        } catch (IOException e) {
            System.out.println("Error saat menyimpan solusi ke file: " + e.getMessage());
        }
    } else {
        System.out.println(x:"Tidak ada solusi yang disimpan ke dalam file");
    }
}
}
```

3.1.5. GBFS.java

```java
package solver;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import utils.Utils;
import java.util.Scanner;
import obj.*;

public class GBFS {
    private List<Board> solutionPath;
    private Board currentBoard;
    private int nodeCount;
    private Set<Board> visitedBoards;
    private Scanner scanner;
    private TreeHeuristic th;
    private DistHeuristic dh;

    public GBFS(){
        solutionPath = new ArrayList<Board>();
        nodeCount = 0;
        visitedBoards = new HashSet<Board>();
        scanner = new Scanner(System.in);
        th = new TreeHeuristic();
        dh = new DistHeuristic();
    }
```

```java
    public void solve(Board initBoard, boolean isTreeHeuristic){
        Utils utils = new Utils();
        solutionPath.add(initBoard);
        currentBoard = new Board(initBoard);
        visitedBoards.add(currentBoard);
        int heuristicValue;

        while (!currentBoard.isFinished()) {
            List<Board> currentPossibleBoards = new ArrayList<Board>(utils.generateAllPossibleMoves(currentBoard,
            currentBoard.getLastMoves(), currentBoard.getLastDist(), currentBoard.getLastPiece()));
```

```java
            Board bestBoard = null;
            int minHeuristicValue = Integer.MAX_VALUE;

            for (Board nextBoard : currentPossibleBoards) {
                nodeCount++;

                if (nextBoard.isFinished()) {
                    bestBoard = nextBoard;
                    break;
                }

                if (isContain(nextBoard)) {
                    continue;
                }

                if(isTreeHeuristic) heuristicValue = th.evaluate(nextBoard);
                else heuristicValue = dh.evaluate(nextBoard);

                if (heuristicValue < minHeuristicValue) {
                    minHeuristicValue = heuristicValue;
                    bestBoard = nextBoard;
                }
            }
```

```
            if (bestBoard == null) {
                break;
            }

            currentBoard = bestBoard;
            visitedBoards.add(currentBoard);
            solutionPath.add(currentBoard);

            if (currentBoard.isFinished()) {
                scanner.nextLine();
                break;
            }
        }
        System.out.println("node: " + this.getNodeCount());
        scanner.nextLine();
    }
```

```
    public List<Board> getSolutionPath() {
        return solutionPath;
    }

    public boolean isSolutionFound() {
        return currentBoard.isFinished();
    }

    public int getSolutionSteps() {
        return solutionPath.size() - 1;
    }

    public int getNodeCount() {
        return nodeCount;
    }

    public boolean isContain(Board inpBoard){
        for(Board b : visitedBoards){
            if(inpBoard.isEqual(b)) return true;
        }
        return false;
    }
```

```
    public void printSolutionPath(){
        if (currentBoard.isFinished()) {
            System.out.println(x:"Solusi ditemukan!");
            System.out.print(s:"Tekan enter untuk lanjut ...");
            scanner.nextLine();
            for (int i = 0; i < solutionPath.size(); i++) {
                System.out.println("\nLangkah Ke-" + i );
                solutionPath.get(i).printBoardState();
            }
            System.out.println("Jumlah Node dieksplor: " + nodeCount );
        } else {
            System.out.println(x:"Tidak ada solusi yang ditemukan " );
        }
    }

    public void writeSolution(String inputFileName) {
        if (currentBoard.isFinished()) {
            String outputFileName = inputFileName.substring(beginIndex:0, inputFileName.lastIndexOf(ch:'.')) + "Solution.txt";
            try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFileName))) {
                writer.write(str:"=================================================\n");
                writer.write(str:"          SOLUSI RUSH HOUR MENGGUNAKAN GBFS          \n");
                writer.write(str:"=================================================\n\n");
                writer.write("Total langkah: " + (solutionPath.size() - 1) + "\n");
                writer.write("Total node dieksplorasi: " + nodeCount + "\n\n");

                for (int i = 0; i < solutionPath.size(); i++) {
                    writer.write("Langkah Ke-" + i + "\n");
                    Board board = solutionPath.get(i);

                    // Menulis board state ke file
                    char[][] boardState = board.getBoardState();
                    int boardRow = board.getBoardRow();
                    int boardCol = board.getBoardCol();
                    int exitRow = board.getExitRow();
                    int exitCol = board.getExitCol();
```

```
            if (exitRow == 0) {
                writer.write(str:" ");
                for (int j = 0; j < boardCol + 1; j++) {
                    if (j == exitCol) {
                        writer.write(str:"K");
                    } else {
                        writer.write(str:" ");
                    }
                }
                writer.write(str:"\n");
            }

            for (int r = 1; r <= boardRow; r++) {
                if (r != exitRow) {
                    writer.write(str:" ");
                } else {
                    if (exitCol == 0) {
                        writer.write(str:"K");
                    } else {
                        writer.write(str:" ");
                    }
                }

                for (int c = 1; c <= boardCol; c++) {
                    writer.write(boardState[r][c]);
                }

                if (r != exitRow) {
                    writer.write(str:" ");
                } else {
                    if (exitCol == boardCol + 1) {
                        writer.write(str:"K");
                    } else {
                        writer.write(str:" ");
                    }
                }
                writer.write(str:"\n");
            }
```

```
                if (exitRow == boardRow + 1) {
                    writer.write(str:" ");
                    for (int j = 0; j < boardCol + 1; j++) {
                        if (j == exitCol) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }
                    writer.write(str:"\n");
                }

                writer.write(str:"\n");
            }

            writer.write(str:"==================================================\n");
            writer.write(str:"                    SOLUSI SELESAI                \n");
            writer.write(str:"==================================================\n");

            System.out.println("Solusi berhasil disimpan ke file: " + outputFileName);
        } catch (IOException e) {
            System.out.println("Error saat menyimpan solusi ke file: " + e.getMessage());
        }
    } else {
        System.out.println(x:"Tidak ada solusi yang disimpan ke dalam file");
    }
}
```

3.1.6. AStar.java

```java
package solver;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
import obj.Board;
import utils.Utils;

public class AStar {
    private Utils utils;
    private int exploredNodes;
    private Scanner scanner;
    private Node solution;
    private Set<Board> visitedBoards;
    private TreeHeuristic th;
    private DistHeuristic dh;
    private int heuristicValue;

    public AStar(Board initialBoard, boolean isTreeHeuristic) {
        utils = new Utils();
        exploredNodes = 0;
        scanner = new Scanner(System.in);
        visitedBoards = new HashSet<Board>();
        solution = null;
        th = new TreeHeuristic();
        dh = new DistHeuristic();
        solve(initialBoard, isTreeHeuristic);

    }
```

```java
    public class Node implements Comparable<Node> {
        private Board state;
        private int cost;
        private Node parent;

        public Node(Board state, int cost, Node parent) {
            this.state = state;
            this.cost = cost;
            this.parent = parent;
        }

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

```java
        public int getCost() {
            return cost;
        }

        public Board getState() {
            return state;
        }

        public Node getParent() {
            return parent;
        }

        public List<Board> getPathOfBoards() {
            List<Board> path = new ArrayList<>();
            Node current = this;

            while(current != null){
                path.add(current.getState());
                current = current.getParent();
            }

            Collections.reverse(path);
            return path;
        }
    }
```

```java
private void solve(Board initialBoard, boolean isTreeHeuristic) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    if(isTreeHeuristic) heuristicValue = th.evaluate(initialBoard);
    else heuristicValue = dh.evaluate(initialBoard);
    pq.add(new Node(initialBoard, heuristicValue, parent:null));

    while(!pq.isEmpty()){
        Node current = pq.poll();
        Board currentBoard = current.getState();

        if(visitedBoards.contains(currentBoard)){
            continue;
        }
        exploredNodes++;
        visitedBoards.add(currentBoard);
        if(currentBoard.isFinished()){
            solution = current;
            break;
        }

        List<Board> nextBoards = utils.generateAllPossibleMoves(currentBoard, currentBoard.getLastMoves(),
        currentBoard.getLastDist(), currentBoard.getLastPiece());
        for(Board nextBoard : nextBoards){
            if(!visitedBoards.contains(nextBoard)){
                if(isTreeHeuristic) heuristicValue = th.evaluate(nextBoard);
                else heuristicValue = dh.evaluate(nextBoard);
                Node nextNode = new Node(nextBoard, current.getCost()+1+heuristicValue, current);
                pq.add(nextNode);
            }
        }
    }
}
```

```java
public Node getSolution() {
    return solution;
}

public int getExploredNodesCount() {
    return exploredNodes;
}

public void printSolutionPath() {
    if(solution != null){
        List<Board> path = solution.getPathOfBoards();
        for(int i = 0; i < path.size(); i++){
            System.out.println("\nLangkah Ke-" + i + " (Papan ke-" + (i+1) + "):");
            path.get(i).printBoardState();
        }
        System.out.println("Kedalaman: " + (path.size() - 1) );
        System.out.println("Jumlah Node: " + exploredNodes );
    }else{
        System.out.println(x:"Tidak ada solusi yang ditemukan");
    }
}

public void writeSolution(String inputFileName) {
    if (solution != null) {
        String outputFileName = inputFileName.substring(beginIndex:0, inputFileName.lastIndexOf(ch:'.')) + "Solution.txt";
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFileName))) {
            writer.write(str:"================================================\n");
            writer.write(str:"          SOLUSI RUSH HOUR MENGGUNAKAN A*           \n");
            writer.write(str:"================================================\n\n");
            writer.write("Total langkah: " + (solution.getPathOfBoards().size() - 1) + "\n");
            writer.write("Total node dieksplorasi: " + exploredNodes + "\n\n");
```

```java
            for (int i = 0; i < solution.getPathOfBoards().size(); i++) {
                writer.write("Langkah Ke-" + i + "\n");
                Board board = solution.getPathOfBoards().get(i);

                char[][] boardState = board.getBoardState();
                int boardRow = board.getBoardRow();
                int boardCol = board.getBoardCol();
                int exitRow = board.getExitRow();
                int exitCol = board.getExitCol();

                if (exitRow == 0) {
                    writer.write(str:" ");
                    for (int j = 0; j < boardCol + 1; j++) {
                        if (j == exitCol) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }
                    writer.write(str:"\n");
                }
```

```
                for (int r = 1; r <= boardRow; r++) {
                    if (r != exitRow) {
                        writer.write(str:" ");
                    } else {
                        if (exitCol == 0) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }
```

```
                    for (int c = 1; c <= boardCol; c++) {
                        writer.write(boardState[r][c]);
                    }

                    if (r != exitRow) {
                        writer.write(str:" ");
                    } else {
                        if (exitCol == boardCol + 1) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }
                    writer.write(str:"\n");
                }

                // Menulis exit bagian bawah jika ada
                if (exitRow == boardRow + 1) {
                    writer.write(str:" ");
                    for (int j = 0; j < boardCol + 1; j++) {
                        if (j == exitCol) {
                            writer.write(str:"K");
                        } else {
                            writer.write(str:" ");
                        }
                    }
                    writer.write(str:"\n");
                }

                writer.write(str:"\n");
            }
```

```
                writer.write(str:"==================================================\n");
                writer.write(str:"                    SOLUSI SELESAI                \n");
                writer.write(str:"==================================================\n");

                System.out.println("Solusi berhasil disimpan ke file: " + outputFileName);
            } catch (IOException e) {
                System.out.println("Error saat menyimpan solusi ke file: " + e.getMessage());
            }
        } else {
            System.out.println(x:"Tidak ada solusi yang disimpan ke dalam file");
        }
    }
}
```

3.2. Hasil Uji

3.2.1. Test Case 1 (1.txt)

Hasil Uji Ke-1

Input file	_13523007_1352: 6 6 11 AAB..F ..BCDF kGPPCdF Gh.iiI GHJ... LLJMM. Input valid dan dapat diselesaikan
Tampilan GUI	Uji GBFS Heuristik 1 
Output file	Langkah Ke-31 AABC.F ..BCDF KPP..DF GHJIII GHJ... GLLMM. ========== SOLUSI SELESAI ==========

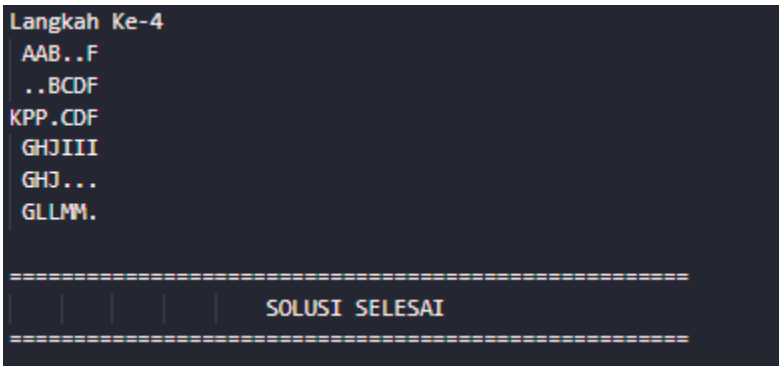
Hasil Uji Ke-2

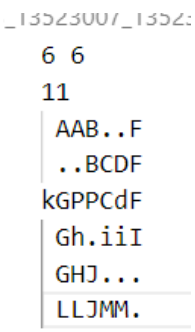
Input file	_13523007_1352: 6 6 11 AAB..F ..BCDF kGPPCdF Gh.iiI GHJ... LLJMM. Input valid dan dapat diselesaikan
Tampilan GUI	Uji GBFS Heuristik 2 
Output file	Langkah Ke-89 AAB..F ..BCDF KPP.CDF GHJIII GHJ... GLL.MM ========== SOLUSI SELESAI ==========

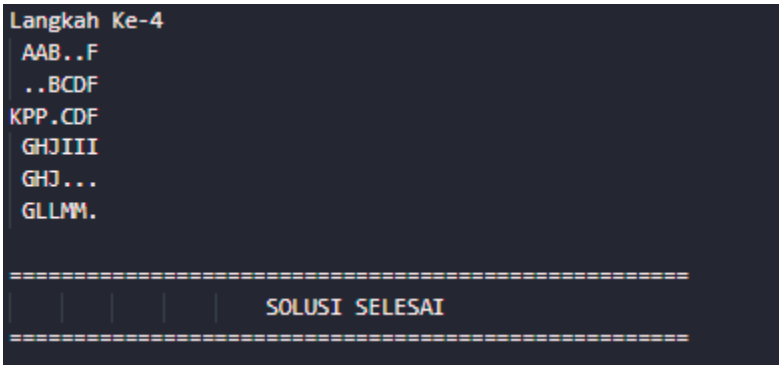
Hasil Uji Ke-3	
Input file	_13523007_1352: 6 6 11 AAB..F ..BCDF kGPPCdF Gh.iiI GHJ... LLJMM. Input valid dan dapat diselesaikan
Tampilan GUI	Uji IDS 

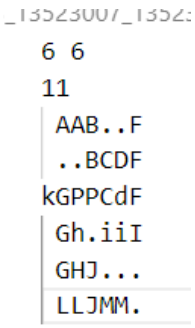
Output file	Langkah Ke-4 AAB..F ..BCDF KPP.CDF GHJIII GHJ... GLLMM. ========== SOLUSI SELESAI ==========

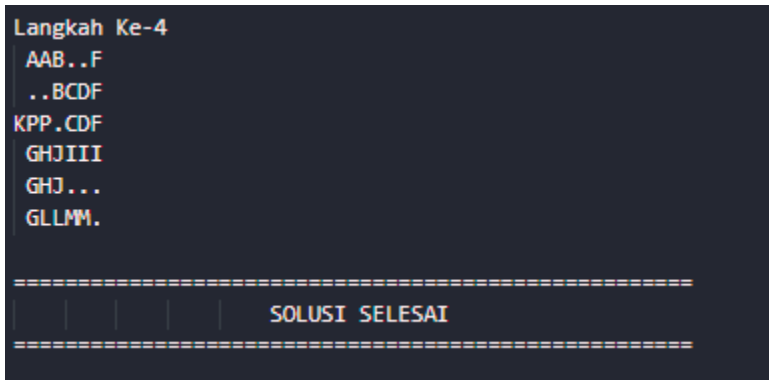
Hasil Uji Ke-4	
Input file	6 6 11 AAB..F ..BCDF kGPPCdF Gh.iiI GHJ... LLJMM. Input valid dan dapat diselesaikan
Tampilan GUI	Uji A* Heuristik 1

	
Output file	

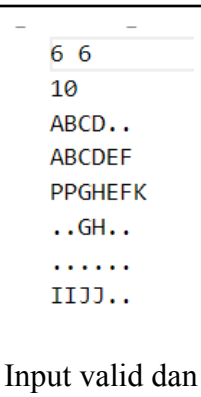
Hasil Uji Ke-5	
Input file	 Input valid dan dapat diselesaikan
Tampilan GUI	Uji A* Heuristik 2

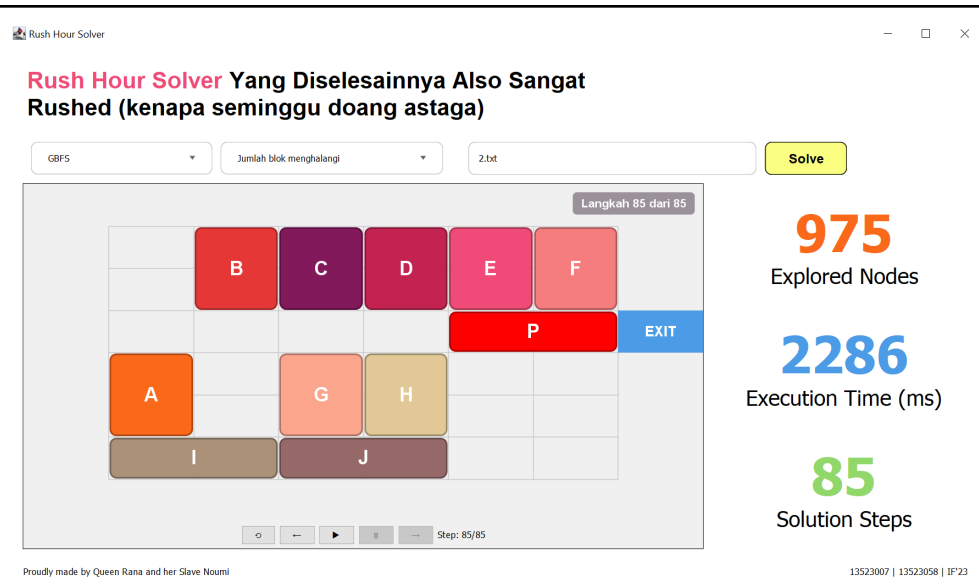
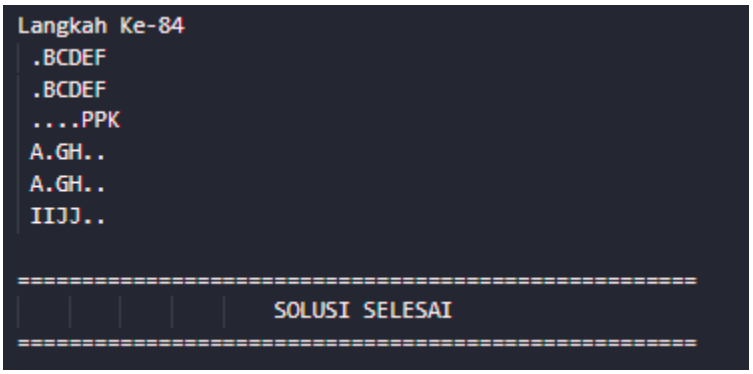
	
Output file	

Hasil Uji Ke-6	
Input file	 Input valid dan dapat diselesaikan
Tampilan GUI	Uji UCS

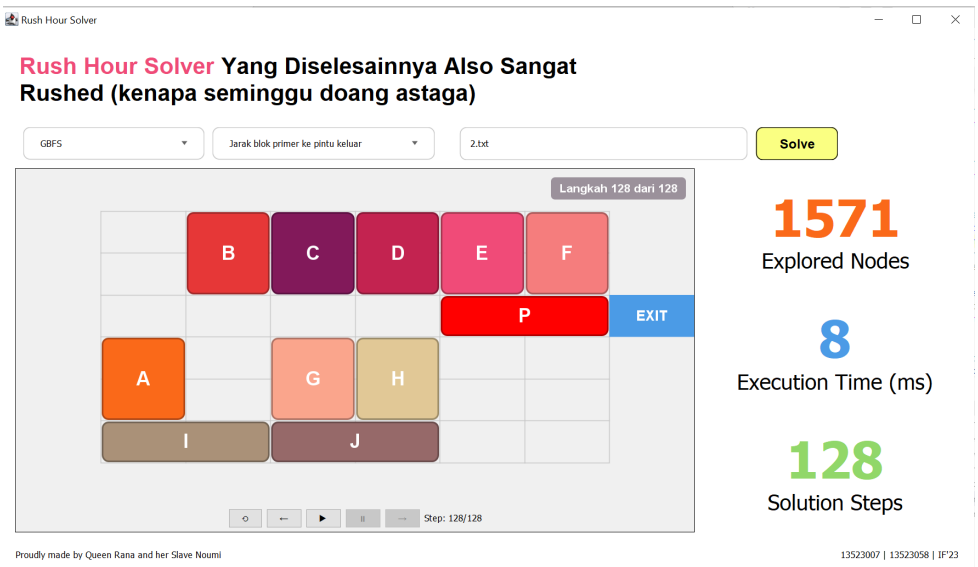
	
Output file	

3.2.2. Test Case 2 (2.txt)

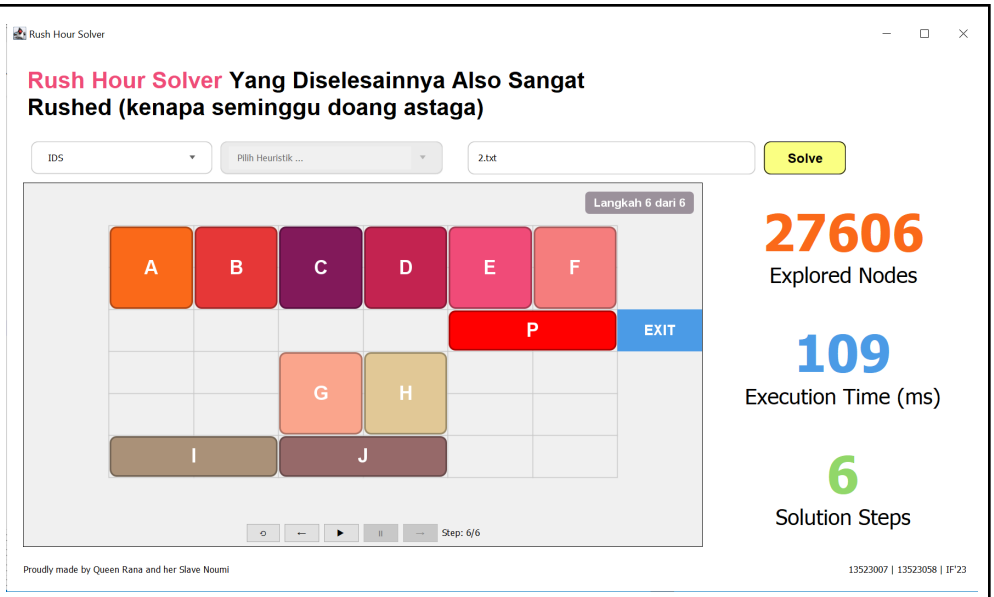
Hasil Uji Ke-1	
Input file	 Input valid dan dapat diselesaikan
Tampilan GUI	Uji GBFS Heuristik 1

	
Output file	

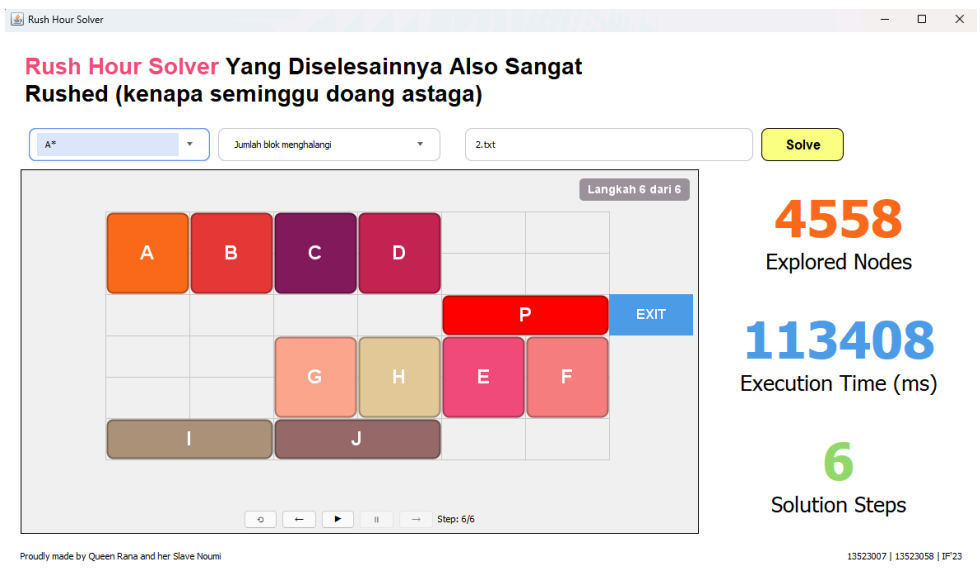
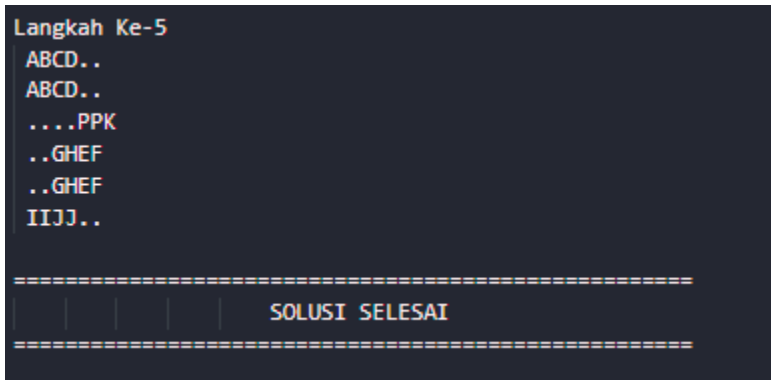
Hasil Uji Ke-2	
Input file	6 6 10 ABCD.. ABCDEF PPGHEFK ..GH.. ..... IIJJ.. Input valid dan dapat diselesaikan
Tampilan GUI	Uji GBFS Heuristik 2

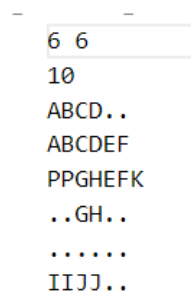
	
Output file	Langkah Ke-127 .BCDEF .BCDEF ....PPK A.GH.. A.GH.. IIJJ.. ========== SOLUSI SELESAI ==========

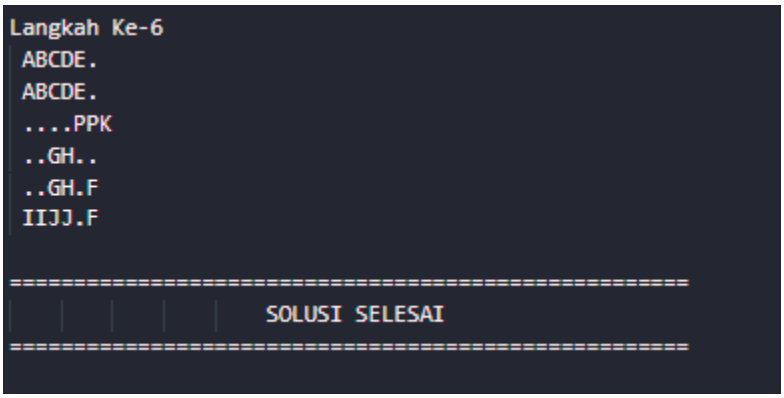
Hasil Uji Ke-3	
Input file	6 6 10 ABCD.. ABCDEF PPGHEFK ..GH.. ..... IIJJ.. Input valid dan dapat diselesaikan
Tampilan GUI	Uji IDS

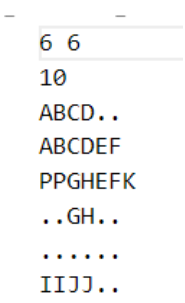
	
Output file	Langkah Ke-5 ABCDEF ABCDEF ....PPK ..GH.. ..GH.. IIJJ.. ========== SOLUSI SELESAI ==========

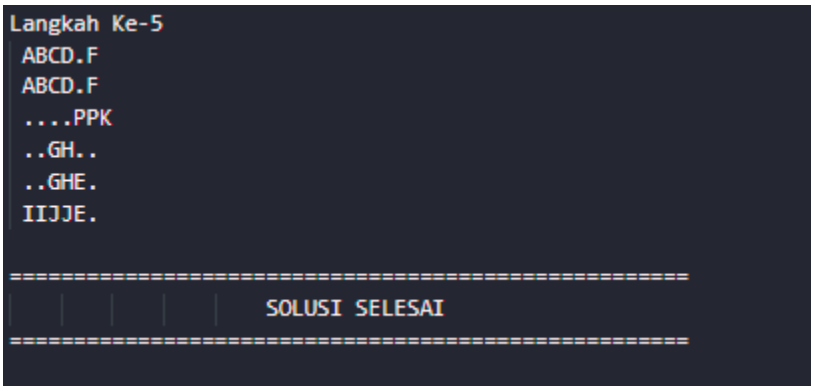
Hasil Uji Ke-4	
Input file	6 6 10 ABCD.. ABCDEF PPGHEFK ..GH.. ..... IIJJ.. Input valid dan dapat diselesaikan
Tampilan GUI	Uji A* Heuristik 1

	
Output file	

Hasil Uji Ke-5	
Input file	 Input valid dan dapat diselesaikan
Tampilan GUI	Uji A* Heuristik 2

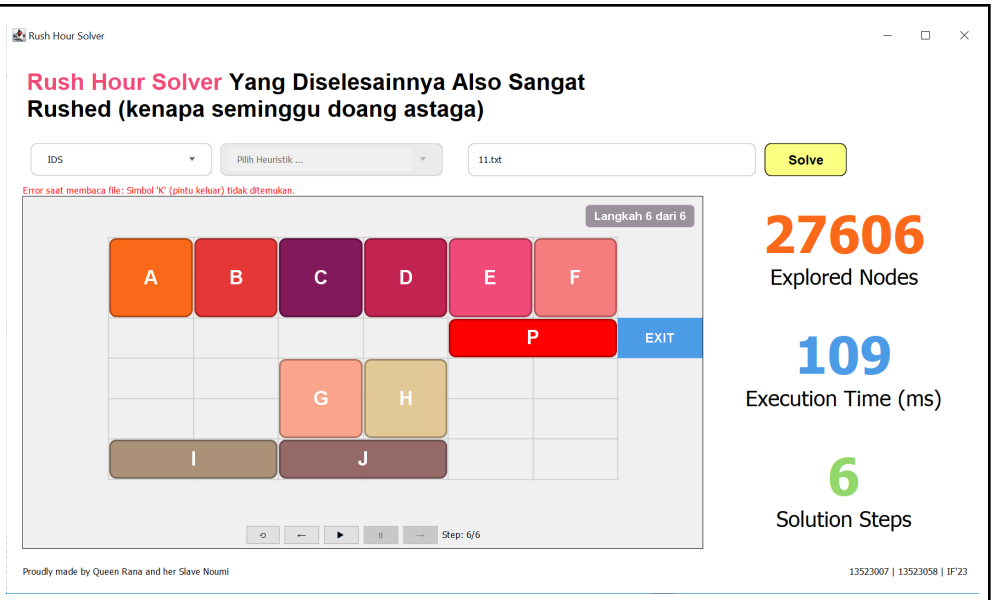
	
Output file	

Hasil Uji Ke-6	
Input file	 Input valid dan dapat diselesaikan
Tampilan GUI	Uji UCS

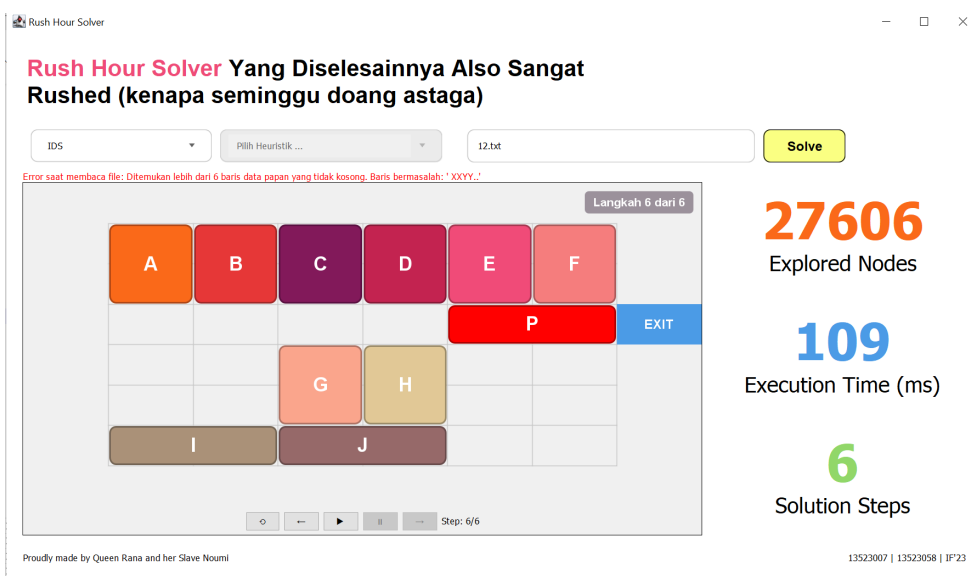
	
Output file	

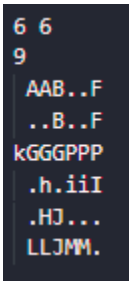
3.2.3. Test Case Input Invalid

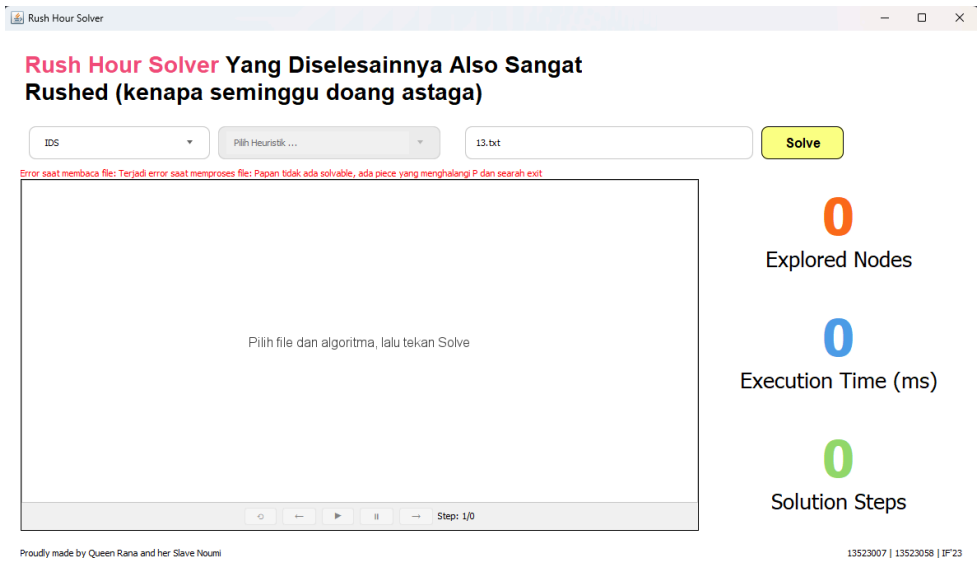
Hasil Uji Ke-1	
Input file	1 6 6 2 11 3 AAB..F 4 ..BCDF 5 GPPCdF 6 Gh.iiI 7 GHJ... 8 LLJMM. Input invalid

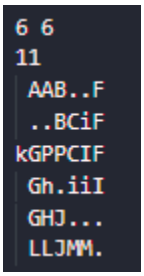
Tampilan GUI	 (hasil papan bekas penyelesaian sebelumnya yang valid)
Output file	-

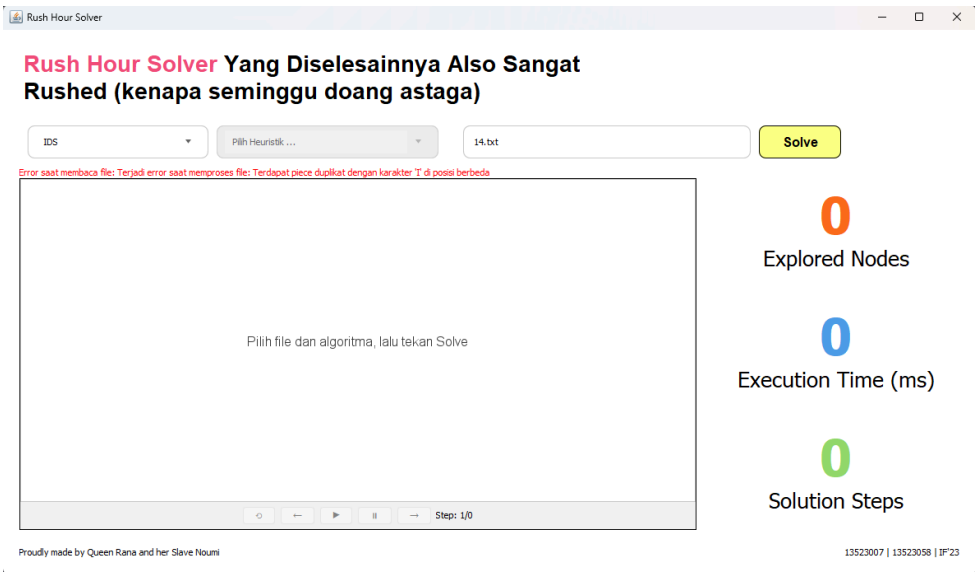
Hasil Uji Ke-2	
Input file	1 6 6 2 11 3 AAB..F 4 ..BCDF 5 kGPP.dF 6 Gh.iiI 7 GHJ... 8 LLJMM. 9 XXYY.. Input invalid

Tampilan GUI	 (hasil papan bekas penyelesaian sebelumnya yang valid)
Output file	-

Hasil Uji Ke-3	
Input file	 Input invalid

Tampilan GUI	
Output file	-

Hasil Uji Ke-4	
Input file	 Input invalid

Tampilan GUI	
Output file	-

3.3. Analisis Hasil Uji

Sebelum meninjau kompleksitas tiap algoritma tree-search, perlu dievaluasi terlebih dahulu kompleksitas dari masing-masing heuristik. Heuristik pertama, yaitu TreeHeuristic memiliki kompleksitas waktu sebesar $O(b^3)$ dengan b merupakan *branching factor* (state board), heuristik ini menjamin hasil yang lebih optimal, tetapi memiliki waktu pemrosesan yang cukup lama. Di sisi lain, heuristik kedua, yaitu DistHeuristic memiliki kompleksitas waktu sebesar O(1) sehingga heuristik ini jauh lebih cepat, tetapi hasilnya kurang optimal.

Terdapat empat algoritma pathfinding yang digunakan. Berikut perbandingan dari tiap algoritma pathfinding:

1) Uniform Cost Search (UCS)
   - Kompleksitas waktu:

     UCS memiliki kompleksitas waktu sebesar $O(b^d)$ di mana b adalah *branching factor* dan d adalah *depth*. UCS memiliki kompleksitas dengan besar tersebut karena menelusuri tiap *branch/node* di satu kedalaman, lalu berlanjut menelusuri node di kedalaman berikutnya.
   - Kompleksitas ruang:

     UCS memiliki kompleksitas ruang yang sama dengan kompleksitas waktunya, yakni $O(b^d)$ karena, pada kasus terburuk, UCS akan menelusuri semua node.

2) Greedy Best First Search (GBFS)
   - Kompleksitas waktu:

     GBFS pada dasarnya serupa dengan DFS, namun hanya menelusuri satu jalur dari percabangan awal yang memiliki cost terkecil. Maka, tanpa mempertimbangkan heuristik, GBFS memiliki kompleksitas waktu sebesar O(bd). Jika menggunakan TreeHeuristic, GBFS memiliki kompleksitas waktu sebesar O($b^3 d$) dan memiliki kompleksitas waktu sebesar O(bd) jika menggunakan DistHeuristic.
   - Kompleksitas ruang:

     GBFS memiliki kompleksitas ruang sebesar O(d) karena hanya menelusuri jalur dari node terkecil sampai mencapai tujuan.

3) A* Search
   - Kompleksitas waktu:

     A* Search memiliki kompleksitas waktu yang serupa dengan UCS, tetapi ditambah dengan mempertimbangkan heuristik. Jika menggunakan TreeHeuristic, maka kompleksitas waktu dari A* Search adalah sebesar O($b^{(d+3)}$). Sedangkan, jika menggunakan DistHeuristic, kompleksitas waktunya adalah O(b^d).
   - Kompleksitas ruang:

     A* Search memiliki kompleksitas ruang yang serupa dengan UCS, yaitu sebesar O(b^d).

4) Iterative Deepening Search (IDS)
   - Kompleksitas waktu:

     IDS merupakan kombinasi dari DFS dan BFS, di mana IDS mengambil kompleksitas waktu dari BFS yang lebih cepat daripada DFS dengan membatasi kedalaman dari searching *depth-based*. Maka, kompleksitas waktu IDS adalah O(b^d).
   - Kompleksitas ruang:

     IDS mengadopsi kompleksitas ruang dari DFS yang lebih kecil dibandingkan BFS karena menelusuri satu *branch* dulu sampai batas kedalaman sehingga ruang yang digunakan lebih sedikit daripada BFS. Maka, kompleksitas ruang dari IDS adalah O(bd).

BAB IV

IMPLEMENTASI BONUS

4.1. Penjelasan Bonus Algoritma Tambahan: *Iterative Deepening Search* (IDS)

Berikut adalah *source code* program untuk implementasi algoritma IDS

IDSNode.java

```java
package solver;

import java.util.ArrayList;
import java.util.List;
import obj.*;

public class IDSNode {
    private Board currentBoard;
    private List<Board> pathOfBoards;
    private int depth;

    public IDSNode(Board inpBoard, List<Board> inpBoards, int depth){
        this.currentBoard = inpBoard;
        this.pathOfBoards = new ArrayList<Board>(inpBoards);
        this.depth = depth;
    }

    IDSNode(Board inpBoard){
        this.currentBoard = inpBoard;
        this.pathOfBoards = new ArrayList<Board>();
        pathOfBoards.add(inpBoard);
        this.depth = 0;
    }

    IDSNode(IDSNode prevNode){
        this.currentBoard = prevNode.currentBoard;
        this.pathOfBoards = new ArrayList<Board>(prevNode.pathOfBoards);
        this.depth = prevNode.depth;
    }

    IDSNode (Board appendBoard, IDSNode prevNode){
        this.pathOfBoards = new ArrayList<Board>(prevNode.getPathOfBoards());
        this.pathOfBoards.add(appendBoard);
        this.currentBoard = appendBoard;
        this.depth = prevNode.getDepth()+1;
    }
```

```
    public Board getCurrentBoard(){
        return currentBoard;
    }

    public List<Board> getPathOfBoards(){
        return pathOfBoards;
    }

    public int getDepth(){
        return depth;
    }
}
```

IDS.java

```
package solver;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

import utils.*;

import obj.Board;

public class IDS {
    private Stack<IDSNode> treeStack;
    private int curMaxDepth;
    private IDSNode solution;
    private int exploredNodes;
    private boolean isFinished;
```

```java
public IDS(Board initBoard){
    this.solution = null;          // Menyimpan node solusi jika ditemukan
    this.exploredNodes = 0;        // Menghitung node yang dieksplorasi/digenerate
    this.curMaxDepth = 0;          // Batas kedalaman dimulai dari 0 untuk Iterative Deepening
    this.isFinished = false;

    Utils utils = new Utils();

    IDSNode rootNode = new IDSNode(initBoard); // Membuat node awal (kedalaman 0


    while (this.solution == null) {
        this.treeStack = new Stack<IDSNode>();
        this.treeStack.push(rootNode);

        while (!this.treeStack.isEmpty()) {
            IDSNode currentNode = this.treeStack.pop();
            Board currentBoardState = currentNode.getCurrentBoard();

            if (currentBoardState.isFinished()) {
                this.solution = currentNode;
                this.isFinished = true;
                break;
            }

            if (currentNode.getDepth() < this.curMaxDepth) {

                List<Board> nextPossibleBoards;
                nextPossibleBoards = utils.generateAllPossibleMoves(currentBoardState,
                currentBoardState.getLastMoves(),
                currentBoardState.getLastDist(), currentBoardState.getLastPiece());


                for (int i = nextPossibleBoards.size() - 1; i >= 0; i--) {
                    Board nextBoard = nextPossibleBoards.get(i);
                    IDSNode childNode = new IDSNode(nextBoard, currentNode);
                    this.treeStack.push(childNode);
                    this.exploredNodes++;
                }
            }
        }

        if (this.solution != null) {
            break;
        }
```

```
        this.curMaxDepth++;

        if (this.curMaxDepth > 30) {
            System.out.println("IDS mencapai batas kedalaman praktis (" + this.curMaxDepth + ") tanpa menemukan solusi.");
            break;
        }
    }

}

public IDSNode getSolution() {
    return this.solution;
}

public int getExploredNodesCount() {
    return this.exploredNodes;
}

public int getMaxDepthReached() {
    return this.curMaxDepth;
}

public void printSolutionPath() {
    if (this.solution != null) {
        List<Board> path = solution.getPathOfBoards();
        for (int i = 0; i < path.size(); i++) {
            System.out.println("\nLangkah Ke-" + i + " (Papan ke-" + (i+1) + "):");
            path.get(i).printBoardState();
        }
        System.out.println("Kedalaman: " + this.solution.getDepth() );
        System.out.println("Jumlah Node: " + exploredNodes);
    } else {
        System.out.println("Tidak ada solusi yang ditemukan hingga kedalaman maksimum yang dijelajahi: " + this.curMaxDepth);
    }
}
```

Algoritma ini menjamin solusi optimal karena algoritma ini pada dasarnya adalah sekumpulan DFS dengan setiap DFS dibatasi suatu kedalaman yang dilakukan secara berurutan dengan kedalaman maksimal yang terus bertambah, sehingga ketika di kedalaman saat ini juga ditemukan solusi, maka sudah dipastikan solusi tersebut adalah solusi dengan kedalaman terpendek yang ekivalen dengan langkah penyelesaian papan yang paling sedikit.

4.1.2. Penjelasan Bonus Heuristik Tambahan: Heuristik Berdasarkan Jarak

Berikut adalah *source code* program untuk implementasi algoritma heuristik berdasarkan jarak.

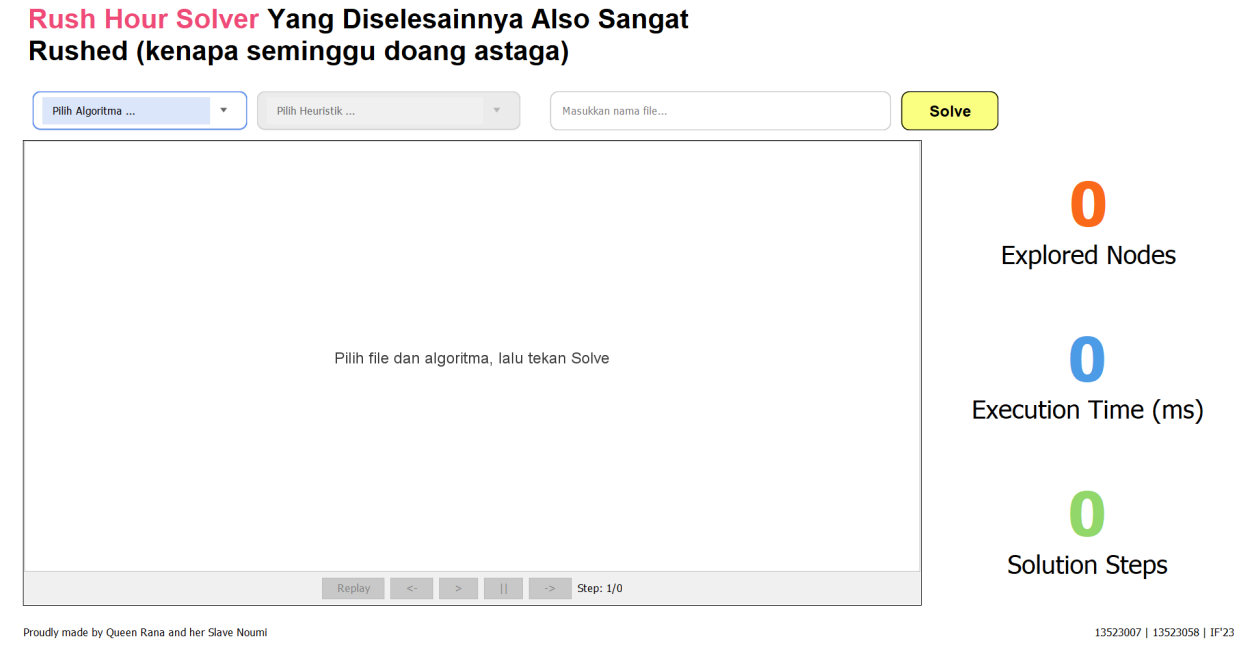
DistHeuristic.java

```java
package solver;

import obj.Board;
import obj.Piece;

// Heuristic: Jarak primary piece ke exit
public class DistHeuristic extends Heuristic {

    @Override
    public int evaluate(Board b){
        Piece primaryPiece = b.getPiece(pieceChar:'P');
        if(primaryPiece == null) return Integer.MAX_VALUE;
        if(primaryPiece.isHorizontal()){
            if(b.getExitCol() < b.getStartColPiece(primaryPiece)){ // exit is left
                return b.getStartColPiece(primaryPiece)-1;
            }else{ // exit is right
                return b.getExitCol()-(b.getStartColPiece(primaryPiece)+primaryPiece.getLen());
            }
        }else{
            if(b.getExitRow() < b.getStartRowPiece(primaryPiece)){ // exit is up
                return b.getStartRowPiece(primaryPiece)-1;
            }else{ // exit is down
                return b.getExitRow()-(b.getStartRowPiece(primaryPiece)+primaryPiece.getLen());
            }
        }
    }
}
```

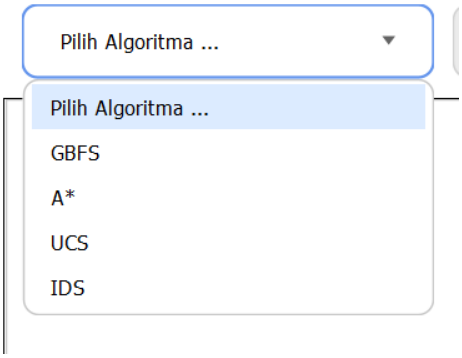
Penjelasan lebih lengkap tentang heuristik ini telah dibahas pada bagian 2.1.1. Analisis Fungsi Heuristik Pencarian.

4.1.3. Penjelasan Bonus GUI

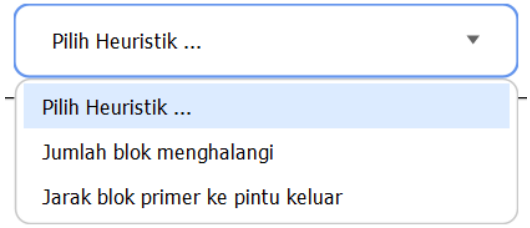
Penyusunan GUI menggunakan library Swing. Berikut adalah tampilan dan penjelasan perkomponen GUI kami.



- **Dropdown Pilihan Algoritma**
  Merupakan dropdown yang memberikan penggunanya untuk memilih algoritma apa yang ingin digunakan. Pilihan algoritma terdiri dari GBFS, A*, UCS, dan IDS. Berikut tampilan dropdown setelah di-expand.



- **Dropdown Pilihan Heuristik**
  Merupakan dropdown yang memberikan penggunanya untuk memilih heuristik apa yang ingin digunakan. Hanya algoritma GBFS dan A* yang memiliki pilihan heuristik. Pilihan heuristik terdiri dari jumlah blok menghalangi dan jarak blok primer ke pintu keluar. Berikut tampilan dropdown setelah di-expand.



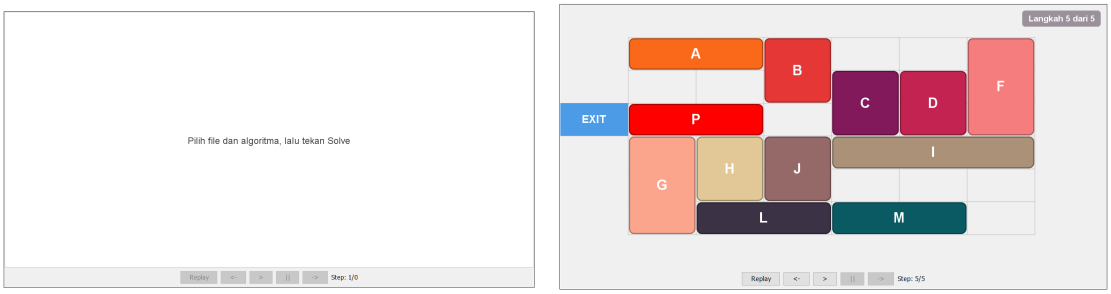
- **Input Field Nama File**

  Merupakan *input field* yang untuk pengguna memasukkan .txt test case yang ingin diselesaikan. Posisi path sudah disesuaikan sehingga hanya mengecek file dalam folder test.

- **Tombol Solve**

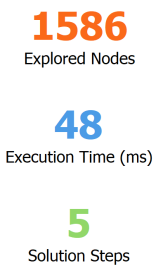
  Merupakan *button* yang ketika ditekan akan memulai algoritma penyelesaian papan sesuai algoritma dan heuristik yang dipilih.

- **Animasi Langkah Solusi**



  Merupakan bagian yang menampilkan animasi langkah solusi ketika solusi didapatkan. Terdapat beberapa tombol *player* di bagian bawah untuk mengatur keberjalanan animasi. Pengguna juga dapat melihat kembali langkah-langkah sebelumnya menggunakan tombol tersebut. Di bagian atas kanan ditunjukkan juga langkah ke berapa dari total berapa langkah yang saat ini ditampilkan oleh GUI.

- **Penayangan Hasil**



Bagian ini berfungsi untuk menunjukkan keterangan-keterangan dari proses penyelesaian algoritma yaitu jumlah node yang dieksplorasi, waktu penyelesaian, dan jumlah langkah solusi yang dihasilkan.

BAB V

LAMPIRAN

5.1. Lampiran

5.1.1. Pranala Repository

github.com/numshv/Tucil3_13523007_13523058

5.1.2. Tabel Keterselesaian

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