IF2211 Strategi Algoritma

Laporan Tugas Kecil 3

oleh

Muhamad Nazih Najmudin 13523144

A. Deskripsi Tugas

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

a. Papan

Papan merupakan tempat permainan dimainkan. Papan terdiri atas cell, yaitu sebuah singular point dari papan. Sebuah piece akan menempati cell-cell pada papan. Ketika permainan dimulai, semua piece telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi piece dan orientasi, antara horizontal atau vertikal. Hanya primary piece yang dapat digerakkan keluar papan melewati pintu keluar. Piece yang bukan primary piece tidak dapat digerakkan keluar papan. Papan memiliki satu pintu keluar yang pasti berada di dinding papan dan sejajar dengan orientasi primary piece.

b. Piece

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

c. Primary Piece

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

d. Pintu Keluar

Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan

e. Gerakan

Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan

kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

B. Algoritma yang digunakan

a. *Uniform Cost Search* (UCS)

Algoritma UCS merupakan algoritma pencarian rute yang berdasarkan biaya untuk mencapai suatu simpul tertentu. Algoritma ini akan memproses simpul yang memiliki biaya terendah *g(n)* dihitung dari *root* ke *node* tersebut.

1. Kasus permasalahan diproses dalam bentuk pohon status yang terdiri dari simpul dan sisi. Simpul merupakan status dari permasalahan yang sedang ditinjau, dan sisi menyimpan informasi mengenai jarak antar simpul, sehingga *g(n)* atau jarak akar ke suatu simpul dapat diketahui.
2. Akar diambil sebagai simpul yang diproses untuk pertama kali.
3. Simpul diproses hingga seluruh simpul cabang dari simpul tersebut diketahui beserta nilai *g(n)*-nya.
4. Seluruh simpul yang merupakan cabang dari simpul yang diproses saat ini, dimasukkan ke dalam struktur data *priority queue* yang terurut menaik berdasarkan nilai *g(n)* dari simpul-simpul di dalamnya.
5. Setelah proses selesai, proses dilanjutkan dengan pemrosesan simpul yang merupakan *head* dari *priority queue* terkini.
6. Proses dilakukan secara iteratif untuk terus memproses *head* dari *priority queue* hingga menemukan simpul tujuan. Suatu permasalahan dapat dikatakan tidak memiliki solusi apabila *priority queue* telah kosong dan masih belum menemukan simpul yang merupakan solusi.
7. Jalur atau rute penyelesaian diperoleh dengan menelusuri ke atas dari simpul tujuan ke simpul akar, lalu dibalik untuk mendapatkan rute dari akar ke tujuan.

b. *Greedy Best First Search*

Algoritma GBFS merupakan algoritma pencarian rute yang berdasarkan kedekatan jarak atau biaya dari simpul tertentu ke simpul tujuan. Algoritma ini akan memproses simpul yang memiliki jarak ke tujuan terendah *h(n)* berdasarkan heuristik yang digunakan.

1. Kasus permasalahan diproses dalam bentuk pohon status yang terdiri dari simpul dan sisi. Simpul merupakan status dari permasalahan yang sedang ditinjau dan menyimpan informasi mengenai berdasarkan heuristik tertentu, sehingga *h(n)* atau jarak suatu simpul ke simpul solusi dapat diketahui.
2. Akar diambil sebagai simpul yang diproses untuk pertama kali.
3. Simpul diproses hingga seluruh simpul cabang dari simpul tersebut diketahui beserta nilai *h(n)*-nya.
4. Seluruh simpul yang merupakan cabang dari simpul yang diproses saat ini, dimasukkan ke dalam struktur data *priority queue* yang terurut menaik berdasarkan nilai *h(n)* dari simpul-simpul di dalamnya.
5. Setelah proses selesai, proses dilanjutkan dengan pemrosesan simpul yang merupakan *head* dari *priority queue* terkini.

6. Proses dilakukan secara iteratif untuk terus memproses *head* dari *priority queue* hingga menemukan simpul tujuan. Suatu permasalahan dapat dikatakan tidak memiliki solusi apabila *priority queue* telah kosong dan masih belum menemukan simpul yang merupakan solusi.
7. Jalur atau rute penyelesaian diperoleh dengan menelusuri ke atas dari simpul tujuan ke simpul akar, lalu dibalik untuk mendapatkan rute dari akar ke tujuan.

c. A* *Algorithm*

Algoritma A* merupakan gabungan dari algoritma UCS dan GBFS, yaitu pencarian rute dengan mempertimbangkan $f(n) = g(n) + h(n)$ minimum untuk mencapai simpul tujuan dari simpul n.

1. Kasus permasalahan diproses dalam bentuk pohon status yang terdiri dari simpul dan sisi. Simpul merupakan status dari permasalahan yang sedang ditinjau, dan sisi menyimpan informasi mengenai jarak antar simpul untuk mengetahui $g(n)$, beserta pertimbanag heuristik $h(n)$ ke simpul tujuan, sehingga $f(n)$ simpul tersebut dapat diketahui.
2. Akar diambil sebagai simpul yang diproses untuk pertama kali.
3. Simpul diproses hingga seluruh simpul cabang dari simpul tersebut diketahui beserta nilai $f(n)$-nya.
4. Seluruh simpul yang merupakan cabang dari simpul yang diproses saat ini, dimasukkan ke dalam struktur data *priority queue* yang terurut menaik berdasarkan nilai $f(n)$ dari simpul-simpul di dalamnya.
5. Setelah proses selesai, proses dilanjutkan dengan pemrosesan simpul yang merupakan *head* dari *priority queue* terkini.
6. Proses dilakukan secara iteratif untuk terus memproses *head* dari *priority queue* hingga menemukan simpul tujuan. Suatu permasalahan dapat dikatakan tidak memiliki solusi apabila *priority queue* telah kosong dan masih belum menemukan simpul yang merupakan solusi.
7. Jalur atau rute penyelesaian diperoleh dengan menelusuri ke atas dari simpul tujuan ke simpul akar, lalu dibalik untuk mendapatkan rute dari akar ke tujuan.

d. *Branch and Bound*

Algoritma *Branch and Bound* merupakan algoritma pencarian rute optimal dengan penelusuran pohon. Untuk mendapatkan rute yang optimal, B&B melakukan *pruning* cabang-cabang yang pasti tidak menghasilkan solusi optimal, yaitu dengan membandingkan biayanya dengan biaya terbaik saat ini. Pada implementasi ini, biaya dihitung berdasarkan $f(n) = g(n) + h(n)$ seperti A*.

1. Kasus permasalahan diproses dalam bentuk pohon status yang terdiri dari simpul dan sisi. Simpul merupakan status dari permasalahan yang sedang ditinjau, dan sisi menyimpan informasi mengenai jarak antar simpul untuk mengetahui $g(n)$, beserta pertimbanag heuristik $h(n)$ ke simpul tujuan, sehingga biaya $f(n)$ simpul tersebut dapat diketahui.
2. Akar diambil sebagai simpul yang diproses untuk pertama kali.

3. Simpul diproses hingga seluruh simpul cabang dari simpul tersebut diketahui beserta nilai *f(n)*-nya.
4. Tidak seluruh simpul cabang akan diproses. Hanya simpul-simpul cabang dengan biaya *f(n)* lebih kecil atau sama dengan simpul yang sedang diproses saat ini saja yang akan dipilih. Sisanya, cabang akan di-*prune* dan tidak di proses.
5. Simpul-simpul terpilih dimasukkan ke dalam struktur data *priority queue* yang terurut menaik berdasarkan nilai *f(n)* dari simpul-simpul di dalamnya.
6. Setelah proses selesai, proses dilanjutkan dengan pemrosesan simpul yang merupakan *head* dari *priority queue* terkini.
7. Proses dilakukan secara iteratif untuk terus memproses *head* dari *priority queue* hingga menemukan simpul tujuan. Suatu permasalahan dapat dikatakan tidak memiliki solusi apabila *priority queue* telah kosong dan masih belum menemukan simpul yang merupakan solusi.
8. Jalur atau rute penyelesaian diperoleh dengan menelusuri ke atas dari simpul tujuan ke simpul akar, lalu dibalik untuk mendapatkan rute dari akar ke tujuan.

C. Analisis Permainan Rush Hour

Penerapan algoritma-algoritma tersebut untuk menyelesaikan permainan Rush Hour dapat dianalisis sebagai berikut.

a. Definisi *f(n)*, *g(n)*, dan *h(n)*

Jarak *g(n)* merupakan jarak dari akar ke suatu simpul *n* pada pohon. Jarak *h(n)* merupakan jarak dari suatu simpul *n* ke simpul tujuan yang diperoleh berdasarkan heuristik *h*. Biaya *f(n)* merupakan penjumlahan dari *g(n)* dan *h(n)*. Pada kasus permainan Rush Hour, *g(n)* merupakan banyaknya langkah yang perlu diambil dari kondisi awal papan untuk mencapai kondisi tertentu. Adapun *h(n)* merupakan banyaknya langkah yang perlu diambil untuk membuat *primary piece* mencapai pintu keluar.

b. Analisis *Uniform Cost Search* (UCS)

Pada kasus permainan Rush Hour, jarak antar simpul bernilai konstan 1 langkah. Sehingga, jarak *g(n)* pada setiap simpulnya konsisten sesuai dengan kedalaman pohon, dan berlaku sama untuk setiap simpul pada kedalaman yang sama. Hal ini membuat algoritma UCS pada kasus permainan Rush Hour menjadi berperilaku sama seperti algoritma BFS (*Best First Search*).

c. Analisis *Greedy Best First Search*

Pada kasus permainan Rush Hour, GBFS tidak dapat menjamin solusi optimal. GBFS cenderung mengambil jalur yang sekilas terlihat optimal, yaitu simpul yang memiliki nilai heuristik rendah, tetapi sebenarnya dapat mengarah pada rute yang jauh lebih panjang atau bahkan buntu. Sifat *greedy* pada algoritma GBFS membuatnya rentan terjebak dalam minimum lokal, bukan minimum global, sehingga GBFS tidak dapat menjamin solusi optimal.

d. Analisis A* *Algorithm*

Admissible heuristics didefinisikan sebagai heuristik *h* yang untuk setiap simpul *n* pada kasus permasalahan, selalu memenuhi biaya $h(n) \leq h^*(n)$ dengan $h^*(n)$ adalah biaya sebenarnya yang diperlukan dari suatu simpul *n* ke simpul tujuan. Terdapat empat heuristik yang dapat dipilih dalam program, yaitu

1. jarak ke *piece* utama ke pintu keluar,
2. jumlah *piece* yang menghalangi pintu keluar,
3. jumlah *piece* penghalang pintu keluar yang terkunci, serta
4. banyak langkah yang diperlukan oleh seluruh *piece* yang menghalangi untuk mengosongkan jalan utama.

Heuristik juga dapat dikombinasikan dua hingga empat heuristik yang mencakup *max*, *min*, *sum*, dan *avg*. Pada kasus ini, dapat dijamin bahwa

$$h_3(n) \leq h_2(n) \leq h_1(n) \leq (h_1(n) + h_4(n)) \leq h^*(n)$$

Sehingga seluruh heuristik bersifat *admissible*. Secara matematis dapat dijamin pula bahwa $max(h_i(n)) \leq h^*(n)$, $min(h_i(n)) \leq h^*(n)$, dan $avg(h_i(n)) \leq h^*(n)$. Adapun $sum(h_i(n))$ tidak dapat dijamin *admissible*.

Secara teori, algoritma A* yang menggunakan heuristik *admissible* bekerja lebih baik dibanding UCS yang bekerja tanpa heuristik, terlebih pada kasus permainan Rush Hour, algoritma UCS berperilaku seperti BFS biasa. Hal ini dikarenakan heuristik yang digunakan pada algoritma A* dapat menuntun untuk mendapatkan solusi optimal.

e. Analisis *Branch and Bound*

Menyelesaikan persoalan permainan Rush Hour dengan algoritma B&B tidak jauh berbeda dari algoritma UCS, GBFS, dan A*. Setiap kondisi papan direpresentasikan sebagai simpul pada pohon, dengan jarak atau biaya antar simpul konstan sebesar 1 langkah, serta dapat mempertimbangkan biaya heuristik dari simpul tersebut ke simpul tujuan. Sebagai tambahan, B&B membatasi simpul yang diproses berdasarkan pembatasan biaya *f(n)* agar selalu merupakan nilai yang minimum. Dengan B&B, solusi dapat dipastikan optimal tergantung pada heuristik yang digunakan.

D. Kode Sumber

Pranala repositori: https://github.com/nazihstei/Tucil3_13523144/releases/latest

a. Struktur Repositori

Tucil3_13523144/

```
Tucil3_13523144/
├── README.md*
├── bin/
├── doc/
│   └── initial commit*
├── lib/
│   └── jansi-2.4.0.jar*
├── output.txt*
├── run.cmd*
├── run.sh*
├── src/
│   ├── Main.java*
```

```
|    ├── model/
|    |    ├── Block.java*
|    |    ├── Board.java*
|    |    ├── BoardTree.java*
|    |    └── Piece.java*
|    ├── solver/
|    |    ├── algorithm/
|    |    |    ├── ASTAR.java*
|    |    |    ├── Algorithm.java*
|    |    |    ├── BNB.java*
|    |    |    ├── GBFS.java*
|    |    |    ├── IRoutePlanning.java*
|    |    |    └── UCS.java*
|    |    └── heuristic/
|    |         ├── BlockingPieceCount.java*
|    |         ├── CombinedHeuristic.java*
|    |         ├── DistanceToExit.java*
|    |         ├── Heuristic.java*
|    |         ├── MovementToMakeWay.java*
|    |         └── SolidnessBlockingPiece.java*
|    └── utils/
|         ├── DualOutput.java*
|         ├── FileHandler.java*
|         ├── Home.java*
|         └── Terminal.java*
└── test/
```

b. Main.java

Main.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.fusesource.jansi.AnsiConsole;

import model.*;
import solver.algorithm.*;
import solver.heuristic.*;
import utils.*;

public class Main {

    /* MAIN PROGRAM */
    public static void main(String[] args) throws InterruptedException {

        /* PREPARATION */
        AnsiConsole.systemInstall();
        Scanner input = new Scanner(System.in);

        try {
            /* LOAD CONFIGURATION FILE */
            Thread.sleep(0);
            Board rootBoard = null;
            while (rootBoard == null) {
                Home.Header();
                System.out.println("[*] Silahkan masukkan file konfigurasi");
                System.out.print  ("    >> ./test/");
                String filepath = "./test/" + input.nextLine();
                System.out.println();
                System.out.println("[*] Memuat konfigurasi ...");
                Thread.sleep(1000);
                rootBoard = FileHandler.loadFile(filepath);
                if (rootBoard!= null && !rootBoard.isExitValid()) {
                    System.err.println("[X] EXIT TIDAK VALID");
                    rootBoard = null;
                }
                Home.Footer();
            }
            Home.Header();
            System.out.println("[*] Konfigurasi berhasil dimuat!");
```

```
            Home.Footer();

            /* INPUT ALGORITHM AND HAURISTIC */
            String algorithmInput = null;
            String heuristicInput = null;
            while (algorithmInput == null) {
                Home.Header();
                System.out.println("[*] Berikut adalah algorithma yang dapat dipilih:");
                System.out.println("    [1] UCS  : Uniform Cost Search");
                System.out.println("    [2] GBFS : Greedy Best-First Search");
                System.out.println("    [3] A*   : A star");
                System.out.println("    [4] B&B  : Branch and Bound");
                System.out.println();
                System.out.println("[*] Silahkan pilih algorithma yang ingin digunakan");
                System.out.println("[-] Keterangan  : masukkan nomornya");
                System.out.print  ("    >> ");
                algorithmInput = input.nextLine().trim();
                // System.out.println();
                Home.Footer();
                if (!(  algorithmInput.equals("1") ||
                        algorithmInput.equals("2") ||
                        algorithmInput.equals("3") ||
                        algorithmInput.equals("4"))) algorithmInput = null;
            }
            Home.Header();
            switch (algorithmInput) {
                case "1" :
                    System.out.println("[*] Algorithma UCS tidak menggunakan heuristik");
                    Home.Footer();
                    break;
                case "4" :
                    String useHeuristicChoice = null;
                    while (useHeuristicChoice == null) {
                        System.out.println("[*] Apakah ingin menggunakan heuristik? (Y/N)");
                        System.out.print  ("    >> ");
                        useHeuristicChoice = input.nextLine().toUpperCase().trim();
                        if (!(  useHeuristicChoice.equals("Y") ||
                                useHeuristicChoice.equals("N"))) useHeuristicChoice = null;
                    }
                    Home.Footer();
                    if (useHeuristicChoice.equals("N")) {
                        break;
                    }
                default :
                    heuristicInput = null;
                    while (heuristicInput == null) {
                        Home.Header();
                        System.out.println("[*] Berikut adalah heuristik yang dapat dipilih:");
                        System.out.println("    [1] Distance to Exit");
                        System.out.println("    [2] Count of Blocking Piece");
                        System.out.println("    [3] Solidness of Blocking Piece");
                        System.out.println("    [4] Movement to Make Way");
                        System.out.println("    [5] Kombinasi dari heuristik di atas");
                        System.out.println();
                        System.out.println("[*] Silahkan pilih heuristik yang ingin digunakan");
                        System.out.println("[-] Keterangan  : masukkan nomornya");
                        System.out.print  ("    >> ");
                        heuristicInput = input.nextLine().trim();
                        if (!(  heuristicInput.equals("1") ||
                        heuristicInput.equals("2") ||
                        heuristicInput.equals("3") ||
                        heuristicInput.equals("4") ||
                        heuristicInput.equals("5"))) heuristicInput = null;
                        if (heuristicInput!=null && heuristicInput.equals("5")) {
                            System.out.println();
                            System.out.println("[*] Silahkan pilih kombinasi heuristik yang ingin digunakan");
                            System.out.println("[+] Pilih juga bagaimana kombinasinya (MAX/MIN/SUM/AVG)");
                            System.out.println("[-] format input: mode h1 h2 h3 h4");
                            System.out.println("[-] Contoh      : MAX 1 3");
                            System.out.print  ("    >> ");
                            heuristicInput = input.nextLine().trim();
                            Pattern pattern = Pattern.compile("^(MAX|MIN|SUM|AVG)\\s+([1-4](?:\\s+[1-4]){0,3})$");
                            Matcher matcher  = pattern.matcher(heuristicInput);
                            if (!matcher.matches()) heuristicInput = null;
                        }
                        Home.Footer();
                    }
                    break;
            }

            /* SET ALGORITHM AND HEURISTIC */
            Algorithm solver = null;
            Heuristic heuristic = null;
            switch (heuristicInput) {
                case "1" -> {heuristic = new DistanceToExit(); break;}
                case "2" -> {heuristic = new BlockingPieceCount(); break;}
                case "3" -> {heuristic = new SolidnessBlockingPiece(); break;}
                case "4" -> {heuristic = new MovementToMakeWay(); break;}
                case null-> {break;}
                default  -> {
                    ArrayList<String> heuConf = new ArrayList<>(Arrays.asList(heuristicInput.split(" ")));
                    Heuristic[] heuList = new Heuristic[heuConf.size()-1];
                    for (int i=0; i<heuConf.size()-1; i++) {
                        Heuristic h = null;
                        switch (heuConf.get(i+1)) {
                            case "1" -> {h = new DistanceToExit(); break;}
```

```
                        case "2" -> {h = new BlockingPieceCount(); break;}
                        case "3" -> {h = new SolidnessBlockingPiece(); break;}
                        case "4" -> {h = new MovementToMakeWay(); break;}
                    }
                    heuList[i] = h;
                }
                heuristic = new CombinedHeuristic(heuList, heuConf.getFirst());
            }
        }
        switch (algorithmInput) {
            case "1" -> {solver = new UCS(rootBoard); break;}
            case "2" -> {solver = new GBFS(rootBoard, heuristic); break;}
            case "3" -> {solver = new ASTAR(rootBoard, heuristic); break;}
            default  -> {
                if (heuristic == null)  solver = new BNB(rootBoard);
                else                    solver = new BNB(rootBoard, heuristic);
                break;
            }
        }

        /* RUN SOLVER */
        Home.Header();
        String realtimeChoice = null;
        while (realtimeChoice == null) {
            System.out.println  ("[*] Aktifkan RealTime mode? (Y/N)");
            System.out.println  ("[-] Note: mode ini dapat memperlambat eksekusi");
            System.out.print    ("    >> ");
            realtimeChoice = input.nextLine().toUpperCase().trim();
            if (!(  realtimeChoice.equals("Y") ||
                    realtimeChoice.equals("N"))) realtimeChoice = null;
        }
        long startTime = 0;
        switch (realtimeChoice) {
            case "Y" -> {
                startTime = System.currentTimeMillis();
                solver.solve(true);
            }
            case "N" -> {
                Home.Header();
                System.out.println("[*] Please Wait ...");
                Home.Footer();
                startTime = System.currentTimeMillis();
                solver.solve(false);
            }
        }
        long endTime = System.currentTimeMillis();
        long durantion = endTime - startTime;

        /* PRINT OUTPUT TO FILE */
        Home.Header();
        System.out.println("[*] Menulis ke file output.txt ...");
        Home.Footer();
        DualOutput.activate("output.txt");
        Home.Solution(solver, durantion, false);
        DualOutput.deactivate();

        /* PRINT RESULT TO TERMINAL */
        Home.Solution(solver, durantion, true);
        System.out.println();
        System.out.println  ("[*] Tekan enter untuk menampilkan CLI animation");
        System.out.print    ("    >> ");
        input.nextLine();

        /* PRINT RESULT ANIMATION */
        Home.AnimateSolution(solver, durantion);

    } catch (Exception e) {
        DualOutput.deactivate();
        e.printStackTrace();

    } finally {
        /* TERMINATION */
        input.close();
    }
  }
}
```

c. Model

Block.java

```
package model;

public class Block implements Comparable<Block> {

    /* ATTRIBUTE */
    private char tag;
    private int row;
    private int col;

    /* CONSTRUCTOR */
```

```
    public Block(int row, int col) {
        this.tag = ' ';
        this.row = row;
        this.col = col;
    }
    public Block(char tag, int row, int col) {
        this.tag = tag;
        this.row = row;
        this.col = col;
    }

    /* COPY CONSTRUCTOR */
    public Block(Block b) {
        this.tag = b.tag;
        this.row = b.row;
        this.col = b.col;
    }

    /* GETTER and SETTER */
    public char getTag() {
        return this.tag;
    }
    public int getRow() {
        return this.row;
    }
    public int getCol() {
        return this.col;
    }
    public void setTag(char tag) {
        this.tag = tag;
    }
    public void setRow(int row) {
        this.row = row;
    }
    public void setCol(int col) {
        this.col = col;
    }

    /* CHECK */
    public Boolean isValid() {
        return this.tag != ' ';
    }
    public Boolean isEmpty() {
        return this.tag == '.';
    }
    public Boolean isExit() {
        return this.tag == 'K';
    }
    public Boolean isPiece() {
        return this.isValid() && !(this.isEmpty() || this.isExit());
    }
    public Boolean isPrimary() {
        return this.tag == 'P';
    }

    /* DISTANCE OPERATION */
    public int rowDistanceTo(Block b) {
        return this.row - b.row;
    }
    public int colDistanceTo(Block b) {
        return this.col - b.col;
    }
    public Double euclideanDistanceTo(Block b) {
        return Math.sqrt(Math.pow(this.rowDistanceTo(b), 2) + Math.pow(this.colDistanceTo(b), 2));
    }
    public int ManhattanDistance(Block b) {
        return Math.abs(this.row - b.row) + Math.abs(this.col - b.col);
    }

    /* PRINT BLOCK */
    @Override
    public String toString() {
        return String.format("(%c: %d, %d)", this.tag, this.row, this.col);
    }

    /* COMPARE */
    @Override
    public int compareTo(Block b) {
        int rowDiff = this.row - b.row;
        int colDiff = this.col - b.col;
        if (rowDiff < 0) {
            return -1;
        } else if (rowDiff > 0) {
            return 1;
        } else {
            if (colDiff < 0) {
                return -1;
            } else if (colDiff > 0) {
                return 1;
            } else {
                return 0;
            }
        }
    }

}
```

Board.java

```
package model;

import java.util.ArrayList;
import java.util.HashMap;

import org.fusesource.jansi.Ansi;

import model.Piece.Direction;

public class Board {

    /* ATTRIBUTE */
    private int row;
    private int col;
    private HashMap<Character, Piece> pieces;
    private ArrayList<ArrayList<Block>> map;
    private Block goal;
    private char moveTag;
    private String moveDirection;

    /* GETTER */
    public Piece getPrimaryPiece() {
        return this.pieces.get('P');
    }
    public Block getGoal() {
        return this.goal;
    }
    public ArrayList<ArrayList<Block>> getcopyMap() {
        ArrayList<ArrayList<Block>> result = new ArrayList<>();
        for (ArrayList<Block> mapRow : this.map) {
            ArrayList<Block> newMapRow = new ArrayList<>();
            for (Block blok : mapRow) {
                newMapRow.add(new Block(blok));
            }
            result.add(newMapRow);
        }
        return result;
    }
    public ArrayList<ArrayList<Block>> getMap() {
        return this.map;
    }

    /* CONSTRUCTOR */
    public Board(ArrayList<String> text) {
        // set row and col
        this.row = text.size();
        this.col = text.stream().mapToInt(String::length).max().orElse(0);
        this.pieces = new HashMap<>();
        this.map = new ArrayList<>();
        this.moveTag = '\0';
        this.moveDirection = "";

        // set pieces and map
        for (int i=0; i<this.row; i++) {
            this.map.add(new ArrayList<>());

            for (int j=0; j<this.col; j++) {

                // check if K in right side
                if (text.get(i).length() <= j) {
                    this.map.get(i).add(j, new Block(i, j));
                    // regular map
                } else {
                    this.map.get(i).add(j, new Block(text.get(i).charAt(j), i, j));
                }

                // add pieces
                Block addedBlock = this.map.get(i).get(j);
                if (addedBlock.isPiece()) {
                    if (this.pieces.containsKey(addedBlock.getTag())) {
                        this.pieces.get(addedBlock.getTag()).addBlock(addedBlock);
                    } else {
                        this.pieces.put(addedBlock.getTag(), new Piece(addedBlock));
                    }
                }
                // add goal
                if (addedBlock.isExit()) {
                    this.goal = addedBlock;
                }
            }
        }
    }

    /* COPY CONSTRUCTOR */
    public Board(Board b) {
        this.row = b.row;
        this.col = b.col;
        this.pieces = new HashMap<>();
        this.map = new ArrayList<>();
        this.moveTag = b.moveTag;
```

```
        this.moveDirection = b.moveDirection;

        // set pieces and map
        for (int i=0; i<this.row; i++) {
            this.map.add(new ArrayList<>());

            for (int j=0; j<this.col; j++) {
                this.map.get(i).add(j, new Block(b.map.get(i).get(j)));

                // add pieces
                Block addedBlock = this.map.get(i).get(j);
                if (addedBlock.isPiece()) {
                    if (this.pieces.containsKey(addedBlock.getTag())) {
                        this.pieces.get(addedBlock.getTag()).addBlock(addedBlock);
                    } else {
                        this.pieces.put(addedBlock.getTag(), new Piece(addedBlock));
                    }
                }
                // add goal
                if (addedBlock.isExit()) {
                    this.goal = addedBlock;
                }
            }
        }
    }

    /* VALIDITY CHECK */
    public Boolean haveCoordinate(int row, int col) {
        return  (row >= 0 && col >= 0)
                && (row < this.row)
                && (col < this.col);
    }
    public Boolean isExitValid() {
        if (this.goal == null) {
            return false;
        }
        int r = this.goal.getRow();
        int c = this.goal.getCol();
        int R = this.row-1;
        int C = this.col-1;
        Direction dir = this.getPrimaryPiece().getDirection();
        int pr = this.getPrimaryPiece().getHead().getRow();
        int pc = this.getPrimaryPiece().getHead().getCol();

        Boolean positionValid =
            (r == 0 && this.map.get(r+1).get(c).isValid()) ||
            (r == R && this.map.get(r-1).get(c).isValid()) ||
            (c == 0 && this.map.get(r).get(c+1).isValid()) ||
            (c == C && this.map.get(r).get(c-1).isValid());
        Boolean primaryValid =
            (dir.equals(Direction.HORIZONTAL) && r == pr) ||
            (dir.equals(Direction.VERTICAL) && c == pc);

            return positionValid && primaryValid;
    }
    public Boolean isSolved() {
        if (!this.isExitValid()) return false;
        return  this.getPrimaryPiece().nextForward(this) == this.goal ||
                this.getPrimaryPiece().nextBackward(this) == this.goal;
    }

    /* PIECES MOVEMENT */
    public ArrayList<Board> movePiece(char tag) {
        ArrayList<Board> result = new ArrayList<>();
        Board moveUp = null;
        Board moveDown = null;
        Board moveLeft = null;
        Board moveRight = null;

        Block forward = this.pieces.get(tag).nextForward(this);
        Block backward = this.pieces.get(tag).nextBackward(this);

        if (this.pieces.get(tag).canMove(forward, this)) {
            // move up
            if (this.pieces.get(tag).getDirection().equals(Direction.VERTICAL) ||
            this.pieces.get(tag).getDirection().equals(Direction.BOTH)) {
                moveUp = new Board(this);
                moveUp.pieces.get(tag).moveForward(moveUp);
                moveUp.moveTag = tag;
                moveUp.moveDirection = "UP";
            }
            // move left
            if (this.pieces.get(tag).getDirection().equals(Direction.HORIZONTAL) ||
            this.pieces.get(tag).getDirection().equals(Direction.BOTH)) {
                moveLeft = new Board(this);
                moveLeft.pieces.get(tag).moveForward(moveLeft);
                moveLeft.moveTag = tag;
                moveLeft.moveDirection = "LEFT";
            }
        }
        if (this.pieces.get(tag).canMove(backward, this)) {
            // move down
            if (this.pieces.get(tag).getDirection().equals(Direction.VERTICAL) ||
            this.pieces.get(tag).getDirection().equals(Direction.BOTH)) {
                moveDown = new Board(this);
                moveDown.pieces.get(tag).moveBackward(moveDown);
```

```
                moveDown.moveTag = tag;
                moveDown.moveDirection = "DOWN";
            }
            // move right
            if (this.pieces.get(tag).getDirection().equals(Direction.HORIZONTAL) ||
            this.pieces.get(tag).getDirection().equals(Direction.BOTH)) {
                moveRight = new Board(this);
                moveRight.pieces.get(tag).moveBackward(moveRight);
                moveRight.moveTag = tag;
                moveRight.moveDirection = "RIGHT";
            }
        }

        // check before push
        if (moveUp != null) result.add(moveUp);
        if (moveDown != null) result.add(moveDown);
        if (moveLeft != null) result.add(moveLeft);
        if (moveRight != null) result.add(moveRight);

        return result;
    }
    public ArrayList<Board> moveAllPieces() {
        ArrayList<Board> result = new ArrayList<>();
        for (Character key : this.pieces.keySet()) {
            result.addAll(this.movePiece(key));
        }
        return result;
    }

    /* PRINT BOARD */
    @Override
    public String toString() {
        return this.toStringColor(4, true);
    }
    public String toString(int indent) {
        String indentString = "";
        for (int i = 0; i < indent; i++) {
            indentString = indentString + " ";
        }
        String result = "";
        result = result + String.format("[%c] %s\n", this.moveTag, this.moveDirection);
        for (ArrayList<Block> mapRow : this.map) {
            result = result + indentString;
            for (Block blok : mapRow) {
                result = result + blok.getTag() + " ";
            }
            if (result.length() > 1) result = result.substring(0, result.length()-1);
            result = result + "\n";
        }
        if (result.length() > 1) result = result.substring(0, result.length()-1);
        return result;
    }

    /* PRINT BOARD WITH COLOR */
    public String toStringColor(int indent, Boolean highlight) {
        String indentString = "";
        for (int i = 0; i < indent; i++) {
            indentString = indentString + " ";
        }
        StringBuilder result = new StringBuilder();
        result.append(Ansi.ansi().fg(Ansi.Color.YELLOW).a("[").a(this.moveTag).a("]
").a(this.moveDirection).reset()).append("\n");

        ArrayList<Block> highligtBlocks = new ArrayList<>();
        if (this.moveTag != '\0') {
            Piece movePiece = this.pieces.get(this.moveTag);
            highligtBlocks.addAll(movePiece.getBlocks());
            if (this.moveDirection.equals("UP") || this.moveDirection.equals("LEFT")) {
                highligtBlocks.add(movePiece.nextBackward(this));
            }
            if (this.moveDirection.equals("DOWN") || this.moveDirection.equals("RIGHT")) {
                highligtBlocks.add(movePiece.nextForward(this));
            }
        }

        for (ArrayList<Block> mapRow : this.map) {
            result.append(indentString);
            for (Block blok : mapRow) {
                String blokTag = String.valueOf(blok.getTag());

                // color
                if (blok.getTag() == 'P') {
                    blokTag =
Ansi.ansi().fg(Ansi.Color.GREEN).a(Ansi.Attribute.INTENSITY_BOLD).a(blok.getTag()).reset().toString();
                } else if (blok.getTag() == 'K') {
                    blokTag = Ansi.ansi().fg(Ansi.Color.RED).a(blok.getTag()).reset().toString();
                } else if (blok.getTag() == this.moveTag) {
                    blokTag =
Ansi.ansi().fg(Ansi.Color.BLUE).a(Ansi.Attribute.INTENSITY_BOLD).a(blok.getTag()).reset().toString();
                }

                // highlight
                if (highlight && highligtBlocks.contains(blok)) {
                    blokTag = Ansi.ansi().fg(Ansi.Color.BLACK).bg(Ansi.Color.WHITE).a(blokTag).reset().toString();
                    String space = Ansi.ansi().bg(Ansi.Color.WHITE).a(" ").reset().toString();
                    result.append(blokTag).append(space);
```

```
                } else {
                    result.append(blokTag).append(" ");
                }

            }
            result.append("\n");
        }
        if (result.length() > 0) {
            result.deleteCharAt(result.length() - 1); // Hapus newline terakhir
        }
        return result.toString();
    }


    /* ADDITIONAL */
    public ArrayList<Piece> getBlockingPieces() {
        ArrayList<Piece> result = new ArrayList<>();
        Piece primary = this.getPrimaryPiece();
        if (primary.getDirection().equals(Direction.VERTICAL) || primary.getDirection().equals(Direction.BOTH)) {
            for (int i = primary.getHead().getRow(); i >= 0; i--) {
                Block b = this.map.get(i).get(goal.getCol());
                if (b.isPiece() && !b.isPrimary()) {
                    result.add(this.pieces.get(b.getTag()));
                }
            }
            for (int i = primary.getTail().getRow(); i < this.row; i++) {
                Block b = this.map.get(i).get(goal.getCol());
                if (b.isPiece() && !b.isPrimary()) {
                    result.add(this.pieces.get(b.getTag()));
                }
            }
        }
        if (primary.getDirection().equals(Direction.HORIZONTAL) || primary.getDirection().equals(Direction.BOTH)) {
            for (int j = primary.getHead().getCol(); j >= 0; j--) {
                Block b = this.map.get(goal.getRow()).get(j);
                if (b.isPiece() && !b.isPrimary()) {
                    result.add(this.pieces.get(b.getTag()));
                }
            }
            for (int j = primary.getTail().getCol(); j < this.col; j++) {
                Block b = this.map.get(goal.getRow()).get(j);
                if (b.isPiece() && !b.isPrimary()) {
                    result.add(this.pieces.get(b.getTag()));
                }
            }
        }
        return result;
    }

    /* COMPARATOR */
    public static Boolean isSame(Board b1, Board b2) {
        if (b1 == null || b2 == null) {
            return false;
        }
        if (
            b1.row != b2.row || b1.col != b2.col ||
            b1.goal.getRow() != b2.goal.getRow() ||
            b1.goal.getCol() != b2.goal.getCol()) {
            return false;
        }
        for (int i=0; i<b1.row; i++) {
            for (int j=0; j<b1.col; j++) {
                if (b1.map.get(i).get(j).getTag() != b2.map.get(i).get(j).getTag()) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

BoardTree.java

```
package model;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.TreeSet;

public class BoardTree implements Comparable<BoardTree> {

    /* ATTRIBUTE */
    private BoardTree root;
    private Board node;
    private ArrayList<BoardTree> branches;
    private long depth;
    private static HashMap<BoardTree, ArrayList<BoardTree>> leaves = new HashMap<>();
    private static HashMap<BoardTree, TreeSet<BoardTree>> trash = new HashMap<>();

    /* CONSTRUCTOR */
    public BoardTree(Board node) {
        this.root = null;
        this.node = node;
        this.depth = 0;
```

```
    this.branches = new ArrayList<>();
    BoardTree.leaves.put(this, new ArrayList<>());
    BoardTree.leaves.get(this).add(this);
    BoardTree.trash.put(this, new TreeSet<>());
}
public BoardTree(Board node, BoardTree root) {
    this.root = root;
    this.node = node;
    this.depth = root.getDepth() + 1;
    this.branches = new ArrayList<>();
    BoardTree.leaves.get(this.getFirstRoot()).add(this);
    BoardTree.trash.get(this.getFirstRoot()).add(this);
}

/* GETTER and SETTER */
public BoardTree getRoot() {
    return this.root;
}
public BoardTree getFirstRoot() {
    if (this.isRoot()) return this;
    return this.root.getFirstRoot();
}
public long getDepth() {
    return this.depth;
}
public Board getNode() {
    return this.node;
}
public ArrayList<BoardTree> getBranches() {
    return this.branches;
}
public static ArrayList<BoardTree> getLeaves(BoardTree root) {
    return BoardTree.leaves.get(root);
}
public void setNode(Board b) {
    this.node = b;
}
public long getMaxDepth() {
    long maxDepth = 0;
    for (BoardTree leaf : BoardTree.getLeaves(this)) {
        maxDepth = Math.max(maxDepth, leaf.getDepth());
    }
    return maxDepth;
}

/* NODE COUNT */
public long nodeCount() {
    long temp = 0;
    for (BoardTree branch : this.branches) {
        temp += branch.nodeCount();
    }
    if (this.isRoot()) {
        return 1 + this.branches.size() + temp;
    } else {
        return this.branches.size() + temp;
    }
}

/* BRANCHES */
public void addBranch(Board b) {
    if (b == null) return;
    int sizeBefore = BoardTree.trash.get(this.getFirstRoot()).size();
    BoardTree newBranch = new BoardTree(b, this);
    int sizeAfter = BoardTree.trash.get(this.getFirstRoot()).size();
    // System.err.println(BoardTree.trash);
    if (sizeAfter > sizeBefore) {
        this.branches.add(newBranch);
        // if (BoardTree.leaves.containsKey(this)) BoardTree.leaves.get(this).remove(this);
        if (BoardTree.leaves.containsKey(this.getFirstRoot())) BoardTree.leaves.get(this.getFirstRoot()).remove(this);
    }
}
public void addBranches(ArrayList<Board> data) {
    if (data == null) return;
    for (Board b : data) {
        this.addBranch(b);
    }
}

/* CONDITIONAL CHECK */
public Boolean isRoot() {
    return this.root == null;
}
public Boolean isLeaf() {
    return this.branches.size() == 0;
}

/* GENERATE BRANCHES */
public void generateBranches() {
    ArrayList<Board> nextMovement = this.node.moveAllPieces();
    this.addBranches(nextMovement);
}
public BoardTree getGoal() {
    // find goal in leaf
    BoardTree goal = null;
    for (BoardTree leaf : BoardTree.getLeaves(this)) {
        if (leaf.getNode().isSolved()) {
            if (goal == null) {
                goal = leaf;
            } else if (leaf.getDepth() <= goal.getDepth()) {
                goal = leaf;
            }
        }
    }
    return goal;
}

/* COMPARE */
@Override
public int compareTo(BoardTree b) {
    if (Board.isSame(this.getNode(), b.getNode())) {
```

```
            return 0;
        }
        return 1;
    }
}
```

Piece.java

```
package model;

import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.PriorityQueue;


public class Piece {

    /* CONST ENUMERATE */
    public enum Direction {
        VERTICAL,
        HORIZONTAL,
        BOTH
    }

    /* ATTRIBUTE */
    private Deque<Block> blocks;
    private char tag;
    private Direction direction;

    /* CONSTRUCTOR */
    public Piece(char tag) {
        this.tag = tag;
        this.blocks = new ArrayDeque<>();
    }
    public Piece(char tag, Deque<Block> blocks) {
        this.tag = tag;
        this.blocks = blocks;
        this.setDirection();
    }
    public Piece(Block block) {
        if (block.getTag() == '.' || block.getTag() == ' ' || block.getTag() == 'K') return;
        this.tag = block.getTag();
        this.blocks = new ArrayDeque<>();
        this.blocks.add(block);
        this.setDirection();
    }
    public Piece(Piece p) {
        this.tag = p.tag;
        this.blocks = new ArrayDeque<>();
        for (Block b : p.blocks) { // make new Deque, but shallow copy the blocks
            this.blocks.addLast(b);
        }
        this.direction = p.direction;
    }

    /* SET DIRECTION */
    public void setDirection() {
        if (blocks.size() == 1) {
            this.direction = Direction.BOTH;
        } else {
            if (blocks.getFirst().getRow() == blocks.getLast().getRow()) {
                this.direction = Direction.HORIZONTAL;
            } else {
                this.direction = Direction.VERTICAL;
            }
        }
    }
    /* SORT PIECE BLOCKS */
    public void sort() {
        PriorityQueue<Block> temp = new PriorityQueue<>(this.blocks);
        this.blocks = new ArrayDeque<>(temp);
    }

    /* GETTER and SETTER */
    public char getTag() {
        return this.tag;
    }
    public void addBlock(Block b) {
        this.blocks.add(b);
        this.setDirection();
        this.sort();
    }

    /* ADDITONAL GETTER */
    public Block getHead() {
        return this.blocks.getFirst();
    }
    public Block getTail() {
        return this.blocks.getLast();
    }
    public Direction getDirection() {
```

```
        return this.direction;
    }
    public ArrayList<Block> getBlocks() {
        ArrayList<Block> result = new ArrayList<>();
        Deque<Block> temp = new ArrayDeque<>();
        while (!this.blocks.isEmpty()) {
            Block b = this.blocks.poll();
            result.addLast(b);
            temp.addLast(b);
        }
        this.blocks = temp;
        return result;
    }

    /* BLOCK ACCESS */
        public Block nextForward(Board b) {
        try {
            if (this.direction.equals(Direction.VERTICAL) || this.direction.equals(Direction.BOTH)) {
                Block head = this.getHead();
                return b.getMap().get(head.getRow()-1).get(head.getCol());
            } else {
                Block head = this.getHead();
                return b.getMap().get(head.getRow()).get(head.getCol()-1);
            }
        } catch (Exception e) {
            return null;
        }
    }
    public Block nextBackward(Board b) {
        try {
            if (this.direction.equals(Direction.VERTICAL) || this.direction.equals(Direction.BOTH)) {
                Block tail = this.getTail();
                return b.getMap().get(tail.getRow()+1).get(tail.getCol());
            } else {
                Block tail = this.getTail();
                return b.getMap().get(tail.getRow()).get(tail.getCol()+1);
            }
        } catch (Exception e) {
            return null;
        }
    }

    /* MOVEMENT */
    public Boolean canMove(Block b, Board B) {
        // validity check
        if (b==null || !b.isValid() || this.blocks.isEmpty()) return false;

        Boolean forward     = (b == this.nextForward(B) && b.isEmpty());
        Boolean backward    = (b == this.nextBackward(B) && b.isEmpty());
        return (forward || backward);
    }
    public Piece moveForward(Board B) {
        if (!this.canMove(this.nextForward(B), B)) {
            return this;
        }
        if (this.blocks.isEmpty()) {
            return this;
        } else {
            Deque<Block> temp = new ArrayDeque<>();
            this.nextForward(B).setTag(this.tag);
            temp.addLast(this.nextForward(B));
            this.getHead().setTag('.');
            this.blocks.removeFirst();
            temp.addAll(this.moveForward(B).blocks);
            this.blocks = temp;
            this.sort();
            return this;
        }
    }
    public Piece moveBackward(Board B) {
        if (!this.canMove(this.nextBackward(B), B)) {
            return this;
        }
        if (this.blocks.isEmpty()) {
            return this;
        } else {
            Deque<Block> temp = new ArrayDeque<>();
            this.nextBackward(B).setTag(this.tag);
            temp.addFirst(this.nextBackward(B));
            this.getTail().setTag('.');
            this.blocks.removeLast();
            for (Block blok : this.moveBackward(B).blocks) {
                temp.addFirst(blok);
            }
            this.blocks = temp;
            this.sort();
            return this;
        }

    }

    /* PRINT PIECE */
    @Override
    public String toString() {
        String result = String.format("[%s] %c : ", this.direction, this.tag);
        for (Block blok : this.blocks) {
```

```
            result = result + " " + blok.toString();
        }
        return result;
    }

}
```

d. Solver/Algorithm

Algorithm.java

```
package solver.algorithm;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;
import java.util.TreeSet;

import model.Board;
import model.BoardTree;
import utils.Home;

public class Algorithm {

    /* ATTRIBUTE */
    protected BoardTree tree;
    protected PriorityQueue<BoardTree> queue;
    protected ArrayList<Board> solution;
    protected ArrayList<BoardTree> trash;
    protected Comparator<BoardTree> comparator;
    protected Comparator<BoardTree> setComparator;

    /* CONSTRUCTOR */
    public Algorithm(Board b) {
        this.tree = new BoardTree(b);
        this.queue = new PriorityQueue<>();
        this.solution = null;
        this.trash = new ArrayList<>();
    }
    public Algorithm(Board b, Comparator<BoardTree> comparator) {
        this.tree = new BoardTree(b);
        this.queue = new PriorityQueue<>(comparator);
        this.solution = null;
        this.trash = new ArrayList<>();
        this.comparator = comparator;
    }

    /* GETTER */
    public BoardTree getTree() {
        return this.tree;
    }
    public ArrayList<Board> getSolution() {
        return this.solution;
    }

    /* SOLVER */
    public ArrayList<Board> solve(Boolean debug) {
        this.solver(System.currentTimeMillis(), debug);

        // find goal
        BoardTree goal = this.tree.getGoal();
        if (goal == null) return null;

        // tracing to root
        ArrayList<Board> result = new ArrayList<>();
        BoardTree node = goal;
        while (node != null && !node.isRoot()) {
            result.addFirst(node.getNode());
            node = node.getRoot();
        }
        result.addFirst(this.tree.getNode());
        this.solution = result;
        return result;
    }
    public Boolean solver(long startDuration, Boolean debug) {

        if (this.tree == null) {
            return false;
        }
        if (this.tree.getNode().isSolved()) {
            return true;
        }
        BoardTree b = this.tree;
        this.queue.offer(b);
        while (!this.queue.isEmpty()) {
            b = this.queue.poll();

            // DEBUG
            if (debug) {
```

```
                printSolverState(b, this, startDuration);
            }

            if (b.getNode().isSolved()) {
                break;
            }
            b.generateBranches();

            // filter
            Set<BoardTree> temp1;
            if (this.setComparator == null) {
                temp1 = new TreeSet<>(this.queue);
            } else {
                temp1 = new TreeSet<>(this.setComparator);
                temp1.addAll(this.queue);
            }
            Set<BoardTree> temp2 = new TreeSet<>(this.trash);
            Set<BoardTree> temp3 = new TreeSet<>(b.getBranches());

            if (this.comparator == null) {
                this.queue = new PriorityQueue<>(temp1);
            } else {
                this.queue = new PriorityQueue<>(this.comparator); this.queue.addAll(temp1);
            }
            this.trash = new ArrayList<>(temp2);
            List<BoardTree> branches = new ArrayList<>(temp3);

            // System.out.println("SEBELUM DIHAPUS: " + branches.size());
            BoardTree[] nodeRemove = new BoardTree[branches.size()]; int j = 0;
            for (int i = 0; i<branches.size(); i++) {
                BoardTree node = branches.get(i);
                for (BoardTree checkNode : this.trash) {
                    if (Board.isSame(node.getNode(), checkNode.getNode())) {
                        // System.out.println("IS SAME DIPANGGIL DI SISNI  : " + node);
                        nodeRemove[j] = node;
                        j++;
                        break;
                    }
                }
            }
            for (int i = 0; i < j; i++) {
                BoardTree node = nodeRemove[i];
                this.trash.add(node);
                branches.remove(node);
                // System.out.println("REMOVE SUCCESS              : " + node);
            }
            // System.out.println("SETELAH DIHAPUS: " + branches.size());
            this.trash.add(b);
            this.queue.addAll(branches);
        }
        return b.getNode().isSolved();
    }

    /* GET: g(n) */
    public static Long g(BoardTree b) {
        return b.getDepth();
    }

    /* PRINT RESULT */
    @Override
    public String toString() {
        return this.toString(4, true);
    }
    public String toString(int indent, Boolean color) {

        StringBuilder result = new StringBuilder();
        String indentString = "";
        for (int i = 0; i < indent; i++) {
            indentString = indentString + " ";
        }

        if (this.solution == null) {
            result.append(indentString).append("DON'T HAVE SOLUTION");
        } else if (this.solution.size() == 0) {
            result.append(indentString).append("DON'T HAVE SOLUTION");
        } else if (this.solution.size() == 1 && !this.solution.get(0).isSolved()) {
            result.append(indentString).append("DON'T HAVE SOLUTION");
        } else {
            for (Board step : this.solution) {
                result.append(String.format("\n[*] Gerakan %d\n", this.solution.indexOf(step)));
                if (color) {
                    result.append(step);
                } else {
                    result.append(step.toString(indent));
                }
                result.append("\n");
            }
            if (result.length() > 0) {
                result.deleteCharAt(result.length() - 1); // Hapus newline terakhir
            }
        }
        return result.toString();
    }

    /* STATIC METHOD */
    public void printSolverState(BoardTree checkedNode, Algorithm solver, long startDuration) {
```

```
        try {
            Home.Loading(checkedNode, solver, startDuration);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

ASTAR.java

```
package solver.algorithm;

import model.Board;
import model.BoardTree;
import solver.heuristic.Heuristic;

public class ASTAR extends Algorithm {

    /* CONSTRUCTOR */
    public ASTAR(Board b, Heuristic h) {
        super(b, new ASTARcomparator(h));
        this.setComparator = new ASTARSetComparator(h);
    }

}

class ASTARcomparator implements IRoutePlanning {

    /* ATTRIBUTE */
    private Heuristic h;

    /* CONSCTRUCTOR */
    public ASTARcomparator(Heuristic h) {
        this.h = h;
    }

    /* f(n) */
    public Double f(BoardTree b) {
        return Algorithm.g(b) + this.h.calculate(b.getNode());
    }

    /* COMPARATOR ALGORITHM: f(n) = g(n) + h(n) */
    public int compare(BoardTree b1, BoardTree b2) {
        Double f1 = this.f(b1);
        Double f2 = this.f(b2);
        if (f1 - f2 == 0) return 0;
        return (int) ((f1 - f2)/Math.abs(f1 - f2));
    }

}

class ASTARSetComparator extends ASTARcomparator {

    /* CONSTRUCTOR */
    public ASTARSetComparator(Heuristic h) {
        super(h);
    }

    /* ADDITIONAL COMPARATOR */
    @Override
    public int compare(BoardTree b1, BoardTree b2) {
        if (this.f(b1) < this.f(b2)) {
            return -1;
        }
        if (Board.isSame(b1.getNode(), b2.getNode())) {
            return 0;
        }
        return 1;
    }
}
```

BNB.java

```
package solver.algorithm;

import model.Board;
import model.BoardTree;
import solver.heuristic.Heuristic;

public class BNB extends Algorithm {

    /* ATTRIBUTE */
    private Double bestCost;

    /* CONSTRUCTOR */
    public BNB(Board b) {
        super(b, new BNBcomparator(null));
```

```
        this.bestCost = null;
        this.setComparator = new BNBSetComparator(null, this);
    }
    public BNB(Board b, Heuristic h) {
        super(b, new BNBcomparator(h));
        this.bestCost = null;
        this.setComparator = new BNBSetComparator(h, this);
    }

    /* GETTER */
    public Double getBestCost() {
        return this.bestCost;
    }
    public void setBestCost(Double val) {
        this.bestCost = val;
    }

}

class BNBcomparator implements IRoutePlanning {

    /* ATTRIBUTE */
    private Heuristic h;

    /* CONSTRUCTOR */
    public BNBcomparator(Heuristic h) {
        this.h = h;
    }

    /* COST ALGORITHM: f(n) = g(n) + h(n) */
    public Double COST(BoardTree b) {
        Long g;
        Double h;
        if (this.h == null) {
            h = 0.0;
        } else {
            h = this.h.calculate(b.getNode());
        }
        g = Algorithm.g(b);
        return g + h;
    }

    /* COMPARATOR ALGORITHM: f(n) = g(n) + h(n) */
    public int compare(BoardTree b1, BoardTree b2) {
        Double f1 = this.COST(b1);
        Double f2 = this.COST(b2);
        if (f1 - f2 == 0) return 0;
        return (int) ((f1 - f2)/Math.abs(f1 - f2));
    }

}

class BNBSetComparator extends BNBcomparator {

    /* ATTRIBUTE */
    private BNB ref;

    /* CONSTRUCTOR */
    public BNBSetComparator(Heuristic h, BNB ref) {
        super(h);
        this.ref = ref;
    }

    /* ADDITIONAL COMPARATOR */
    @Override
    public int compare(BoardTree b1, BoardTree b2) {
        if (Board.isSame(b1.getNode(), b2.getNode())) {
            return 0;
        }
        if (this.ref.getBestCost() == null) {
            return -1;
        }
        if (this.COST(b1) <= this.ref.getBestCost()) {
            this.ref.setBestCost(this.COST(b1));
            return -1;
        }
        if (this.COST(b1) < this.COST(b2)) {
            return -1;
        }
        return 0;
    }
}
```

GBFS.java

```
package solver.algorithm;

import model.Board;
import model.BoardTree;
```

```
import solver.heuristic.Heuristic;

public class GBFS extends Algorithm {

    /* CONSTRUCTOR */
    public GBFS(Board b, Heuristic h) {
        super(b, new GBFScomparator(h));
    }

}

class GBFScomparator implements IRoutePlanning {

    /* ATTRIBUTE */
    private Heuristic h;

    /* CONSTRUCTOR */
    public GBFScomparator(Heuristic h) {
        this.h = h;
    }

    /* COMAPATOR ALGORITHM: f(n) = h(n) */
    public int compare(BoardTree b1, BoardTree b2) {
        Double f1 = this.h.calculate(b1.getNode());
        Double f2 = this.h.calculate(b2.getNode());
        if (f1 - f2 == 0) return 0;
        return (int) ((f1 - f2)/Math.abs(f1 - f2));
    }

}
```

IRoutePlanning.java

```
package solver.algorithm;

import java.util.Comparator;

import model.BoardTree;

public interface IRoutePlanning extends Comparator<BoardTree> {

    /* COMPARATOR ALGORITHM */
    public int compare(BoardTree b1, BoardTree b2);

}
```

UCS.java

```
package solver.algorithm;

import java.util.PriorityQueue;

import model.Board;
import model.BoardTree;

public class UCS extends Algorithm {

    /* CONSTRUCTOR */
    public UCS(Board b) {
        super(b);
        this.queue = new PriorityQueue<>(new UCScomparator());
    }

}

class UCScomparator implements IRoutePlanning {

    /* COMPARATOR ALGORITHM: f(n) = g(n) */
    public int compare(BoardTree b1, BoardTree b2) {
        Long f1 = Algorithm.g(b1);
        Long f2 = Algorithm.g(b2);
        if (f1 - f2 == 0) return 0;
        return (int) ((f1 - f2)/Math.abs(f1 - f2));
    }

}
```

e. Solver/Heuristic

Heuristic.java

```
package solver.heuristic;

import model.Board;

public abstract class Heuristic {

    /* METHOD */
    public abstract Double calculate(Board b);

}
```

DistanceToExit.java

```
package solver.heuristic;

import model.Block;
import model.Board;
import model.Piece;

public class DistanceToExit extends Heuristic {

    /* Menghitung jarak dari Primary ke Exit */

    /* METHOD */
    public Double calculate(Board b) {
        // find data
        Piece p = b.getPrimaryPiece();
        Block goal = b.getGoal();

        // calculate manhattan distance
        int distance = Math.min(
            goal.ManhattanDistance(p.getHead()),
            goal.ManhattanDistance(p.getTail()));

        return Double.valueOf(distance);
    }

}
```

BlockingPieceCount.java

```
package solver.heuristic;

import java.util.ArrayList;

import model.Board;
import model.Piece;

public class BlockingPieceCount extends Heuristic {

    /* Menghitung jumlah piece yang menghalangi jalan ke Exit */

    /* METHOD */
    public Double calculate(Board b) {
        // find data
        ArrayList<Piece> blockingPiece = b.getBlockingPieces();
        return Double.valueOf(blockingPiece.size());
    }

}
```

SolidnessBlockingPiece.java

```
package solver.heuristic;

import java.util.ArrayList;

import model.Board;
import model.Piece;

public class SolidnessBlockingPiece extends Heuristic {

    /* Menghutung jumlah piece yang menghalangi jalan yang tidak dapat digerakkan */

    /* METHOD */
    public Double calculate(Board b) {
        // find data
        ArrayList<Piece> blockingPieces = b.getBlockingPieces();
```

```
        int count = 0;
        for (Piece p : blockingPieces) {
            if (p.canMove(p.nextForward(b), b)) continue;
            if (p.canMove(p.nextBackward(b), b)) continue;
            count++;
        }

        return Double.valueOf(count);
    }

}
```

MovementToMakeWay.java

```
package solver.heuristic;

import java.util.ArrayList;

import model.Block;
import model.Board;
import model.Piece;
import model.Piece.Direction;

public class MovementToMakeWay extends Heuristic {

    /* METHOD */
    public Double calculate(Board b) {
        // find data
        ArrayList<Piece> blockingPieces = b.getBlockingPieces();
        int count = 0;
        for (Piece p : blockingPieces) {
            Block intercept = intercept(b.getPrimaryPiece(), p);
            if (
                p.nextForward(b) != null && p.nextForward(b).isValid() &&
                p.nextBackward(b) != null && p.nextBackward(b).isValid()
            ) {
                count += Math.min(intercept.ManhattanDistance(p.getHead()), intercept.ManhattanDistance(p.getTail()));
            } else if (p.nextForward(b) != null && p.nextForward(b).isValid()) {
                count += intercept.ManhattanDistance(p.getTail());
            } else if (p.nextBackward(b) != null && p.nextBackward(b).isValid()) {
                count += intercept.ManhattanDistance(p.getHead());
            } else {
                count += 100000;
            }
        }

        return Double.valueOf(count);
    }

    /* HELPER */
    private static Block intercept(Piece primary, Piece other) {
        if (primary.getDirection().equals(other.getDirection())) {
            return null;
        }
        if (primary.getDirection().equals(Direction.HORIZONTAL)) {
            return other.getBlocks().get(primary.getHead().getRow() - other.getHead().getRow());
        } else { // VERTICAL
            return other.getBlocks().get(primary.getHead().getCol() - other.getHead().getCol());
        }
    }

}
```

CombinedHeuristic.java

```
package solver.heuristic;

import java.util.ArrayList;
import java.util.Collections;

import model.Board;

public class CombinedHeuristic extends Heuristic{

    /* CONSTANT ENUM */
    enum HEURISTIC_TYPE {
        MAX,
        MIN,
        SUM,
        AVG
    }

    /* ATTRIBUTE */
    private ArrayList<Heuristic> heuristic;
    private HEURISTIC_TYPE mode;

    /* CONSTRUCTOR */
```

```
    public CombinedHeuristic(Heuristic[] h, String mode) {
        this.heuristic = new ArrayList<>();
        for (int i=0; i<h.length; i++) {
            this.heuristic.add(h[i]);
        }
        switch (mode) {
            case "MAX" -> {this.mode = HEURISTIC_TYPE.MAX; break;}
            case "MIN" -> {this.mode = HEURISTIC_TYPE.MIN; break;}
            case "SUM" -> {this.mode = HEURISTIC_TYPE.SUM; break;}
            case "AVG" -> {this.mode = HEURISTIC_TYPE.AVG; break;}
        }
    }

    /* METHOD */
    public Double calculate(Board b) {
        ArrayList<Double> allResults = new ArrayList<>();
        for (Heuristic h : this.heuristic) {
            allResults.add(h.calculate(b));
        }
        Double finalResult = 0.0;
        switch (this.mode) {
            case HEURISTIC_TYPE.MAX -> {finalResult = Collections.max(allResults);}
            case HEURISTIC_TYPE.MIN -> {finalResult = Collections.min(allResults);}
            case HEURISTIC_TYPE.SUM -> {
                Double sum = 0.0;
                for (Double result : allResults) {sum += result;}
                finalResult = sum;
            }
            case HEURISTIC_TYPE.AVG -> {
                Double sum = 0.0;
                for (Double result : allResults) {sum += result;}
                finalResult = sum / allResults.size();
            }
        }
        return finalResult;
    }

}
```

f. Utils

DualOutpur.java

```
package utils;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;

public class DualOutput {

    private static PrintStream originalOut = System.out;
    private static PrintStream fileOut = null;
    // private static PrintStream dualOut = null;
    private static boolean isActive = false;

    public static void activate(String filePath) {
        if (!isActive) {
            try {
                FileOutputStream fileOutputStream = new FileOutputStream(filePath, false);
                fileOut = new PrintStream(fileOutputStream);
                // dualOut = new PrintStream(new DualOutputStream(originalOut, fileOut));
                // System.setOut(dualOut);
                System.setOut(fileOut);
                isActive = true;
                // System.out.println("[DualOutput] Dual output activated. Logging to console and: " + filePath);
            } catch (FileNotFoundException e) {
                System.err.println("[DualOutput] Error activating dual output: " + e.getMessage());
            }
        } else {
            System.out.println("[DualOutput] Dual output is already active.");
        }
    }

    public static void deactivate() {
        if (isActive) {
            System.setOut(originalOut);
            if (fileOut != null) {
                fileOut.close();
            }
            // dualOut = null;
            fileOut = null;
            isActive = false;
            // System.out.println("[DualOutput] Dual output deactivated.");
        } else {
            System.out.println("[DualOutput] Dual output is not active.");
        }
    }

    // private static class DualOutputStream extends java.io.OutputStream {
    //     private java.io.OutputStream one;
```

```
    //     private java.io.OutputStream two;

    //     public DualOutputStream(java.io.OutputStream one, java.io.OutputStream two) {
    //         this.one = one;
    //         this.two = two;
    //     }

    //     @Override
    //     public void write(int b) throws java.io.IOException {
    //         one.write(b);
    //         two.write(b);
    //     }

    //     @Override
    //     public void flush() throws java.io.IOException {
    //         one.flush();
    //         two.flush();
    //     }

    //     @Override
    //     public void close() throws java.io.IOException {
    //         two.close();
    //     }
    // }

}
```

FileHandler.java

```
package utils;

import model.Board;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class FileHandler {

    /* STATIC METHOD */
    public static Board loadFile(String filepath) {
        Path path = Paths.get(filepath);
        try {
            List<String> tempList = Files.readAllLines(path);
            ArrayList<String> fileRows = new ArrayList<>(tempList.subList(2, tempList.size()));
            return new Board(fileRows);
        } catch (IOException e) {
            System.err.println("[X] ERROR ketika membaca file: " + e.getMessage());
            return null;
        }
    }

}
```

Home.java

```
package utils;

import model.Board;
import model.BoardTree;
import solver.algorithm.Algorithm;

/* ADDITIONAL STATIC METHOD */
public class Home {

    /* PUBLIC METHOD */
    public static void Header() {
        Terminal.clearScreen();
        System.out.println();
        System.out.println("[============================================================]");
        System.out.println("[==============[  RUSH HOUR SOLVER MASTER  ]==============]");
        System.out.println("[============================================================]");
        System.out.println("[     Selamat datang di Solver permainan Rush Hour no.1     ]");
        System.out.println("[  di dunia. Terdapat pilihan algorithma yang keren dengan  ]");
        System.out.println("[  beragam pilihan heuristik yang menarik. Selamat mencoba! ]");
        System.out.println("[============================================================]");
        System.out.println();
    }
    public static void Footer() throws InterruptedException {
        Home.noSleepFooter();
        Thread.sleep(2000);
    }
    public static void Loading(BoardTree checkedNode, Algorithm solver, long startDuration) throws InterruptedException {
        Home.Header();
        System.out.println(Home.getMemoryInfo());
```

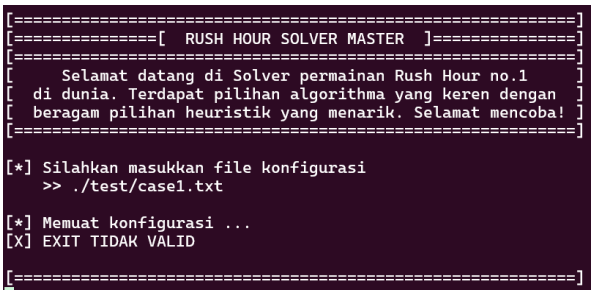
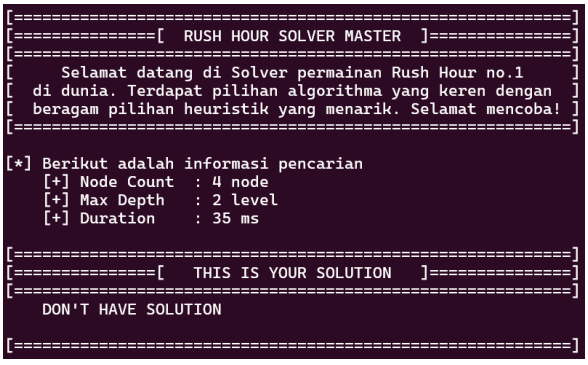
```
        Home.printSearchInfo(checkedNode, solver, startDuration);
        Home.noSleepFooter();
    }
    public static void Solution(Algorithm solver, long duration, Boolean color) throws InterruptedException {
        Home.Header();
        System.out.println(    "[*] Berikut adalah informasi pencarian");
        System.out.printf("    [+] Node Count  : %d node\n", solver.getTree().nodeCount());
        System.out.printf("    [+] Max Depth   : %d level\n", solver.getTree().getMaxDepth());
        System.out.printf("    [+] Duration    : %d ms\n", duration);
        System.out.println();
        System.out.println("[============================================================]");
        System.out.println("[==============[   THIS IS YOUR SOLUTION   ]==============]");
        System.out.println("[============================================================]");
        System.out.println(solver.toString(4, color));
        if (color != null && color) {
            Home.Footer();
        } else {
            Home.noSleepFooter();
        }
    }
    public static void AnimateSolution(Algorithm solver, long duration) {
        try {
            while (true) {
                for (Board b : solver.getSolution()) {
                    Home.Header();
                    System.out.println(    "[*] Berikut adalah informasi pencarian");
                    System.out.printf("    [+] Node Count  : %d node\n", solver.getTree().nodeCount());
                    System.out.printf("    [+] Max Depth   : %d level\n", solver.getTree().getMaxDepth());
                    System.out.printf("    [+] Duration    : %d ms\n", duration);
                    System.out.println();
                    if (b.isSolved()) {
                        System.out.printf("[*] Gerakan %d (SOLVED)\n", solver.getSolution().indexOf(b));
                        System.out.println(b.toStringColor(4, false));
                        Home.noSleepFooter();
                        Thread.sleep(2000);
                    } else {
                        System.out.printf("[*] Gerakan %d\n", solver.getSolution().indexOf(b));
                        System.out.println(b.toStringColor(4, false));
                        Home.noSleepFooter();
                        Thread.sleep(250);
                    }
                }
            }
        } catch (Exception e) {
            System.err.println("[*] Program selesai");
        } finally {
            Terminal.clearScreen();
        }
    }

    /* PRIVATE METHOD */
    private static void noSleepFooter() {
        System.out.println();
        System.out.println("[============================================================]");
    }

    private static void printSearchInfo(BoardTree checkedNode, Algorithm solver, long startDuration) {
        System.out.println("[*] Dynamic Tree Information");
        System.out.printf("    [+] Node Count    : %d node\n", solver.getTree().nodeCount());
        System.out.printf("    [+] Current Depth : %d level\n", checkedNode.getDepth());
        System.out.printf("    [+] Exec Time     : %d ms\n", System.currentTimeMillis() - startDuration);
        System.out.println();
        System.out.println("[*] Checked Node");
        System.out.println(checkedNode.getNode());
    }
    private static String getMemoryInfo() {
        Runtime runtime = Runtime.getRuntime();

        long totalMemory = runtime.totalMemory(); // Total memory allocated to the heap
        long freeMemory = runtime.freeMemory();   // Free memory available in the heap
        long maxMemory = runtime.maxMemory();    // Maximum memory the heap can grow to

        String info = "";

        info = info + String.format("[*] Heap Memory Information\n");
        info = info + String.format("    [+] Total Heap    : %s\n", formatSize(totalMemory));
        info = info + String.format("    [+] Free Heap     : %s\n", formatSize(freeMemory));
        info = info + String.format("    [+] Used Heap     : %s\n", formatSize(totalMemory - freeMemory));
        info = info + String.format("    [+] Max Heap      : %s\n", formatSize(maxMemory));

        return info;
    }
    private static String formatSize(long bytes) {
        if (bytes < 1024) return bytes + " B";
        int exp = (int) (Math.log(bytes) / Math.log(1024));
        char unit = "KMGTPE".charAt(exp - 1);
        return String.format("%.2f %sB", bytes / Math.pow(1024, exp), unit);
    }
}
```

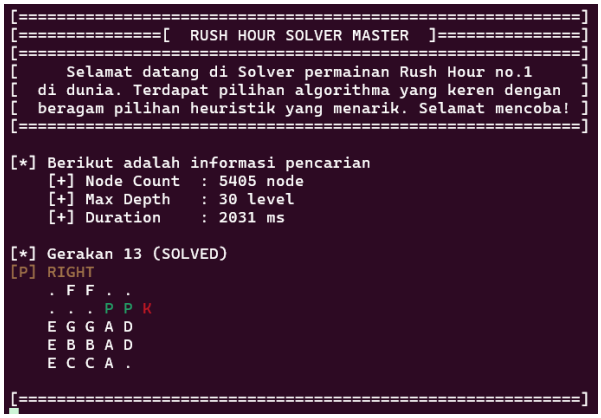
Terminal.java

```
package utils;

import java.io.IOException;

public class Terminal {

    /* STATIC METHOD */
    public static void clearScreen() {
        try {
            String os = System.getProperty("os.name").toLowerCase();

            // Membersihkan layar di Windows
            if (os.contains("win")) {
                new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();

            // Membersihkan layar di Unix-like (Linux, macOS)
            } else if (os.contains("nix") || os.contains("nux") || os.contains("mac")) {
                System.out.print("\033[H\033[2J");
                System.out.flush();

            // OS tidak dikenali
            } else {
                System.out.println("Sistem operasi tidak dikenali, tidak dapat membersihkan layar.");
            }

        } catch (IOException | InterruptedException e) {
            System.err.println("Gagal membersihkan layar: " + e.getMessage());
        }
    }
}
```

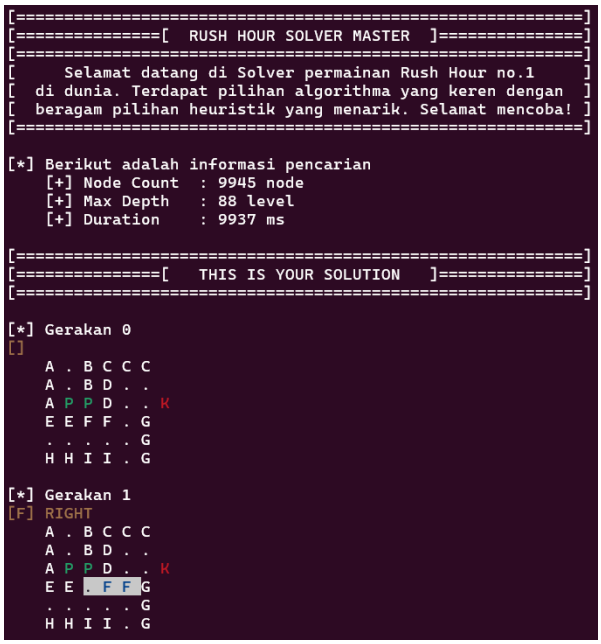
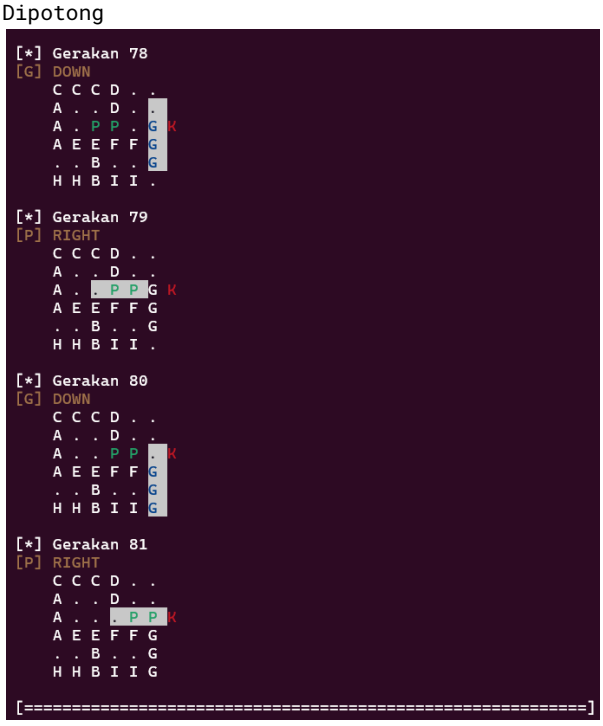
E. Pengujian

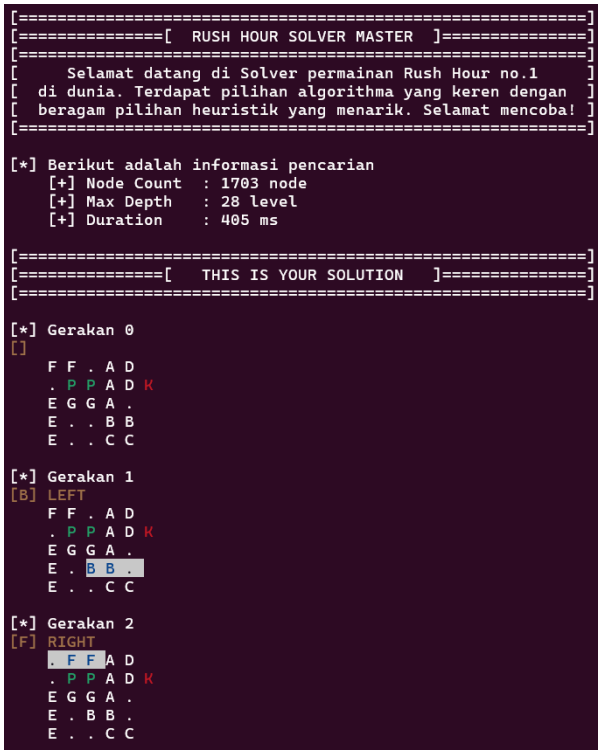
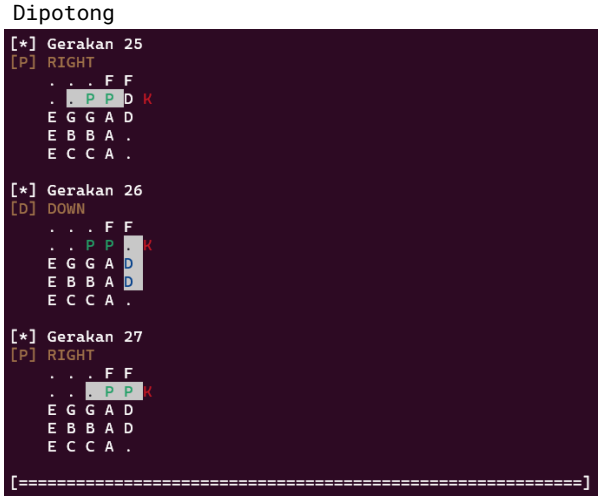
Dilakukan pengujian dengan rincian kasus uji dan hasil sebagai berikut

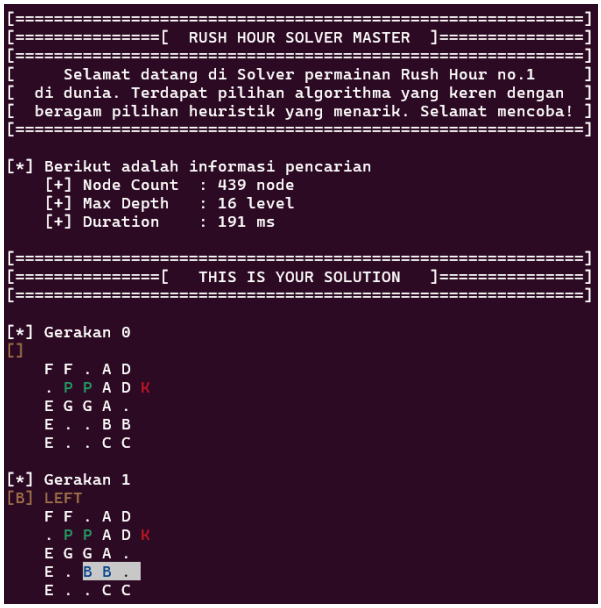
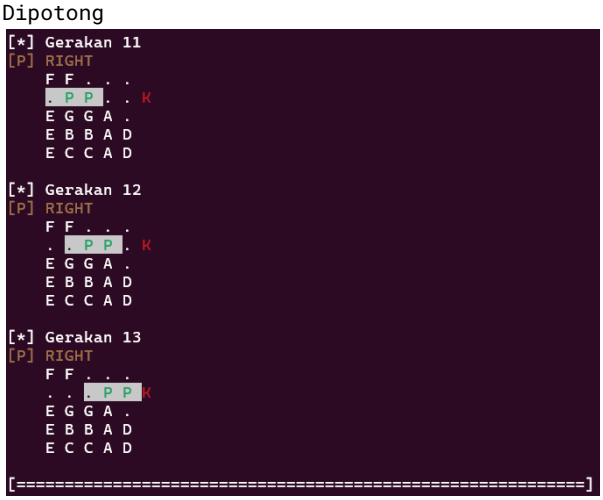
No	Indikator	Input	Hasil
1	UCS, Board tidak valid	5 4 2 K.... .... BAA. B..P ...P	
2	UCS, Tidak ada solusi	5 4 2 K .... B... BAAA B..P ...P	

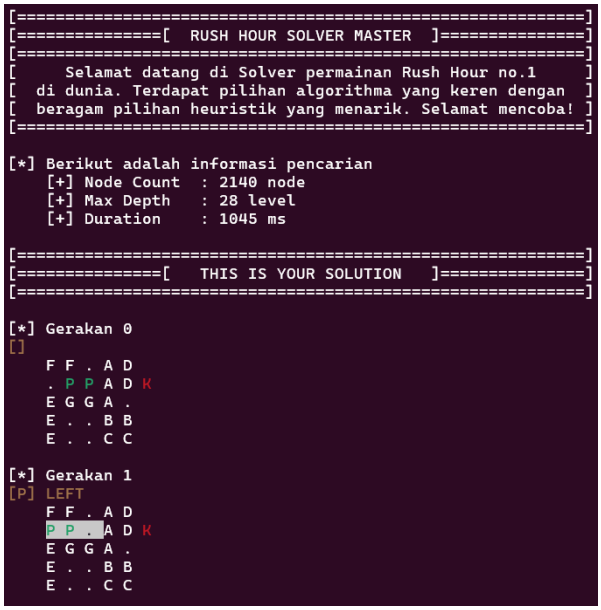
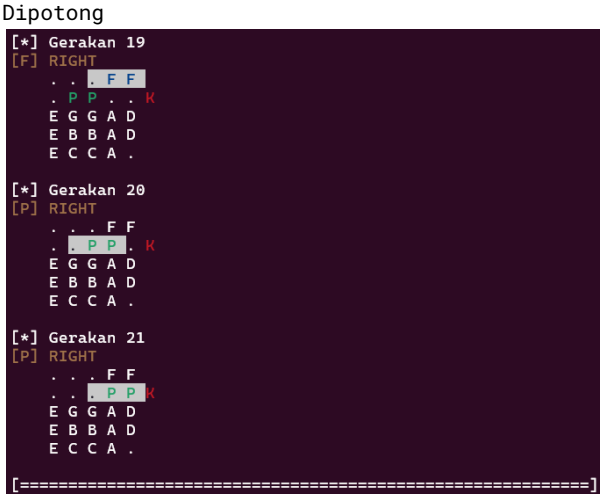
3	UCS, Board sederhana	5 5 7 FF.AD .PPADK EGGA. E..BB E..CC	 [RUSH HOUR SOLVER MASTER] Selamat datang di Solver permainan Rush Hour no.1 di dunia. Terdapat pilihan algorithma yang keren dengan beragam pilihan heuristik yang menarik. Selamat mencoba! [*] Berikut adalah informasi pencarian [+] Node Count : 5405 node [+] Max Depth : 30 level [+] Duration : 2031 ms [THIS IS YOUR SOLUTION] [*] Gerakan 0 [] F F . A D . P P A D K E G G A . E . . B B E . . C C [*] Gerakan 1 [P] LEFT F F . A D P P . A D K E G G A . E . . B B E . . C C [*] Gerakan 2 [C] LEFT F F . A D P P . A D K E G G A . E . . B B E . C C . [*] Gerakan 3 [B] LEFT F F . A D P P . A D K E G G A . E . B B . E . C C . [*] Gerakan 4 [B] LEFT F F . A D P P . A D K E G G A . E B B . . E . C C . [*] Gerakan 5 [C] LEFT

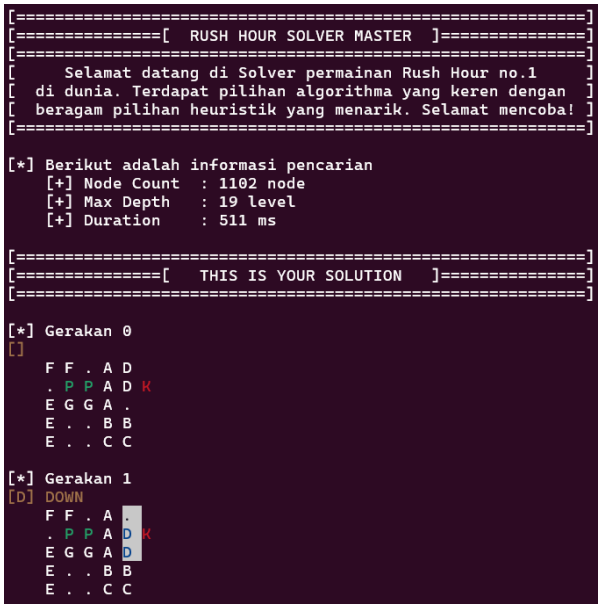
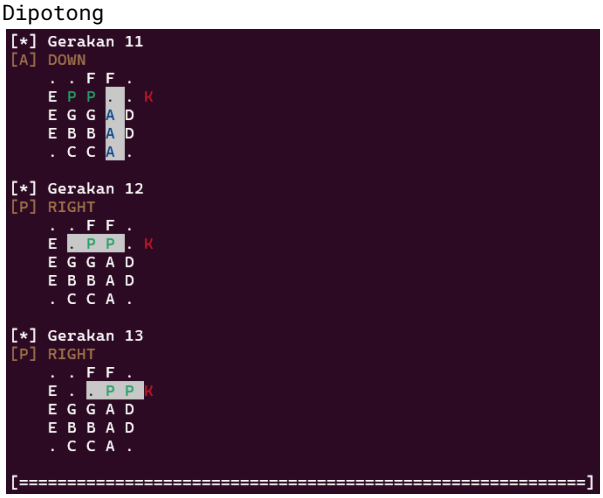
```
    F F . A D
    P P . A D K
    E G G A .
    E B B . .
    E C C . .

[*] Gerakan 6
[F] RIGHT
    . F F A D
    P P . A D K
    E G G A .
    E B B . .
    E C C . .

[*] Gerakan 7
[D] DOWN
    . F F A .
    P P . A D K
    E G G A D
    E B B . .
    E C C . .

[*] Gerakan 8
[A] DOWN
    . F F . .
    P P . A D K
    E G G A D
    E B B A .
    E C C . .

[*] Gerakan 9
[A] DOWN
    . F F . .
    P P . . D K
    E G G A D
    E B B A .
    E C C A .

[*] Gerakan 10
[D] DOWN
    . F F . .
    P P . . . K
    E G G A D
    E B B A D
    E C C A .

[*] Gerakan 11
[P] RIGHT
    . F F . .
    . P P . . K
    E G G A D
    E B B A D
    E C C A .

[*] Gerakan 12
[P] RIGHT
    . F F . .
    . . P P . K
    E G G A D
    E B B A D
    E C C A .

[*] Gerakan 13
[P] RIGHT
    . F F . .
    . . . P P K
    E G G A D
    E B B A D
    E C C A .

[==========================================================]
```

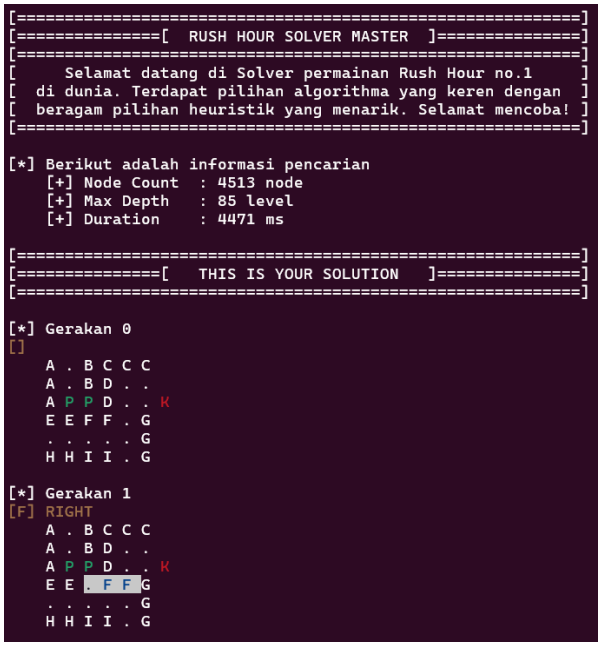
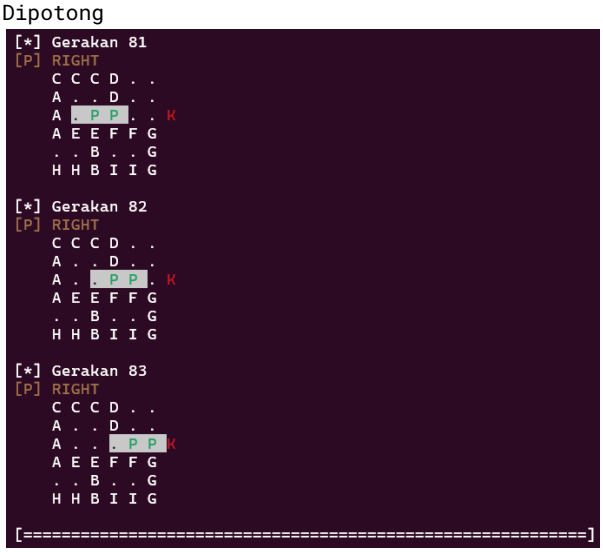
4	UCS, Board kompleks		 Dipotong 

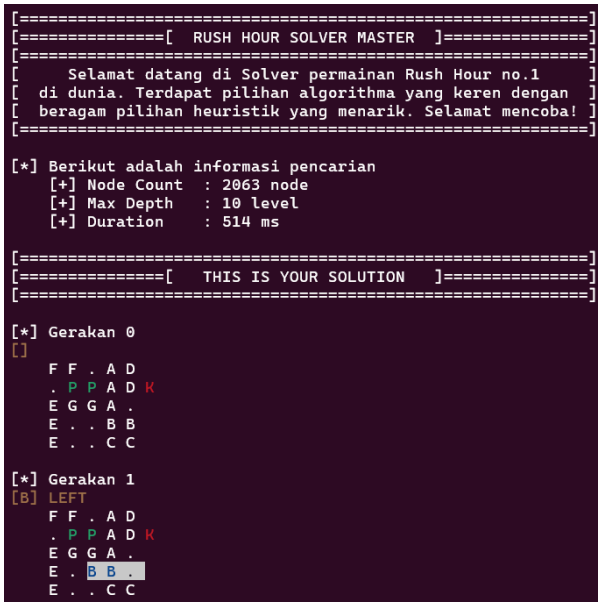
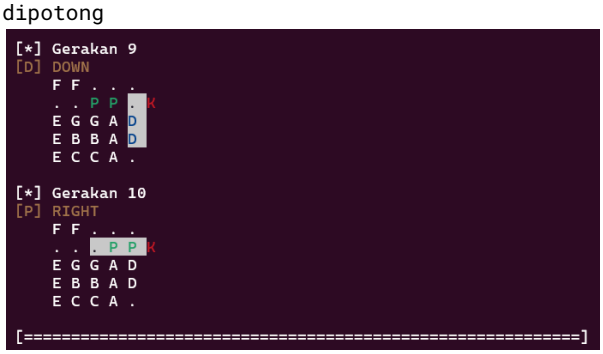
5	GBFS, Heuristik 1	Sama seperti case 3	 Dipotong 

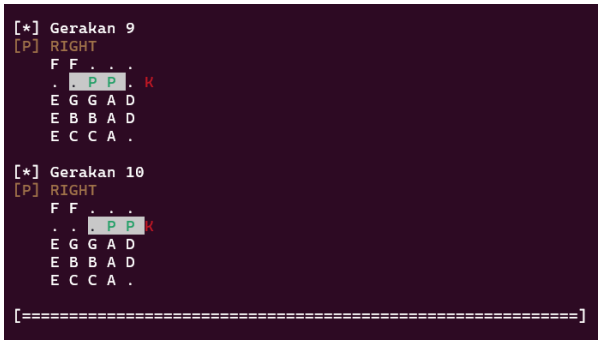
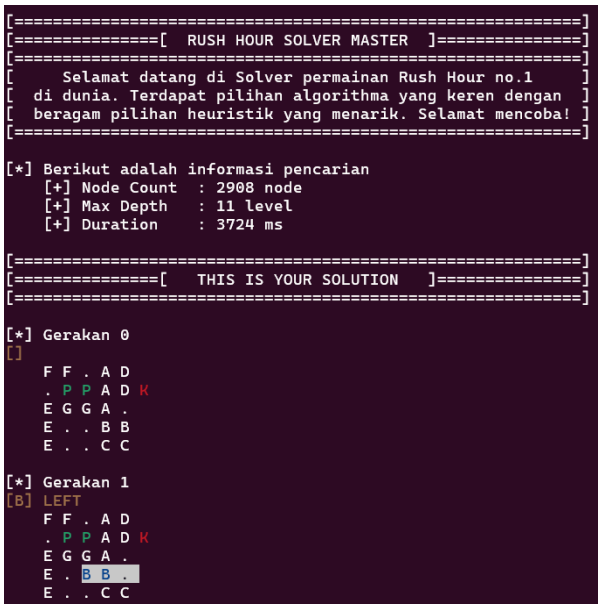
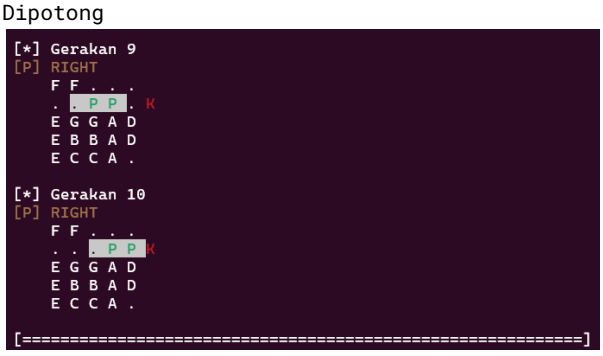
6	GBFS, Heuristik 2	Sama seperti case 3	 Dipotong 

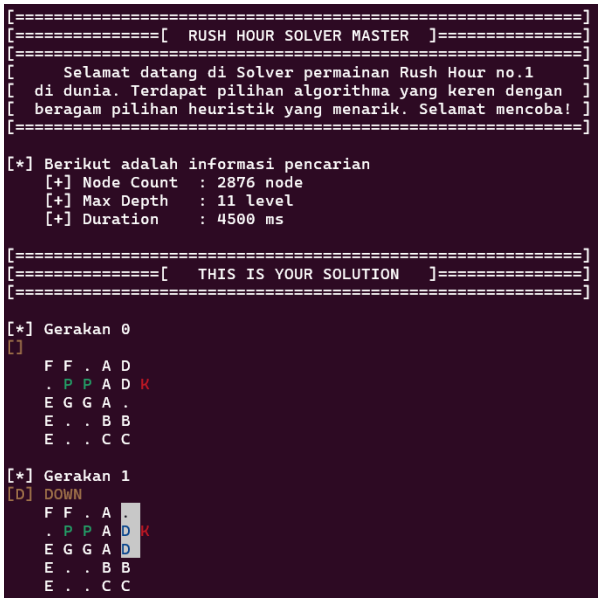
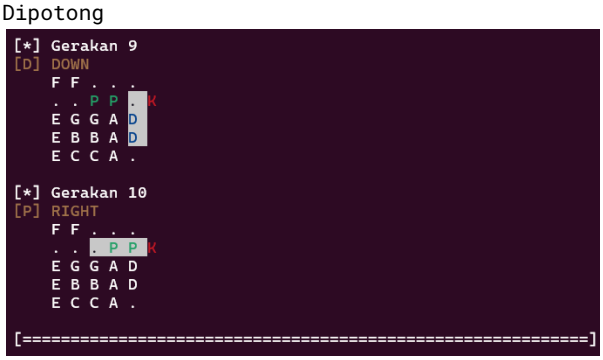
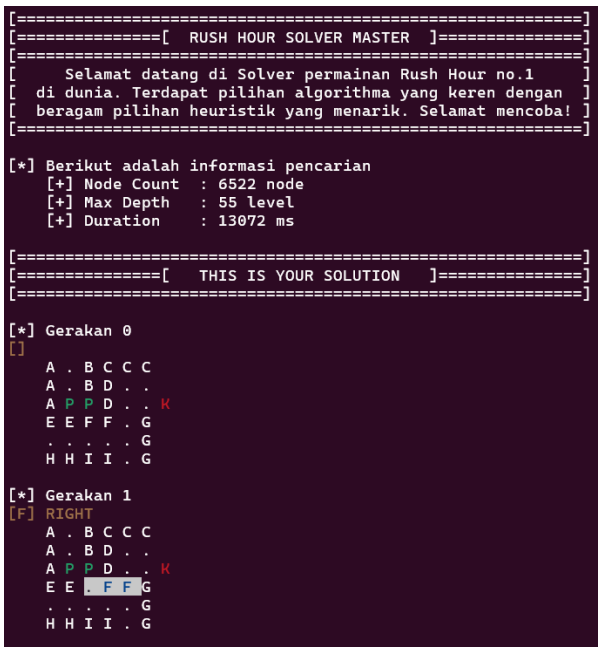
7	GBFS, Heuristik 3	Sama seperti case 3	 Dipotong 

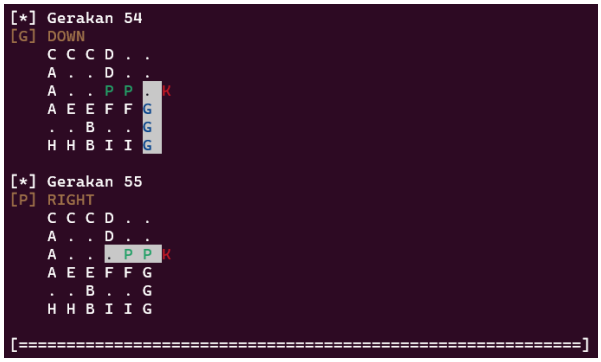
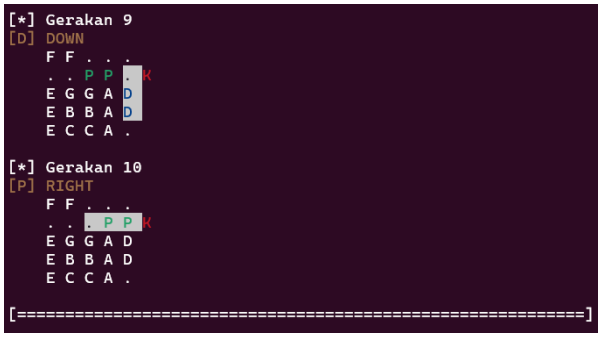
8	GBFS, Heuristik 4	Sama seperti case 3	 Dipotong 

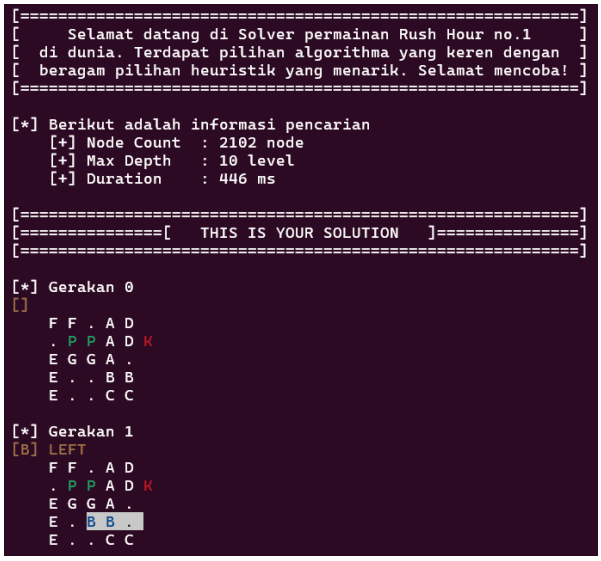
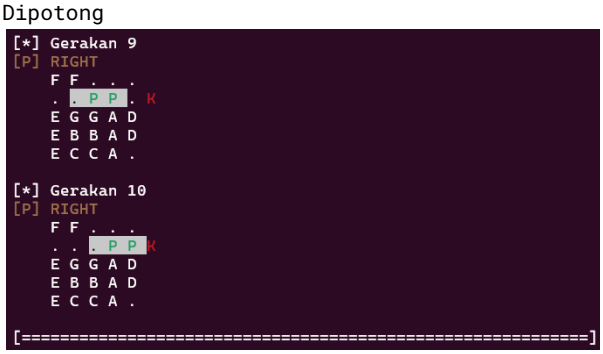
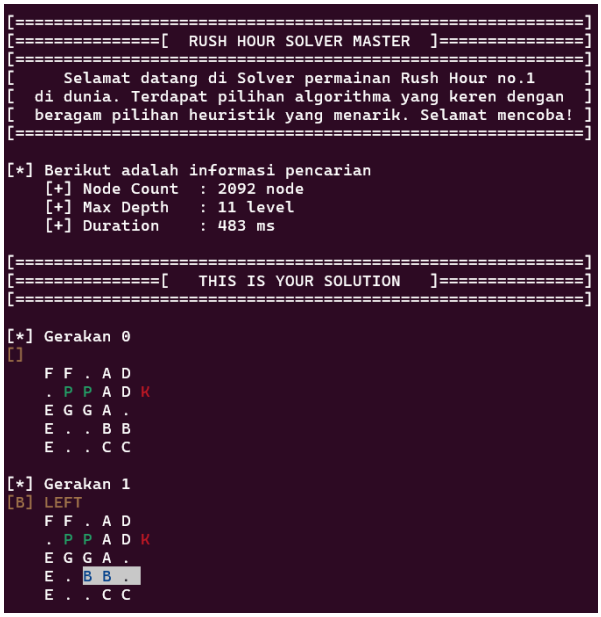
9	GBFS, MIN 1 2 3 4	Sama seperti case 4	 Dipotong 

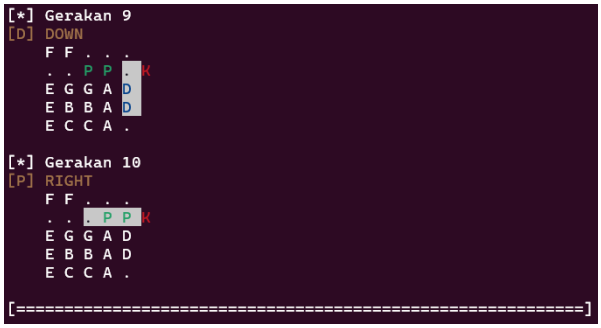
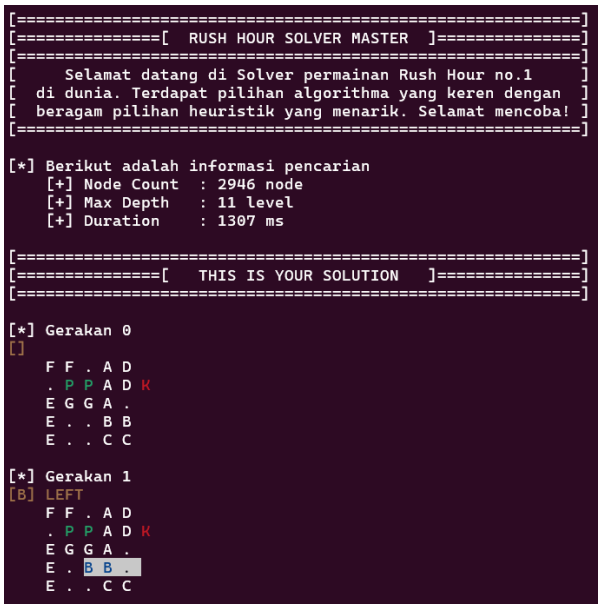
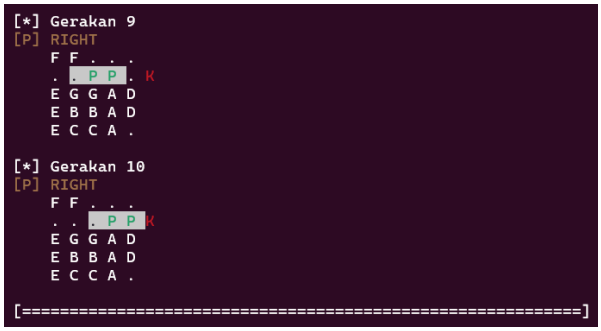
10	A*, Heuristik 1	Sama seperti case 3	 dipotong 
11	A*, Huristik 2	Sama seperti case 3	 Dipotong

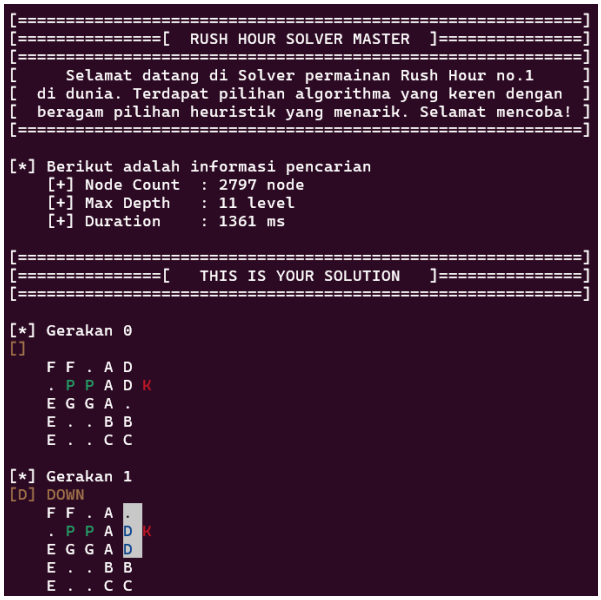
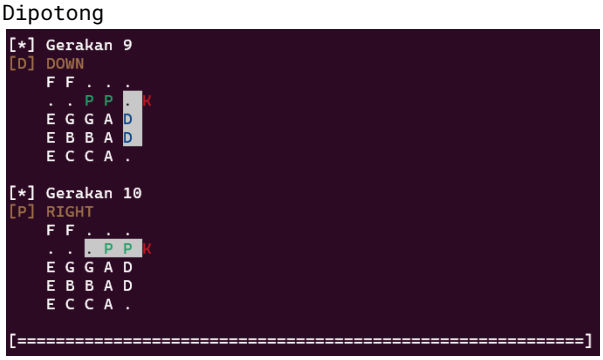
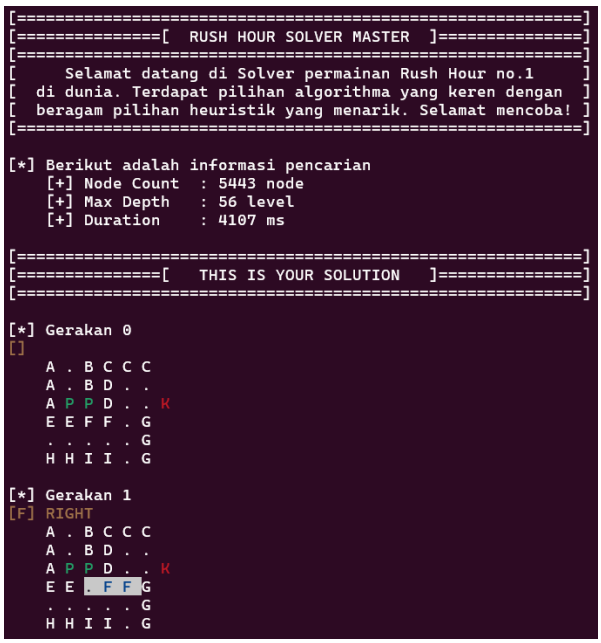
			
12	A*, Heuristik 3	Sama seperti case 3	 Dipotong 

13	A*, Heuristik 4	Sama seperti case 3	 Dipotong 
14	A*, MAX 1 2 3 4	Sama seperti case 4	 Dipotong

			
15	B&B, Tanpa heuristik	Sama seperti case 3	 Dipotong 

16	B&B, Heuristik 1	Sama seperti case 3	 Dipotong 
17	B&B, Heuristik 2	Sama seperti case 3	 Dipotong

			
18	B&B, Heuristik 3	Sama seperti case 3	 Dipotong 

19	B&B, Heursitik 4	Sama seperti case 3	 Dipotong 
20	B&B, AVG 1 2 3 4	Sama seperti case 4	 Dipotong

<table>
<tr>
<td></td>
<td></td>
<td></td>
<td>

```
[*] Gerakan 53
[P] RIGHT
    C C C D . .
    A . . D . .
    A . P P . . K
    A E E F F G
    . . B . . G
    H H B I I G

[*] Gerakan 54
[P] RIGHT
    C C C D . .
    A . . D . .
    A . . P P . K
    A E E F F G
    . . B . . G
    H H B I I G

[*] Gerakan 55
[P] RIGHT
    C C C D . .
    A . . D . .
    A . . . P P K
    A E E F F G
    . . B . . G
    H H B I I G

[==========================================================]
```

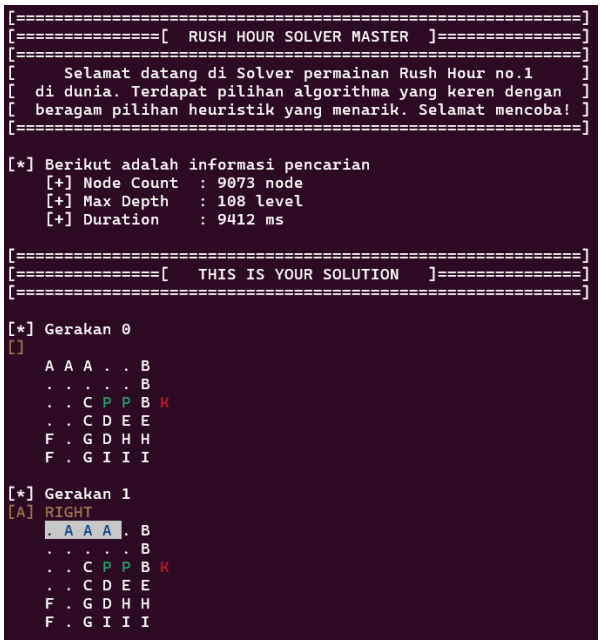
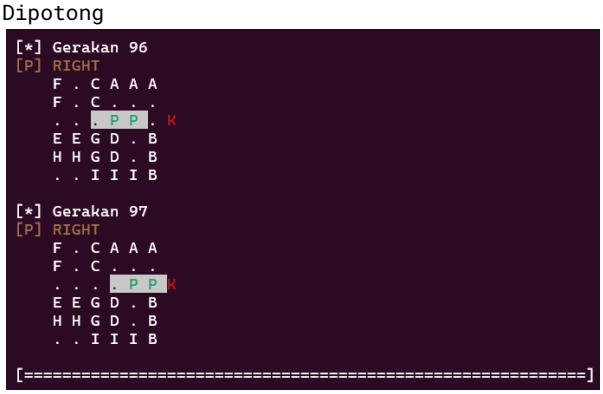
</td>
</tr>
<tr>
<td>21</td>
<td>UCS,
EXPERT</td>
<td> 
</td>
<td>

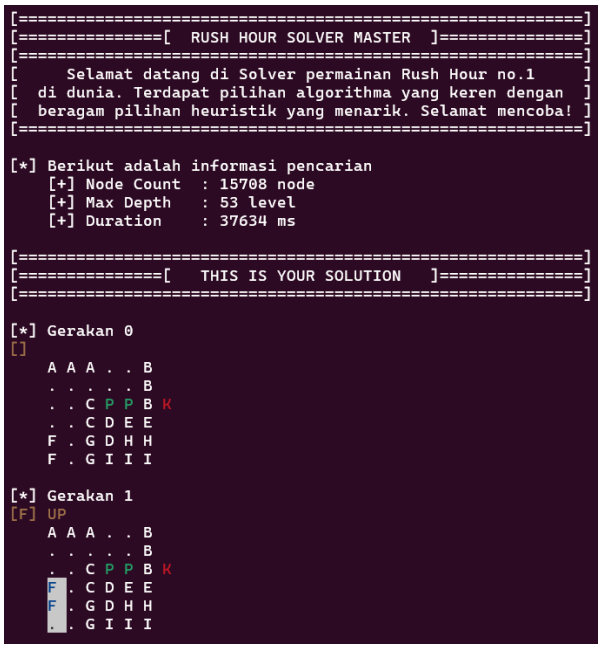
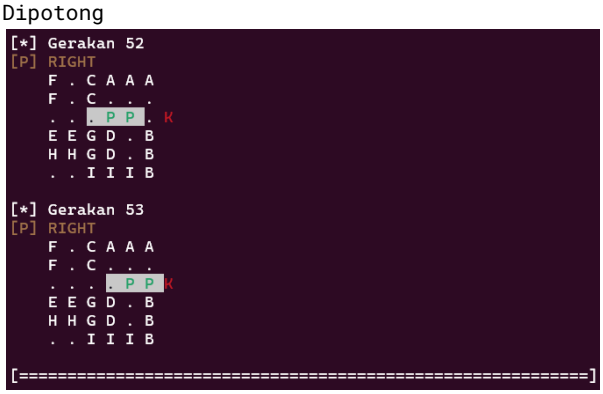
```
[==========================================================]
[===============[  RUSH HOUR SOLVER MASTER  ]===============]
[==========================================================]
[      Selamat datang di Solver permainan Rush Hour no.1     ]
[  di dunia. Terdapat pilihan algorithma yang keren dengan   ]
[  beragam pilihan heuristik yang menarik. Selamat mencoba! ]
[==========================================================]

[*] Berikut adalah informasi pencarian
    [+] Node Count  : 21882 node
    [+] Max Depth   : 90 level
    [+] Duration    : 48720 ms

[==========================================================]
[===============[   THIS IS YOUR SOLUTION   ]===============]
[==========================================================]

[*] Gerakan 0
[]
    A A A . . B
    . . . . . B
    . . C P P B K
    . . C D E E
    F . G D H H
    F . G I I I

[*] Gerakan 1
[A] RIGHT
    . A A A . B
    . . . . . B
    . . C P P B K
    . . C D E E
    F . G D H H
    F . G I I I
```

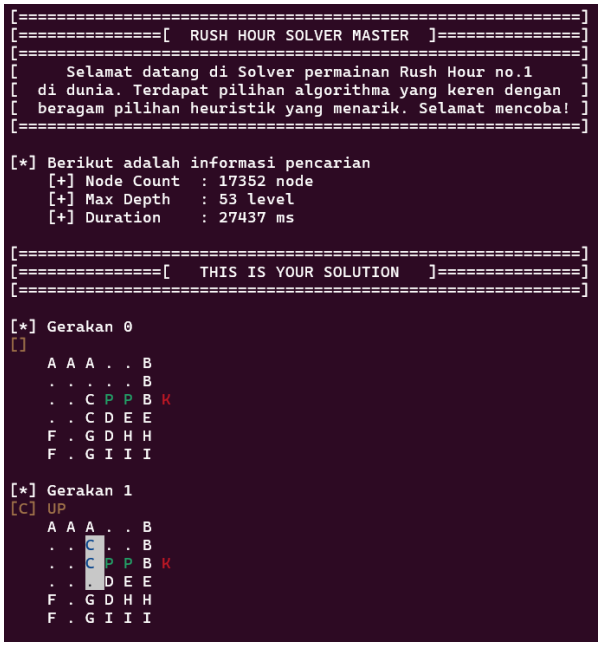
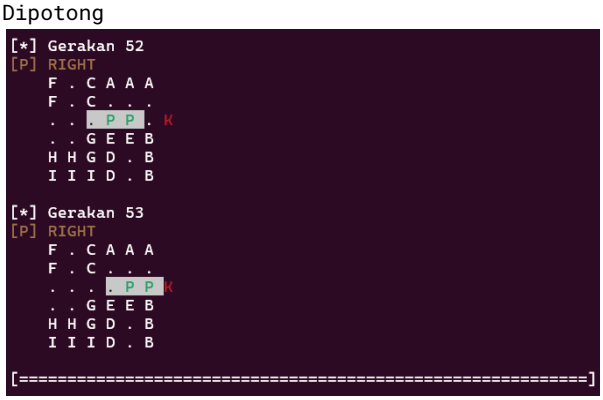
Dipotong

```
[*] Gerakan 85
[B] DOWN
    . . C A A A
    F . C . . .
    F . . P P . K
    E E G D . B
    H H G D . B
    . . I I I B

[*] Gerakan 86
[P] RIGHT
    . . C A A A
    F . C . . .
    F . . . P P K
    E E G D . B
    H H G D . B
    . . I I I B

[==========================================================]
```

</td>
</tr>
</table>

22	GBFS, EXPERT	Case sama seperti case 21. Heuristik yang dipilih: 3	 Dipotong 

23	A*, EXPERT	Case sama seperti case 21. Heuristik yang dipilih: AVG 1 3 4	 Dipotong 

24	B&B, EXPERT	Case sama seperti case 21. Heuristik yang dipilih: MAX 1 2	 Dipotong 

F. Hasil dan Analisis

Hasil menunjukkan bahwa program dapat berjalan dengan baik dan menghasilkan solusi untuk seluruh kasus asli dari permainan Rush Hour. Algoritma A* dan B&B yang menghitung biaya dengan fungsi $f(n) = g(n) + h(n)$ menjadi pilihan yang paling memberikan hasil optimal.

G. Penjelasan Bonus

a. Algoritma Tambahan: *Branch and Bound*

Algoritma ini ditambahkan dengan penghitungan biaya yang sama dengan A*, tetapi melakukan *pruning* cabang-cabang sesuai dengan definisi algoritma B&B.

b. Heuristik Tambahan

Terdapat 4 heuristik yang dapat saling dikombinasikan, mencakup:

1. Jarak *piece* utama ke pintu keluar
2. Jumlah *piece* yang menghalangi pintu keluar
3. Jumlah *piece* penghalang yang terkunci atau tidak dapat digerakkan

4. Jumlah langkah minimum *piece* penghalang untuk menyingkir, dengan asumsi tidak ada *piece* lain yang menghalangi *piece* penghalang tersebut.

Kombinasi heuristik dapat dipilih dengan mengkombinasikan 2 hingga 4 heuristik dengan agregasi berupa *max*, *min*, *sum*, dan *avg*.

H. Lampiran

a. Tabel Pemenuhan Spesifikasi

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	😎	
2	Program berhasil dijalankan	😎	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	😎	
4	Program dapat membaca masukan berkas .txt serta menyimpan solusi 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	😎	