LAPORAN TUGAS KECIL 3 - IF2211 STRATEGI ALGORITMA

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Semester II tahun 2024/2025

disusun oleh:

Jessica Allen (13523059)

Benedict Presley (13523067)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA - KOMPUTASI

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

BAB I
PENJELASAN ALGORITMA

1.1 Algoritma Uniform Cost Search (UCS)

Uniform Cost Search adalah algoritma pencarian berdasarkan biaya kumulatif terendah dari node awal menuju node tujuan. Algorima ini menjamin ketemunya solusi yang optimal apabila costnya tidak negatif. UCS adalah bentuk generalisasi dari Breadth First Search (BFS), tetapi UCS memperhitungkan bobot atau cost antar simpul. UCS memastikan bahwa simpul pertama yang sampai ke tujuan adalah simpul dengan biaya minimum, menjadikannya optimal dan lengkap.

Karakteristik dari algoritma ini adalah merupakan bentuk generalisasi dari Breadth First Search (BFS), tetapi UCS memperhitungkan bobot atau cost (biaya kumulatif (g(n))) antar simpul. UCS memastikan bahwa simpul pertama yang sampai ke tujuan adalah simpul dengan biaya minimum, menjadikannya optimal dan lengkap.

Fungsi evaluasi dari algoritma ini adalah:

$$f(n) = g(n)$$

dengan g(n) menrupakan total biaya dari node awal ke node n.

Cara kerja dari algoirma ini adalah dengan menyimpan simpul awal ke dalam antrian prioritas atau priority queue dengan nilai biaya awal, yang biasanya berjumlah 0. Kemudian, ambil simpul dengan biaya terendah, jika simpul adalah tujuan, hentikan dan kembalikan jalur. Jika belum, ekspansi simpul dengan menambahkan semua tetangganya ke dalam queue dengan biaya kumulatif diperbarui. Jika simpul sudah pernah dikunjungi sebelumnya dengna biaya yang lebih murah, abaikan ekspansi. Lalu, ulangi langkah-langkah tersebut selama queue belum kosong.

Contoh kasus ketika UCS menjadi algoritma pathfinding yang sangat cocok adalah ketika kita ingin mencari sebuah rute termurah dari kota A ke kota B pada peta jalan yang berbobot. Bobot ini dapat berupa bobot jarak ataupun bobot waktu tempuh. UCS akan sangat cocok untuk menyelesaikan masalah ini karena selalu memilih jalan dengan total biaya yang paling kecil sejauh ini.

Kelebihan dari algoritma ini dapat selalu menemukan solusi yang paling optimal dan cocok untuk graf berbobot. UCS dijamin pasti akan menemukan solusi jika memang solusi tersebut ada untuk menyelesaikan permasalahan. Akan tetapi, UCS sebagai algoritma pathfinding juga memiliki beberapa kekurangan, salah satunya adalah tidak efisien dalam waktu dan memori, terutama pada graf yang lebih besar. Selain itu, UCS juga dapat menjelajah node yang tidak relevan terlebih dahulu karena tidak mengetahui arah tujuan (tidak menggunakan heuristik).

1.2 Algoritma Greedy Best First Search

Greedy Best First Search adalah algoritma pencarian yang hanya mempertimbangkan perkiraan jarak dari simpul sekarang ke tujuan. Algoritma ini termasuk dalam algoritma heuristik. Algoritma ini selalu memilih node yang terlihat paling dekat dari tujuan berdasarkan estimasi, bukan hanya biaya aktual dari awal.

Fungsi evaluasi dari algoritma ini adalah

$$f(n) = h(n)$$

dengan h(n) merupakan nilai heuristik, yaitu perkiraan biaya dari simpul n ke tujuan.

Cara kerja dari algoritma ini adalah dengan menyimpan simpul awal dalam antrian prioritas berdasarkan nilai heuristik, kemudian ambil simpul dengan nilai heuristik terendah. Jika simpul merupakan simpul tujuan, maka algoritma berhenti sampai di sana. Jika tidak, maka ekspansi simpul dengan memasukkan seluruh tetangganya ke queue berdasarkan nilai heuristik. Ulangi langkah-langkah sebelumnya hingga simpul tujuan ditemukan.

Salah satu contoh kasus yang dapat memanfaatkan GBFS dengan optimal adalah ketika kita ingin mencari sebuah rute yang tercepat dalam suatu game atau pemetaan, dan kita memiliki estimasi jarak dari suatu simpul ke tujuan, misalnya Euclidean distance dalam peta.

Kelebihan dari algoritma ini adalah sangat cepat dalam eksekusi, karena tidak memperhitungkan jalur panjang. Selain itu, algoritma ini juga sangat cocok untuk ruang pencarian besar serta untuk kasus-kasus ketika kecepatan lebih penting daripada optimalitas. Akan tetapi, terdapat juga beberapa kekurangan dari algoritma ini, yaitu tidak menjamin ditemukannya jalur optimal. Selain itu, algoritma ini juga dapat memunculkan keadaan stuck atau tersesat dalam jalur pencarian jika heuristiknya tidak bagus. Jika graf memiliki loop, dan algoritma tidak memeriksa node yang sudah dikunjungi sebelumnya, maka ada juga kemungkinan bahwa solusi yang dihasilkan dari algoritma ini tidak lengkap.

1.3 Algoritma A*

Algoritma A* atau biasanya dibaca “A star” adalah algoritma pencarian yang menggabungkan algoritma UCS dan juga GBFS. Algoritma ini menggunakan kombinasi biaya aktual (g(n)) dan estimasi biaya ke tujuan (h(n)) dalam menentukan jalur terbaik. Algoritma A* dikenal sebagai salah satu algoritma pencarian yang paling optimal dan efisien apabila heuristik yang didefinisikan baik.

Fungsi evaluasi dari algoritma ini adalah:

$$f(n) = g(n) + h(n)$$

dengan g(n) sebagai biaya dari start ke node n, h(n) sebagai estimasi biaya dari n ke goal, dan f(n) sebagai total estimasi biaya minimum dari awal, ke n, hingga ke tujuan.

Cara kerja dari algoritma ini adalah dengan memasukkan simpul awal ke dalam queue, dengan f(start) = g(start) + h(start). Kemudian, selama queue tidak kosong, langkah selanjutnya adalah mengambil simpul dengan nilai f(n) terkecil. Jika simpul merupakan simpul tujuan, maka algoritma berakhir, Jika tidak, maka lakukan ekspansi dengan memasukkan semua tetangganya dan hitung g(n) baru, hitung h(n) juga berdasarkan fungsi heuristik, kemudian hitung f(n) = g(n) + h(n). Lalu, tambahkan ke queue jika node belum dikunjungi atau lebih murah dari sebelumnya. Ulangi langkah-langkah sebelumnya hingga mencapai tujuan.

Algoritma ini sangat cocok untuk digunakan pada pathfinding di game, misalnya pencarian jalan karakter, rute transportasi seperti Google Maps dan GPS, serta penjadwalan dengan batasan dan biaya tertentu. Terdapat beberapa syarat heuristik untuk algoritma ini, yaitu harus admissible dan consistent. Admissible berarti algoritma ini tidak boleh melebihkan biaya yang sebenarnya ke goal. Consistent atau monotonic berarti h(n) harus lebih kecil dari atau sama dengan c(n, n') + h(n') agar tidak terjadi backtracking.

Kelebihan dari algoritma ini adalah sangat optimal jika heuristiknya admissible dan lebih cepat dari UCS karena algoritma ini terbantu oleh arah heuristik. Selain itu, algoritma ini juga fleksibel karena dapat menyesuaikan dengan berbagai heuristik. Akan tetapi, tentunya algoritma ini juga memiliki kekurangan. Salah satunya, algoritma ini membutuhkan banyak memori untuk menyimpan f(n), g(n), dan visited. Selain itu, waktu dan ruang juga bisa besar apabila ruang pencarian luas atau heuristiknya buruk.

1.4 Algoritma IDS

Iterative Deepening Search (IDS) adalah algoritma pencarian yang menggabungkan kelebihan dari Depth First Search (DFS) dan Breadth First Search (BFS). IDS bertujuan untuk mencari solusi secara inkremental berdasarkan kedalaman, mulai dari kedalaman 0, lalu 1, lalu 2, dan seterusnya hingga menemukan solusi. Pada setiap iterasi, IDS menjalankan DFS terbatas (depth-limited DFS) hingga kedalaman tertentu, dan akan meningkatkannya satu per satu.

Karakteristik utama dari IDS adalah menggunakan sedikit memori seperti DFS (karena tidak menyimpan semua node seperti BFS), tetapi tetap menjamin solusi optimal untuk graf tak berbobot seperti BFS. Hal ini membuat IDS menjadi algoritma yang lengkap dan optimal untuk graf tak berbobot, serta efisien dalam memori.

Fungsi evaluasi IDS dapat dianggap sebagai batas kedalaman saat ini:

$$f(n) = depth(n) \leq L$$

di mana depth(n) adalah kedalaman node saat ini, dan L adalah batas kedalaman iterasi saat ini.

Cara kerja dari algoritma Iterative Deepening Search (IDS) dimulai dengan menetapkan batas kedalaman awal sebesar nol. Pada tahap ini, algoritma akan melakukan pencarian dengan metode Depth First Search (DFS), tetapi hanya hingga batas kedalaman yang telah

ditentukan. Jika pada proses pencarian tersebut ditemukan node tujuan, maka jalur solusi akan langsung dikembalikan dan pencarian dihentikan. Namun, jika tidak ditemukan solusi pada kedalaman tersebut, maka algoritma akan menaikkan batas kedalaman sebanyak satu tingkat dan mengulangi proses DFS dari awal dengan batas yang baru. Proses ini terus diulang secara bertahap, setiap kali menaikkan kedalaman pencarian, sampai solusi berhasil ditemukan atau sampai seluruh kemungkinan jalur telah dieksplorasi. Pendekatan ini memungkinkan IDS untuk menjaga efisiensi memori seperti DFS, namun tetap menjamin ditemukannya solusi optimal seperti BFS pada graf tak berbobot.

Kelebihan dari algoritma Iterative Deepening Search (IDS) terletak pada kemampuannya menggabungkan efisiensi memori dari Depth First Search (DFS) dengan keoptimalan dan kelengkapan dari Breadth First Search (BFS). Karena IDS menggunakan pendekatan DFS terbatas (depth-limited), algoritma ini hanya membutuhkan ruang memori yang relatif kecil, bahkan jauh lebih hemat dibandingkan BFS yang harus menyimpan semua simpul pada satu tingkat kedalaman dalam memori. Selain itu, IDS menjamin bahwa solusi yang ditemukan adalah solusi optimal dalam hal jumlah langkah, karena pencarian selalu dimulai dari kedalaman nol dan meningkat secara bertahap, sehingga solusi paling dangkal akan ditemukan terlebih dahulu. Di samping itu, IDS juga bersifat lengkap, artinya jika terdapat solusi dalam graf, maka algoritma ini pasti akan menemukannya, menjadikannya sangat andal untuk eksplorasi ruang pencarian yang luas pada graf tak berbobot.

Namun, kelemahannya adalah IDS bisa jadi redundan karena node yang sama bisa dieksplorasi kembali pada setiap iterasi. Hal ini membuat IDS kurang efisien secara waktu, terutama jika solusi berada di kedalaman yang jauh, karena akan terjadi pengulangan pencarian di kedalaman-kedalaman sebelumnya.

BAB II
ANALISIS ALGORITMA

Algoritma pencarian dalam ruang status seperti UCS, GBFS, dan A*, masing-masing memiliki pendekatan yang unik dalam mengevaluasi node atau simpul yang akan dieksplorasi selanjutnya berdasarkan fungsi evaluasi tertentu. Dalam konteks ini, fungsi evaluasi dilambangkan dengan f(n), yaitu total "nilai" yang digunakan oleh algoritma untuk memutuskan node mana yang akan diekspansi terlebih dahulu.

Untuk algoritma Uniform Cost Search (UCS), fungsinya adalah f(n) = g(n), di mana g(n) merupakan biaya total dari simpul awal ke simpul n. Hal ini berarti UCS memperluas node berdasarkan biaya kumulatif terkecil yang telah diketahui sejauh ini, tanpa mempertimbangkan seberapa dekat node tersebut dengan node tujuan. Pendekatan ini menjamin solusi yang optimal dalam graf dengan bobot positif, karena UCS selalu memilih jalur dengan biaya yang terkecil dari awal.

Algoritma Greedy Best First Search (GBFS) menggunakan fungsi evaluasi f(n) = h(n), di mana h(n) merupakan estimasi heuristik dari biaya dari node n ke tujuan. Berbeda dengan UCS, GBFS tidak memperhitungkan biaya dari awal ke simpul n. Dengan kata lain, ia hanya berusaha untuk "menebak" node mana yang paling dekat dengan node tujuan berdasarkan heuristik dan merupakan algoritma yang cepat, tetapi tidak menjamin bahwa jalur tersebut murah ataupun optimal karena dapat terjebak pada jalur yang terlihat dekat tetapi sebenarnya lebih mahal secara total. Algoritma ini dapat dikatakan bersifat "myopic" atau pendek pandang karena hanya fokus pada tujuan tanpa memikirkan sejarah perjalanan simpul.

Sedangkan, untuk algoritma A* menggabungkan kedua pendekatan tersebut melalui fungsi f(n) = g(n) + h(n), yaitu dengan mempertimbangkan total biaya dari awal ke node n serta estimasi ke tujuan dari node n tersebut. Gabungan ini memberikan keseimbangan antara eksplorasi jalur terpendek dan arah menuju tujuan.

Salah satu aspek yang sangat penting dalam algoritma A* adalah sifat admissibility dari fungsi heuristik h(n). Berdasarkan definisi dalam salindia kuliah, sebuah heuristik dikatakan admissible jika tidak melebih-lebihkan biaya minimum sebenarnya dari simpul n ke simpul tujuan, atau secara formal h(n) lebih kecil atau sama dengan h*(n) untuk semua n, di mana h*(n) adalah biaya minimum sebenarnya dari n ke goal. Admissibility penting karena menjamin bahwa A* akan menemukan solusi yang optimal, asalkan semua edge memiliki bobot non-negatif. Hal ini karena A* hanya akan mengembangkan node yang secara realistis mungkin menghasilkan solusi terbaik. Jika heuristik tidak admissible, maka A* dapat terjebak dalam jalur yang tampak menjanjikan, padahal sebenarnya lebih mahal, yang kemudian dapat menghasilkan solusi yang tidak optimal. Selain itu, jika heuristik juga konsisten, memenuhi h(n) lebih kecil dari atau sama dengan c(n, n') + h(n'), maka nilai f(n) tidak akan pernah menurun sepanjang jalur. Konsistensi ini memungkinkan A* menjadi lebih efisien karena node yang telah diekspansi tidak perlu dikunjungi ulang lagi, yang kemudian membuat implementasinya lebih sederhana.

Selain itu, penting juga untuk memahami hubungan antara UCS dan Breadth First Search (BFS), dalam konteks penyelesaian permainan seperti Rush Hour. Kedua algoritma ini memang mirip dalam beberapa hal, terutama ketika semua langkah memiliki bobot yang sama. Dalam kasus seperti ini, UCS akan mengeksplorasi node berdasarkan biaya g(n). Yang setara dengan jumlah langkah atau tingkat kedalaman dari root. Hal ini sama sama dengan BFS yang juga menelusuri semua node di level tertentu sebelum melanjutkan ke level berikutnya.

Akan tetapi, algoritma UCS dan BFS tidak selalu sama atau identik. JIka pergerakan dalam permainan memiliki bobot yang bervariasi, seperti langkah tertentu membutuhkan dua gerakan atau lebih, maka UCS akan memperhitungkan bobot ini dalam g(n), sedangkan BFS akan tetap menilai semua langkah secara setara. Oleh karena itu, UCS tidak selalu identik dengan BFS dalam pengurutan eksplorasi node dan jalur yang dihasilkan, terkhusus dalam masalah dunia nyata ketika bobot langkah bisa berbeda.

Secara teoritis, algoritma A* lebih efisien daripada algoritma UCS dalam hal efisien pencarian pada banyak kasus, termasuk di permainan Rush Hour ini, dengan syarat menggunakan heuristik yang baik (admissible dan konsisten). Meskipun UCS menjamin solusi yang optimal, algoritma ini sering mengekspansi banyak node yang kurang relevan terhadap tujuan karena tidak mempertimbangkan arah atau posisi relatif ke goal. Sebaliknya, algoritma A* dengan heuristik yang tepat dapat langsung memfokuskan pencarian ke arah yang lebih menjanjikan. Hal ini secara signifikan dapat mengurangi jumlah node yang perlu diekspansi dan mempercepat pencarian solusi. A* cenderung memprioritaskan node yang tidak hanya memiliki jalur termurah sejauh ini, tetapi juga tampak menjanjikan untuk menuju ke node tujuan, sehingga kombinasi dari g(n) dan h(n) memberikan keseimbangan eksplorasi dan eksploitasi. Misalnya, jika terdapat dua jalur, satu terdiri atas 6 langkah yang langsung menuju tujuan dan satu lagi 4 langkah tetapi menjauh dulu dari tujuan, UCS tentunya akan memilih jalur kedua, sedangkan A* akan mengenali bahwa jalur pertama bisa jadi lebih menguntungkan karena memperhitungkan arah tujuan melalui h(n)

Algoritma Greedy Best First Search memiliki pendekatan yang sangat cepat dalam mencari solusi, tetapi algoritma ini tidak dijamin menemukan solusi yang optimal. Karena GBFS hanya menggunakan h(n) dan mengabaikan g(n), algoritma ini dapat memilih jalur yang seperti langsung menuju tujuan tetapi sebenarnya lebih mahal secara total. Hal ini menyebabkan GBFS dapat melewati solusi terbaik dan terjebak dalam solusi yang kurang optimal atau bahkan tidak mencapai node tujuan apabila heuristiknya buruk. Karena itu, meskipun cepat dalam beberapa kasus, GBFS secara teoritis tidak dapat diandalkan jika ingin jaminan optimalitas dalam menyelesaikan permainan seperti Rush Hour, apalagi ketika struktur permainan kompleks dan pilihan jalur sangat bervariasi.

Algoritma Iterative Deepening Search menjamin bahwa solusi yang ditemukan adalah solusi optimal dalam hal jumlah langkah, karena pencarian selalu dimulai dari kedalaman nol dan meningkat secara bertahap, sehingga solusi paling dangkal akan ditemukan terlebih dahulu. Di samping itu, IDS juga bersifat lengkap, artinya jika terdapat solusi dalam graf, maka algoritma ini pasti akan menemukannya, menjadikannya sangat andal untuk eksplorasi ruang

pencarian yang luas pada graf tak berbobot. Secara implementasi, IDS dibatasi oleh suatu batasan kedalaman agar tidak terjadi perulangan tak berhenti bila solusi memang tidak ada. Jika jumlah langkah jawaban yang paling optimal sangat besar, IDS secara praktek tidak dapat menemukan solusi untuk kasus tersebut.

BAB III
SOURCE PROGRAM

3.1 Perhitungan Heuristik

Terdapat 2 heuristik yang digunakan. Yang pertama adalah heuristik untuk memperhitungkan jarak piece utama ke exit. Yang kedua adalah heuristik untuk memperhitungkan jarak piece utama ke exit dan juga memperhitungkan brp banyak piece lain yang menghalanginya.

```
package model;

import java.util.HashSet;
import java.util.Set;

public class Heuristic {
    int actualHeight;
    int actualWidth;
    int exitRow;
    int exitCol;
    int mode;

    public Heuristic(int actualHeight, int actualWidth, int exitRow, int
exitCol, int mode) {
        this.actualHeight = actualHeight;
        this.actualWidth = actualWidth;
        this.exitRow = exitRow;
        this.exitCol = exitCol;
        this.mode = mode;
    }

    public int calculateBlockerDistanceHeuristic(Node node) {
        int minR = actualHeight, maxR = -1, minC = actualWidth, maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (node.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        boolean horizontal = (minR == maxR);
        int length = horizontal ? (maxC - minC + 1) : (maxR - minR + 1);
```

```
        int distance;
        Set<Character> blockers = new HashSet<>();

        if (horizontal) {
            if(exitRow != minR) {
                return Integer.MAX_VALUE;
            }

            if(exitCol == actualWidth) {
                distance = exitCol - maxC;
                for(int c = maxC + 1; c < exitCol; c++) {
                    char cell = node.getCell(minR, c);
                    if(cell != '.') {
                        blockers.add(cell);
                    }
                }
            }
            else {
                distance = minC - exitCol;
                for(int c = 0; c < minC; c++) {
                    char cell = node.getCell(minR, c);
                    if(cell != '.') {
                        blockers.add(cell);
                    }
                }
            }
        }
        else {
            if(exitCol != minC) {
                return Integer.MAX_VALUE;
            }

            if(exitRow == actualHeight) {
                distance = exitRow - maxR;
                for (int r = maxR + 1; r < exitRow; r++) {
                    char cell = node.getCell(r, minC);
                    if(cell != '.'){
                        blockers.add(cell);
                    }
                }
            }
            else {
                distance = minR - exitRow;
                for (int r = 0; r < minR; r++) {
                    char cell = node.getCell(r, minC);
                    if(cell != '.') {
                        blockers.add(cell);
                    }
```

```
                }
            }
        }

        return distance + blockers.size();
    }

    public int calculateDistanceHeuristic(Node node) {
        int minR = actualHeight, maxR = -1, minC = actualWidth, maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (node.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        boolean horizontal = (minR == maxR);
        int length = horizontal ? (maxC - minC + 1) : (maxR - minR + 1);
        int distance;

        if (horizontal) {
            if(exitRow ≠ minR) {
                return Integer.MAX_VALUE;
            }

            if(exitCol == actualWidth) {
                distance = exitCol - maxC;
            }
            else {
                distance = minC - exitCol;
            }
        }
        else {
            if(exitCol ≠ minC) {
                return Integer.MAX_VALUE;
            }

            if(exitRow == actualHeight) {
                distance = exitRow - maxR;
            }
            else {
                distance = minR - exitRow;
            }
        }
```

```
        return distance;
    }

    int calculateHeuristic(Node node) {
        if(mode == 0) {
            return calculateDistanceHeuristic(node);
        }
        return calculateBlockerDistanceHeuristic(node);
    }
}
```

3.2 Node (Board State)

```
package model;

import java.io.PrintWriter;
import java.util.Arrays;
public class Node {
    private final char[][] grid;
    private int g; // cost from start to current node
    private int h; // heuristic cost to goal
    private int ID;
    private int parentID;
    private String move;

    public Node(char[][] grid, int g, int h, int ID, int parentID, String
move) {
        this.grid = new char[grid.length][];
        for (int i = 0; i < grid.length; ++i) {
            this.grid[i] = Arrays.copyOf(grid[i], grid[i].length);
        }
        this.g = g;
        this.h = h;
        this.ID = ID;
        this.parentID = parentID;
        this.move = move;
    }

    public Node(Node other) {
        this.grid = new char[other.grid.length][];
        for (int i = 0; i < other.grid.length; ++i) {
            this.grid[i] = Arrays.copyOf(other.grid[i],
other.grid[i].length);
        }
```

```
        this.g = other.g;
        this.h = other.h;
        this.ID = other.ID;
        this.parentID = other.parentID;
        this.move = other.move;
    }

    public int cost() {
        return g + h;
    }

    public void printMove() {
        System.out.println(move);
    }

    public void printMove(PrintWriter writer) {
        writer.println(move);
    }

    public void printGrid() {
        for(int i = 0; i < grid.length; ++i) {
            for(int j = 0; j < grid[i].length; ++j) {
                System.out.print(grid[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
    }

    public void printGrid(PrintWriter writer) {
        for(int i = 0; i < grid.length; ++i) {
            for(int j = 0; j < grid[i].length; ++j) {
                writer.print(grid[i][j]);
            }
            writer.println();
        }
        writer.println();
    }

    public void setMove(String move) {
        this.move = move;
    }

    public void setG(int g) {
        this.g = g;
    }

    public int getG() {
```

```
        return g;
    }

    public void setH(int h) {
        this.h = h;
    }

    public int getH() {
        return h;
    }

    public void setID(int ID) {
        this.ID = ID;
    }

    public int getID() {
        return ID;
    }

    public void setParentID(int parentID) {
        this.parentID = parentID;
    }

    public int getParentID() {
        return parentID;
    }

    public void setCell(int i, int j, char piece) {
        grid[i][j] = piece;
    }

    public char getCell(int i, int j) {
        return grid[i][j];
    }

    public int getPieceDirection(int i, int j) { // 0: horizontal, 1:
vertical
        int[] dx = {0, 1, 0, -1};
        int[] dy = {1, 0, -1, 0};

        for(int d = 0; d < 4; d++) {
            int ni = i + dx[d];
            int nj = j + dy[d];
            if(ni >= 0 && ni < grid.length && nj >= 0 && nj <
grid[0].length) {
                if (grid[ni][nj] == grid[i][j]) {
                    return d & 1;
                }
```

```
            }
        }
        return -1;
    }

    public int getPieceLength(int i, int j) {
        int direction = getPieceDirection(i, j);
        int length = 1;

        if(direction == 1) { // vertical
            for(int ni = i + 1; ni < grid.length; ++ni) {
                if (grid[ni][j] == grid[i][j]) {
                    length++;
                } else {
                    break;
                }
            }
            for(int ni = i - 1; ni >= 0; --ni) {
                if (grid[ni][j] == grid[i][j]) {
                    length++;
                } else {
                    break;
                }
            }
        } else { // horizontal
            for(int nj = j + 1; nj < grid[0].length; ++nj) {
                if (grid[i][nj] == grid[i][j]) {
                    length++;
                } else {
                    break;
                }
            }
            for(int nj = j - 1; nj >= 0; --nj) {
                if (grid[i][nj] == grid[i][j]) {
                    length++;
                } else {
                    break;
                }
            }
        }
        return length;
    }

    public boolean canPutPiece(int i, int j, int direction, int length) {
        if(direction == 1) { // vertical
            for(int ni = i; ni < i + length; ++ni) {
                if (grid[ni][j] != '.') {
                    return false;
```

```
                }
            }
        } else { // horizontal
            for(int nj = j; nj < j + length; ++nj) {
                if (grid[i][nj] != '.') {
                    return false;
                }
            }
        }
        return true;
    }

    public String getStringGrid() {
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < grid.length; ++i) {
            for(int j = 0; j < grid[0].length; ++j) {
                sb.append(grid[i][j]);
            }
        }
        return sb.toString();
    }
}
```

3.3 Algoritma Uniform Cost Search

```
package model;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Queue;

public class SolverUCS { // In this case it's just regular BFS
    char[][] startingGrid;
    HashMap<String, Integer> nodeToID;
    ArrayList<Node> nodes;
    int actualWidth;
    int actualHeight;
    int exitRow;
    int exitCol;
```

```
    int nodeID;
    String filename;

    public SolverUCS(char[][] startingGrid, int actualWidth, int
actualHeight, int exitRow, int exitCol, String filename) {
        this.startingGrid = new char[startingGrid.length][];
        for (int i = 0; i < startingGrid.length; ++i) {
            this.startingGrid[i] = Arrays.copyOf(startingGrid[i],
startingGrid[i].length);
        }
        nodeToID = new HashMap<>();
        nodes = new ArrayList<>();
        this.actualWidth = actualWidth;
        this.actualHeight = actualHeight;
        this.exitRow = exitRow;
        this.exitCol = exitCol;
        this.filename = filename;
        nodeID = 0;
    }

    public boolean isSolved(Node b) {
        int minR = actualHeight, maxR = -1, minC = actualWidth, maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (b.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        boolean horizontal = (minR == maxR);

        if(exitCol == actualWidth) {
            if(!horizontal || exitRow != minR) {
                return false;
            }

            for(int c = maxC + 1; c < actualWidth; ++c) {
                if(b.getCell(minR, c) != '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitCol == -1) {
```

```
            if(!horizontal || exitRow ≠ minR) {
                return false;
            }

            for(int c = minC - 1; c ⩾ 0; --c) {
                if(b.getCell(minR, c) ≠ '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitRow == actualHeight) {
            if(horizontal || exitCol ≠ minC) {
                return false;
            }

            for(int r = maxR + 1; r < actualHeight; ++r) {
                if(b.getCell(r, minC) ≠ '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitRow == -1) {
            if(horizontal || exitCol ≠ minC) {
                return false;
            }

            for(int r = minR - 1; r ⩾ 0; --r) {
                if(b.getCell(r, minC) ≠ '.') {
                    return false;
                }
            }
            return true;
        }

        return false;
    }

    public void showSolution(Node b) {
        ArrayList<Node> path = new ArrayList<>();
        Node currentNode = b;
        while(currentNode.getParentID() ≠ -1) {
            path.add(currentNode);
            currentNode = nodes.get(currentNode.getParentID());
        }
        path.add(currentNode);
```

```
        System.out.println("Papan Awal");
        currentNode.printGrid();

        int moveCount = 0;

        Collections.reverse(path);
        for(Node node : path) {
            moveCount++;
            System.out.print("Gerakan " + moveCount + ": ");
            node.printMove();
            node.printGrid();
            System.out.println();
        }

        Node lastNode = path.get(path.size() - 1);

        int minR = actualHeight, maxR = -1, minC = actualWidth, maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (lastNode.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        if(exitCol == actualWidth) {
            int length = maxC - minC + 1;
            for(int c = minC; c <= maxC; ++c) {
                lastNode.setCell(minR, c, '.');
            }
            for(int c = actualWidth - length; c < actualWidth; ++c) {
                lastNode.setCell(minR, c, 'P');
            }
        }
        else if(exitCol == -1) {
            int length = maxC - minC + 1;
            for(int c = minC; c <= maxC; ++c) {
                lastNode.setCell(minR, c, '.');
            }
            for(int c = 0; c < length; ++c) {
                lastNode.setCell(minR, c, 'P');
            }
        }
        else if(exitRow == actualHeight) {
            int length = maxR - minR + 1;
```

```
            for(int r = minR; r ⩽ maxR; ++r) {
                lastNode.setCell(r, minC, '.');
            }
            for(int r = actualHeight - length; r < actualHeight; ++r) {
                lastNode.setCell(r, minC, 'P');
            }
        }
        else if(exitRow == -1) {
            int length = maxR - minR + 1;
            for(int r = minR; r ⩽ maxR; ++r) {
                lastNode.setCell(r, minC, '.');
            }
            for(int r = 0; r < length; ++r) {
                lastNode.setCell(r, minC, 'P');
            }
        }

        moveCount++;
        System.out.println("Gerakan " + moveCount + ": Keluar");
        lastNode.printGrid();
        System.out.println();
    }

    public void showSolutionToFile(Node b) {
        try (PrintWriter out = new PrintWriter(new FileWriter(filename))) {
            ArrayList<Node> path = new ArrayList<>();
            Node currentNode = b;
            while (currentNode.getParentID() ≠ -1) {
                path.add(currentNode);
                currentNode = nodes.get(currentNode.getParentID());
            }
            path.add(currentNode);

            out.println("Total moves: " + (path.size() + 1));
            out.println();

            Collections.reverse(path);
            for (Node node : path) {
                node.printMove(out);
                node.printGrid(out);
                out.println();
            }

            Node lastNode = path.get(path.size() - 1);

            int minR = actualHeight, maxR = -1, minC = actualWidth, maxC =
-1;
            for (int i = 0; i < actualHeight; ++i) {
```

```
                for (int j = 0; j < actualWidth; ++j) {
                    if (lastNode.getCell(i, j) == 'P') {
                        minR = Math.min(minR, i);
                        maxR = Math.max(maxR, i);
                        minC = Math.min(minC, j);
                        maxC = Math.max(maxC, j);
                    }
                }
            }

            if (exitCol == actualWidth) {
                int length = maxC - minC + 1;
                for (int c = minC; c <= maxC; ++c) {
                    lastNode.setCell(minR, c, '.');
                }
                for (int c = actualWidth - length; c < actualWidth; ++c) {
                    lastNode.setCell(minR, c, 'P');
                }
            } else if (exitCol == -1) {
                int length = maxC - minC + 1;
                for (int c = minC; c <= maxC; ++c) {
                    lastNode.setCell(minR, c, '.');
                }
                for (int c = 0; c < length; ++c) {
                    lastNode.setCell(minR, c, 'P');
                }
            } else if (exitRow == actualHeight) {
                int length = maxR - minR + 1;
                for (int r = minR; r <= maxR; ++r) {
                    lastNode.setCell(r, minC, '.');
                }
                for (int r = actualHeight - length; r < actualHeight; ++r) {
                    lastNode.setCell(r, minC, 'P');
                }
            } else if (exitRow == -1) {
                int length = maxR - minR + 1;
                for (int r = minR; r <= maxR; ++r) {
                    lastNode.setCell(r, minC, '.');
                }
                for (int r = 0; r < length; ++r) {
                    lastNode.setCell(r, minC, 'P');
                }
            }

            lastNode.printMove(out);
            lastNode.printGrid(out);
            out.println();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void solve() {
        Queue<Node> queue = new LinkedList<>();

        Node startNode = new Node(startingGrid, 0, 0, 0, -1, "");

        queue.add(startNode);
        nodeToID.put(startNode.getStringGrid(), 0);
        nodes.add(startNode);

        int solutionNodeID = -1;

        while(!queue.isEmpty()) {
            Node currentNode = queue.poll();

            if (isSolved(currentNode)) {
                solutionNodeID = currentNode.getID();
                break;
            }

            boolean[] visited = new boolean[256];

            for(int i = 0; i < actualHeight; ++i) {
                for(int j = 0; j < actualWidth; ++j) {
                    if (currentNode.getCell(i, j) == '.') {
                        continue;
                    }

                    char pieceID = currentNode.getCell(i, j);
                    if (visited[pieceID]) {
                        continue;
                    }
                    visited[pieceID] = true;

                    int vertical = currentNode.getPieceDirection(i, j);
                    int length = currentNode.getPieceLength(i, j);

                    if(vertical == 1) {
                        // System.out.println(pieceID + " vertical");

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i + k, j, '.');
                        }
```

```
                        int left = i;
                        while(left >= 0 && currentNode.getCell(left, j) ==
'.') {
                            left--;
                        }
                        left++;
                        int right = i;
                        while(right + length - 1 < actualHeight &&
currentNode.getCell(right + length - 1, j) == '.') {
                            right++;
                        }
                        right--;

                        for(int k = left; k <= right; ++k) {
                            if(k == i) {
                                continue;
                            }
                            if (currentNode.canPutPiece(k, j, 1, length)) {
                                Node newNode = new Node(currentNode);
                                for(int l = 0; l < length; ++l) {
                                    newNode.setCell(k + l, j, pieceID);
                                }

                                String newNodeKey = newNode.getStringGrid();
                                if (!nodeToID.containsKey(newNodeKey)) {
                                    newNode.setG(currentNode.getG() + 1);
                                    nodeID++;
                                    newNode.setID(nodeID);

newNode.setParentID(currentNode.getID());
                                    if(k < i) {
                                        newNode.setMove("Move " + pieceID +
" up " + (i - k));
                                    }
                                    else {
                                        newNode.setMove("Move " + pieceID +
" down " + (k - i));
                                    }

                                    nodeToID.put(newNodeKey, nodeID);
                                    nodes.add(newNode);

                                    // System.out.println(pieceID + " " + i
+ " " + j + " " + length);
                                    // newNode.printMove();
                                    // newNode.printGrid();
                                    // System.out.println();
```

```
                                    queue.add(newNode);
                                }
                            }
                        }

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i + k, j, pieceID);
                        }
                    }
                    else {
                        // System.out.println(pieceID + " horizontal");

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i, j + k, '.');
                        }

                        int left = j;
                        while(left >= 0 && currentNode.getCell(i, left) ==
'.') {
                            left--;
                        }
                        left++;
                        int right = j;
                        while(right + length - 1 < actualWidth &&
currentNode.getCell(i, right + length - 1) == '.') {
                            right++;
                        }
                        right--;

                        for(int k = left; k <= right; ++k) {
                            if(k == j) {
                                continue;
                            }
                            if (currentNode.canPutPiece(i, k, 0, length)) {
                                Node newNode = new Node(currentNode);
                                for(int l = 0; l < length; ++l) {
                                    newNode.setCell(i, k + l, pieceID);
                                }

                                String newNodeKey = newNode.getStringGrid();
                                if (!nodeToID.containsKey(newNodeKey)) {
                                    newNode.setG(currentNode.getG() + 1);
                                    nodeID++;
                                    newNode.setID(nodeID);

newNode.setParentID(currentNode.getID());
                                    if(k < j) {
                                        newNode.setMove("Move " + pieceID +
```

```
" left " + (j - k));
                                    }
                                    else {
                                        newNode.setMove("Move " + pieceID +
" right " + (k - j));
                                    }

                                    nodeToID.put(newNodeKey, nodeID);
                                    nodes.add(newNode);

                                    // System.out.println(pieceID + " " + i
+ " " + j + " " + length);
                                    // newNode.printMove();
                                    // newNode.printGrid();
                                    // System.out.println();

                                    queue.add(newNode);
                                }
                            }
                        }

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i, j + k, pieceID);
                        }
                    }
                }
            }
        }

        if (solutionNodeID != -1) {
            showSolution(nodes.get(solutionNodeID));
            showSolutionToFile(nodes.get(solutionNodeID));
        } else {
            System.out.println("No solution found.");
        }
    }
}
```

3.4 Algoritma Greedy Best First Search

```
package model;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
```

```
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.PriorityQueue;

public class SolverGBFS {
    char[][] startingGrid;
    HashMap<String, Integer> nodeToID;
    ArrayList<Node> nodes;
    int actualWidth;
    int actualHeight;
    int exitRow;
    int exitCol;
    int nodeID;
    Heuristic heuristic;
    int heuristicType = 0; // 0: distance, 1: blocker-distance
    String filename;

    public SolverGBFS(char[][] startingGrid, int actualWidth, int
actualHeight, int exitRow, int exitCol, int heuristicType, String filename)
{
        this.startingGrid = new char[startingGrid.length][];
        for (int i = 0; i < startingGrid.length; ++i) {
            this.startingGrid[i] = Arrays.copyOf(startingGrid[i],
startingGrid[i].length);
        }
        nodeToID = new HashMap<>();
        nodes = new ArrayList<>();
        this.actualWidth = actualWidth;
        this.actualHeight = actualHeight;
        this.exitRow = exitRow;
        this.exitCol = exitCol;
        nodeID = 0;
        this.heuristicType = heuristicType;
        heuristic = new Heuristic(actualHeight, actualWidth, exitRow,
exitCol, heuristicType);
        this.filename = filename;
    }

    public boolean isSolved(Node b) {
        int minR = actualHeight, maxR = -1, minC = actualWidth, maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (b.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
```

```
                }
            }
        }

        boolean horizontal = (minR == maxR);

        if(exitCol == actualWidth) {
            if(!horizontal || exitRow ≠ minR) {
                return false;
            }

            for(int c = maxC + 1; c < actualWidth; ++c) {
                if(b.getCell(minR, c) ≠ '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitCol == -1) {
            if(!horizontal || exitRow ≠ minR) {
                return false;
            }

            for(int c = minC - 1; c ⩾ 0; --c) {
                if(b.getCell(minR, c) ≠ '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitRow == actualHeight) {
            if(horizontal || exitCol ≠ minC) {
                return false;
            }

            for(int r = maxR + 1; r < actualHeight; ++r) {
                if(b.getCell(r, minC) ≠ '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitRow == -1) {
            if(horizontal || exitCol ≠ minC) {
                return false;
            }

            for(int r = minR - 1; r ⩾ 0; --r) {
```

```
                if(b.getCell(r, minC) ≠ '.') {
                    return false;
                }
            }
            return true;
        }

        return false;
    }

    public void showSolution(Node b) {
        ArrayList<Node> path = new ArrayList<>();
        Node currentNode = b;
        while(currentNode.getParentID() ≠ -1) {
            path.add(currentNode);
            currentNode = nodes.get(currentNode.getParentID());
        }
        path.add(currentNode);

        System.out.println("Papan Awal");
        currentNode.printGrid();

        int moveCount = 0;

        Collections.reverse(path);
        for(Node node : path) {
            moveCount++;
            System.out.print("Gerakan " + moveCount + ": ");
            node.printMove();
            node.printGrid();
            System.out.println();
        }

        Node lastNode = path.get(path.size() - 1);

        int minR = actualHeight, maxR = -1, minC = actualWidth, maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (lastNode.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        if(exitCol == actualWidth) {
```

```
            int length = maxC - minC + 1;
            for(int c = minC; c ⩽ maxC; ++c) {
                lastNode.setCell(minR, c, '.');
            }
            for(int c = actualWidth - length; c < actualWidth; ++c) {
                lastNode.setCell(minR, c, 'P');
            }
        }
        else if(exitCol == -1) {
            int length = maxC - minC + 1;
            for(int c = minC; c ⩽ maxC; ++c) {
                lastNode.setCell(minR, c, '.');
            }
            for(int c = 0; c < length; ++c) {
                lastNode.setCell(minR, c, 'P');
            }
        }
        else if(exitRow == actualHeight) {
            int length = maxR - minR + 1;
            for(int r = minR; r ⩽ maxR; ++r) {
                lastNode.setCell(r, minC, '.');
            }
            for(int r = actualHeight - length; r < actualHeight; ++r) {
                lastNode.setCell(r, minC, 'P');
            }
        }
        else if(exitRow == -1) {
            int length = maxR - minR + 1;
            for(int r = minR; r ⩽ maxR; ++r) {
                lastNode.setCell(r, minC, '.');
            }
            for(int r = 0; r < length; ++r) {
                lastNode.setCell(r, minC, 'P');
            }
        }

        moveCount++;
        System.out.println("Gerakan " + moveCount + ": Keluar");
        lastNode.printGrid();
        System.out.println();
    }

    public void showSolutionToFile(Node b) {
        try (PrintWriter out = new PrintWriter(new FileWriter(filename))) {
            ArrayList<Node> path = new ArrayList<>();
            Node currentNode = b;
            while (currentNode.getParentID() ≠ -1) {
                path.add(currentNode);
```

```
                currentNode = nodes.get(currentNode.getParentID());
            }
            path.add(currentNode);

            out.println("Total moves: " + (path.size() + 1));
            out.println();

            Collections.reverse(path);
            for (Node node : path) {
                node.printMove(out);
                node.printGrid(out);
                out.println();
            }

            Node lastNode = path.get(path.size() - 1);

            int minR = actualHeight, maxR = -1, minC = actualWidth, maxC =
-1;
            for (int i = 0; i < actualHeight; ++i) {
                for (int j = 0; j < actualWidth; ++j) {
                    if (lastNode.getCell(i, j) == 'P') {
                        minR = Math.min(minR, i);
                        maxR = Math.max(maxR, i);
                        minC = Math.min(minC, j);
                        maxC = Math.max(maxC, j);
                    }
                }
            }

            if (exitCol == actualWidth) {
                int length = maxC - minC + 1;
                for (int c = minC; c <= maxC; ++c) {
                    lastNode.setCell(minR, c, '.');
                }
                for (int c = actualWidth - length; c < actualWidth; ++c) {
                    lastNode.setCell(minR, c, 'P');
                }
            } else if (exitCol == -1) {
                int length = maxC - minC + 1;
                for (int c = minC; c <= maxC; ++c) {
                    lastNode.setCell(minR, c, '.');
                }
                for (int c = 0; c < length; ++c) {
                    lastNode.setCell(minR, c, 'P');
                }
            } else if (exitRow == actualHeight) {
                int length = maxR - minR + 1;
                for (int r = minR; r <= maxR; ++r) {
```

```
                    lastNode.setCell(r, minC, '.');
                }
                for (int r = actualHeight - length; r < actualHeight; ++r) {
                    lastNode.setCell(r, minC, 'P');
                }
            } else if (exitRow == -1) {
                int length = maxR - minR + 1;
                for (int r = minR; r <= maxR; ++r) {
                    lastNode.setCell(r, minC, '.');
                }
                for (int r = 0; r < length; ++r) {
                    lastNode.setCell(r, minC, 'P');
                }
            }

            lastNode.printMove(out);
            lastNode.printGrid(out);
            out.println();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void solve() {
        PriorityQueue<Node> queue = new PriorityQueue<>((a, b) -> a.getH() -
b.getH());

        Node startNode = new Node(startingGrid, 0, 0, 0, -1, "");

        queue.add(startNode);
        nodeToID.put(startNode.getStringGrid(), 0);
        nodes.add(startNode);

        int solutionNodeID = -1;

        while(!queue.isEmpty()) {
            Node currentNode = queue.poll();

            if (isSolved(currentNode)) {
                solutionNodeID = currentNode.getID();
                break;
            }

            boolean[] visited = new boolean[256];

            for(int i = 0; i < actualHeight; ++i) {
                for(int j = 0; j < actualWidth; ++j) {
```

```
                    if (currentNode.getCell(i, j) == '.') {
                        continue;
                    }

                    char pieceID = currentNode.getCell(i, j);
                    if (visited[pieceID]) {
                        continue;
                    }
                    visited[pieceID] = true;

                    int vertical = currentNode.getPieceDirection(i, j);
                    int length = currentNode.getPieceLength(i, j);

                    if(vertical == 1) {
                        // System.out.println(pieceID + " vertical");

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i + k, j, '.');
                        }

                        int left = i;
                        while(left >= 0 && currentNode.getCell(left, j) ==
'.') {
                            left--;
                        }
                        left++;
                        int right = i;
                        while(right + length - 1 < actualHeight &&
currentNode.getCell(right + length - 1, j) == '.') {
                            right++;
                        }
                        right--;

                        for(int k = left; k <= right; ++k) {
                            if(k == i) {
                                continue;
                            }
                            if (currentNode.canPutPiece(k, j, 1, length)) {
                                Node newNode = new Node(currentNode);
                                for(int l = 0; l < length; ++l) {
                                    newNode.setCell(k + l, j, pieceID);
                                }

                                String newNodeKey = newNode.getStringGrid();
                                if (!nodeToID.containsKey(newNodeKey)) {

newNode.setH(heuristic.calculateHeuristic(newNode));
                                    nodeID++;
```

```
                                    newNode.setID(nodeID);

newNode.setParentID(currentNode.getID());
                                    if(k < i) {
                                        newNode.setMove("Move " + pieceID +
" up " + (i - k));
                                    }
                                    else {
                                        newNode.setMove("Move " + pieceID +
" down " + (k - i));
                                    }

                                    nodeToID.put(newNodeKey, nodeID);
                                    nodes.add(newNode);

                                    // System.out.println(pieceID + " " + i
+ " " + j + " " + length);
                                    // newNode.printMove();
                                    // newNode.printGrid();
                                    // System.out.println();

                                    queue.add(newNode);
                                }
                            }
                        }

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i + k, j, pieceID);
                        }
                    }
                    else {
                        // System.out.println(pieceID + " horizontal");

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i, j + k, '.');
                        }

                        int left = j;
                        while(left ⩾ 0 && currentNode.getCell(i, left) ==
'.') {
                            left--;
                        }
                        left++;
                        int right = j;
                        while(right + length - 1 < actualWidth &&
currentNode.getCell(i, right + length - 1) == '.') {
                            right++;
                        }
```

```
                        right--;

                        for(int k = left; k <= right; ++k) {
                            if(k == j) {
                                continue;
                            }
                            if (currentNode.canPutPiece(i, k, 0, length)) {
                                Node newNode = new Node(currentNode);
                                for(int l = 0; l < length; ++l) {
                                    newNode.setCell(i, k + l, pieceID);
                                }

                                String newNodeKey = newNode.getStringGrid();
                                if (!nodeToID.containsKey(newNodeKey)) {

newNode.setH(heuristic.calculateHeuristic(newNode));
                                    nodeID++;
                                    newNode.setID(nodeID);

newNode.setParentID(currentNode.getID());
                                    if(k < j) {
                                        newNode.setMove("Move " + pieceID +
" left " + (j - k));
                                    }
                                    else {
                                        newNode.setMove("Move " + pieceID +
" right " + (k - j));
                                    }

                                    nodeToID.put(newNodeKey, nodeID);
                                    nodes.add(newNode);

                                    // System.out.println(pieceID + " " + i
+ " " + j + " " + length);
                                    // newNode.printMove();
                                    // newNode.printGrid();
                                    // System.out.println();

                                    queue.add(newNode);
                                }
                            }
                        }

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i, j + k, pieceID);
                        }
                    }
                }
```

```
            }
        }

        if (solutionNodeID != -1) {
            showSolution(nodes.get(solutionNodeID));
            showSolutionToFile(nodes.get(solutionNodeID));
        } else {
            System.out.println("No solution found.");
        }
    }
}
```

3.5 Algoritma A*

```
package model;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.PriorityQueue;

public class SolverAstar {
    char[][] startingGrid;
    HashMap<String, Integer> nodeToID;
    ArrayList<Node> nodes;
    int actualWidth;
    int actualHeight;
    int exitRow;
    int exitCol;
    int nodeID;
    Heuristic heuristic;
    int heuristicType = 0; // 0: distance, 1: blocker-distance
    String filename;

    public SolverAstar(char[][] startingGrid, int actualWidth, int
actualHeight, int exitRow, int exitCol, int heuristicType, String filename)
{
        this.startingGrid = new char[startingGrid.length][];
        for (int i = 0; i < startingGrid.length; ++i) {
            this.startingGrid[i] = Arrays.copyOf(startingGrid[i],
startingGrid[i].length);
        }
```

```
        nodeToID = new HashMap<>();
        nodes = new ArrayList<>();
        this.actualWidth = actualWidth;
        this.actualHeight = actualHeight;
        this.exitRow = exitRow;
        this.exitCol = exitCol;
        nodeID = 0;
        this.heuristicType = heuristicType;
        heuristic = new Heuristic(actualHeight, actualWidth, exitRow,
exitCol, heuristicType);
        this.filename = filename;
    }

    public boolean isSolved(Node b) {
        int minR = actualHeight, maxR = -1, minC = actualWidth, maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (b.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        boolean horizontal = (minR == maxR);

        if(exitCol == actualWidth) {
            if(!horizontal || exitRow != minR) {
                return false;
            }

            for(int c = maxC + 1; c < actualWidth; ++c) {
                if(b.getCell(minR, c) != '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitCol == -1) {
            if(!horizontal || exitRow != minR) {
                return false;
            }

            for(int c = minC - 1; c >= 0; --c) {
                if(b.getCell(minR, c) != '.') {
                    return false;
```

```
                }
            }
            return true;
        }
        if(exitRow == actualHeight) {
            if(horizontal || exitCol != minC) {
                return false;
            }

            for(int r = maxR + 1; r < actualHeight; ++r) {
                if(b.getCell(r, minC) != '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitRow == -1) {
            if(horizontal || exitCol != minC) {
                return false;
            }

            for(int r = minR - 1; r >= 0; --r) {
                if(b.getCell(r, minC) != '.') {
                    return false;
                }
            }
            return true;
        }

        return false;
    }

    public void showSolution(Node b) {
        ArrayList<Node> path = new ArrayList<>();
        Node currentNode = b;
        while(currentNode.getParentID() != -1) {
            path.add(currentNode);
            currentNode = nodes.get(currentNode.getParentID());
        }
        path.add(currentNode);

        System.out.println("Papan Awal");
        currentNode.printGrid();

        int moveCount = 0;

        Collections.reverse(path);
        for(Node node : path) {
```

```
            moveCount++;
            System.out.print("Gerakan " + moveCount + ": ");
            node.printMove();
            node.printGrid();
            System.out.println();
        }

        Node lastNode = path.get(path.size() - 1);

        int minR = actualHeight, maxR = -1, minC = actualWidth, maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (lastNode.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        if(exitCol == actualWidth) {
            int length = maxC - minC + 1;
            for(int c = minC; c <= maxC; ++c) {
                lastNode.setCell(minR, c, '.');
            }
            for(int c = actualWidth - length; c < actualWidth; ++c) {
                lastNode.setCell(minR, c, 'P');
            }
        }
        else if(exitCol == -1) {
            int length = maxC - minC + 1;
            for(int c = minC; c <= maxC; ++c) {
                lastNode.setCell(minR, c, '.');
            }
            for(int c = 0; c < length; ++c) {
                lastNode.setCell(minR, c, 'P');
            }
        }
        else if(exitRow == actualHeight) {
            int length = maxR - minR + 1;
            for(int r = minR; r <= maxR; ++r) {
                lastNode.setCell(r, minC, '.');
            }
            for(int r = actualHeight - length; r < actualHeight; ++r) {
                lastNode.setCell(r, minC, 'P');
            }
        }
```

```
        else if(exitRow == -1) {
            int length = maxR - minR + 1;
            for(int r = minR; r <= maxR; ++r) {
                lastNode.setCell(r, minC, '.');
            }
            for(int r = 0; r < length; ++r) {
                lastNode.setCell(r, minC, 'P');
            }
        }

        moveCount++;
        System.out.println("Gerakan " + moveCount + ": Keluar");
        lastNode.printGrid();
        System.out.println();
    }

    public void showSolutionToFile(Node b) {
        try (PrintWriter out = new PrintWriter(new FileWriter(filename))) {
            ArrayList<Node> path = new ArrayList<>();
            Node currentNode = b;
            while (currentNode.getParentID() != -1) {
                path.add(currentNode);
                currentNode = nodes.get(currentNode.getParentID());
            }
            path.add(currentNode);

            out.println("Total moves: " + (path.size() + 1));
            out.println();

            Collections.reverse(path);
            for (Node node : path) {
                node.printMove(out);
                node.printGrid(out);
                out.println();
            }

            Node lastNode = path.get(path.size() - 1);

            int minR = actualHeight, maxR = -1, minC = actualWidth, maxC =
-1;
            for (int i = 0; i < actualHeight; ++i) {
                for (int j = 0; j < actualWidth; ++j) {
                    if (lastNode.getCell(i, j) == 'P') {
                        minR = Math.min(minR, i);
                        maxR = Math.max(maxR, i);
                        minC = Math.min(minC, j);
                        maxC = Math.max(maxC, j);
                    }
```

```
                }
            }

            if (exitCol == actualWidth) {
                int length = maxC - minC + 1;
                for (int c = minC; c <= maxC; ++c) {
                    lastNode.setCell(minR, c, '.');
                }
                for (int c = actualWidth - length; c < actualWidth; ++c) {
                    lastNode.setCell(minR, c, 'P');
                }
            } else if (exitCol == -1) {
                int length = maxC - minC + 1;
                for (int c = minC; c <= maxC; ++c) {
                    lastNode.setCell(minR, c, '.');
                }
                for (int c = 0; c < length; ++c) {
                    lastNode.setCell(minR, c, 'P');
                }
            } else if (exitRow == actualHeight) {
                int length = maxR - minR + 1;
                for (int r = minR; r <= maxR; ++r) {
                    lastNode.setCell(r, minC, '.');
                }
                for (int r = actualHeight - length; r < actualHeight; ++r) {
                    lastNode.setCell(r, minC, 'P');
                }
            } else if (exitRow == -1) {
                int length = maxR - minR + 1;
                for (int r = minR; r <= maxR; ++r) {
                    lastNode.setCell(r, minC, '.');
                }
                for (int r = 0; r < length; ++r) {
                    lastNode.setCell(r, minC, 'P');
                }
            }

            lastNode.printMove(out);
            lastNode.printGrid(out);
            out.println();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void solve() {
        PriorityQueue<Node> queue = new PriorityQueue<>((a, b) -> a.getH() -
```

```
b.getH());

        Node startNode = new Node(startingGrid, 0, 0, 0, -1, "");

        queue.add(startNode);
        nodeToID.put(startNode.getStringGrid(), 0);
        nodes.add(startNode);

        int solutionNodeID = -1;

        while(!queue.isEmpty()) {
            Node currentNode = queue.poll();

            if (isSolved(currentNode)) {
                solutionNodeID = currentNode.getID();
                break;
            }

            boolean[] visited = new boolean[256];

            for(int i = 0; i < actualHeight; ++i) {
                for(int j = 0; j < actualWidth; ++j) {
                    if (currentNode.getCell(i, j) == '.') {
                        continue;
                    }

                    char pieceID = currentNode.getCell(i, j);
                    if (visited[pieceID]) {
                        continue;
                    }
                    visited[pieceID] = true;

                    int vertical = currentNode.getPieceDirection(i, j);
                    int length = currentNode.getPieceLength(i, j);

                    if(vertical == 1) {
                        // System.out.println(pieceID + " vertical");

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i + k, j, '.');
                        }

                        int left = i;
                        while(left >= 0 && currentNode.getCell(left, j) ==
'.') {
                            left--;
                        }
                        left++;
```

```
                        int right = i;
                        while(right + length - 1 < actualHeight &&
currentNode.getCell(right + length - 1, j) == '.') {
                            right++;
                        }
                        right--;

                        for(int k = left; k <= right; ++k) {
                            if(k == i) {
                                continue;
                            }
                            if (currentNode.canPutPiece(k, j, 1, length)) {
                                Node newNode = new Node(currentNode);
                                for(int l = 0; l < length; ++l) {
                                    newNode.setCell(k + l, j, pieceID);
                                }

                                String newNodeKey = newNode.getStringGrid();
                                if (!nodeToID.containsKey(newNodeKey)) {
                                    newNode.setG(currentNode.getG() + 1);

newNode.setH(heuristic.calculateHeuristic(newNode));
                                    nodeID++;
                                    newNode.setID(nodeID);

newNode.setParentID(currentNode.getID());
                                    if(k < i) {
                                        newNode.setMove("Move " + pieceID +
" up " + (i - k));
                                    }
                                    else {
                                        newNode.setMove("Move " + pieceID +
" down " + (k - i));
                                    }

                                    nodeToID.put(newNodeKey, nodeID);
                                    nodes.add(newNode);

                                    // System.out.println(pieceID + " " + i
+ " " + j + " " + length);
                                    // newNode.printMove();
                                    // newNode.printGrid();
                                    // System.out.println();

                                    queue.add(newNode);
                                }
                            }
                        }
```

```
                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i + k, j, pieceID);
                        }
                    }
                    else {
                        // System.out.println(pieceID + " horizontal");

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i, j + k, '.');
                        }

                        int left = j;
                        while(left ⩾ 0 && currentNode.getCell(i, left) ==
'.') {
                            left--;
                        }
                        left++;
                        int right = j;
                        while(right + length - 1 < actualWidth &&
currentNode.getCell(i, right + length - 1) == '.') {
                            right++;
                        }
                        right--;

                        for(int k = left; k ⩽ right; ++k) {
                            if(k == j) {
                                continue;
                            }
                            if (currentNode.canPutPiece(i, k, 0, length)) {
                                Node newNode = new Node(currentNode);
                                for(int l = 0; l < length; ++l) {
                                    newNode.setCell(i, k + l, pieceID);
                                }

                                String newNodeKey = newNode.getStringGrid();
                                if (!nodeToID.containsKey(newNodeKey)) {
                                    newNode.setG(currentNode.getG() + 1);

newNode.setH(heuristic.calculateHeuristic(newNode));
                                    nodeID++;
                                    newNode.setID(nodeID);

newNode.setParentID(currentNode.getID());
                                    if(k < j) {
                                        newNode.setMove("Move " + pieceID +
" left " + (j - k));
                                    }
```

```
                                    else {
                                        newNode.setMove("Move " + pieceID +
" right " + (k - j));
                                    }

                                    nodeToID.put(newNodeKey, nodeID);
                                    nodes.add(newNode);

                                    // System.out.println(pieceID + " " + i
+ " " + j + " " + length);
                                    // newNode.printMove();
                                    // newNode.printGrid();
                                    // System.out.println();

                                    queue.add(newNode);
                                }
                            }
                        }

                        for(int k = 0; k < length; ++k) {
                            currentNode.setCell(i, j + k, pieceID);
                        }
                    }
                }
            }
        }

        if (solutionNodeID != -1) {
            showSolution(nodes.get(solutionNodeID));
            showSolutionToFile(nodes.get(solutionNodeID));
        } else {
            System.out.println("No solution found.");
        }
    }
}
```

3.6 Input Processor

```
package model;

import java.io.*;
import java.util.*;

public class InputParser {
    public static char[][] readInput(String fileName, int[] ukuran, int[]
exitPos) throws IOException {
```

```
        try (BufferedReader reader = new BufferedReader(new
FileReader(fileName))) {
            String[] dimension = reader.readLine().trim().split(" ");
            int height = Integer.parseInt(dimension[0]);
            int width = Integer.parseInt(dimension[1]);

            int nonPrimaryPieces =
Integer.parseInt(reader.readLine().trim());

            int exitRow = -100;
            int exitCol = -100;
            List<String> initialBoard = new ArrayList<>();

            boolean foundK = false;

            for (int i = 0; i < height; i++) {
                String line = reader.readLine();
                if (line == null) {
                    throw new IllegalArgumentException("Jumlah baris papan
tidak sesuai dengan inout yang diberikan.");
                }

                if (!foundK) {
                    boolean spaceChar = false;
                    for (int j = 0; j < line.length(); j++) {
                        if (line.charAt(j) == 'K') {
                            foundK = true;
                            exitRow = i;
                            exitCol = j;
                        }
                        if (line.charAt(j) == ' ') {
                            spaceChar = true;
                        }
                    }
                    if(foundK && i == 0 && exitCol < width && (spaceChar ||
line.length() == 1)) {
                        exitRow = -1;
                        continue;
                    }
                    if(line.length() == width + 1) {
                        if(line.charAt(0) == ' ') {
                            line = line.substring(1);
                        } else if(line.charAt(0) == 'K') {
                            exitCol = -1;
                            line = line.substring(1);
                        } else {
                            line = line.substring(0, width);
                        }
```

```
                    }
                }
                else {
                    if (line.length() == width + 1) {
                        if (line.charAt(0) == ' ') {
                            line = line.substring(1);
                        }
                        else {
                            line = line.substring(0, width);
                        }
                    }
                }

                initialBoard.add(line);
            }

            if (!foundK || exitRow == -1) {
                String line = reader.readLine();
                if (line == null) {
                    throw new IllegalArgumentException("Jumlah baris papan
tidak sesuai dengan input ukuran.");
                }

                if (exitRow == -1) {
                    for (int j = 0; j < line.length(); j++) {
                        if (line.charAt(j) == 'K') {
                            foundK = true;
                            exitRow = height;
                            exitCol = j;
                            break;
                        }
                    }
                }
                else {
                    for (int j = 0; j < line.length(); j++) {
                        if (line.charAt(j) == 'K') {
                            foundK = true;
                            exitRow = height;
                            exitCol = j;
                            break;
                        }
                    }
                }

                initialBoard.add(line);
            }

            if (!foundK) {
```

```
                throw new IllegalArgumentException("Papan tidak memiliki
pintu keluar (K).");
            }


            char[][] grid = new char[height][width];
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    grid[i][j] = initialBoard.get(i).charAt(j);
                }
            }

            ukuran[0] = height;
            ukuran[1] = width;
            exitPos[0] = exitRow;
            exitPos[1] = exitCol;

            return grid;
        }
    }
}
```

BAB IV
MASUKAN DAN KELUARAN PROGRAM

Kasus #1: UCS Kiri

Input	Output
4 4 1 .... .A.. K.APP	Papan Awal . . . . . A . . . A P P . . . . Gerakan 1: . . . . . A . . . A P P . . . . Gerakan 2: Move A up 1 . A . . . A . . . . P P . . . . Gerakan 3: Keluar . A . . . A . . P P . . . . . .

Kasus #2: UCS Kanan

Input	Output
4 4 1	Papan Awal . . . .

.... ..A. PPA.K	. . A . P P A . . . . . Gerakan 1: . . . . . . A . P P A . . . . . Gerakan 2: Move A up 1 . . A . . . A . P P . . . . . . Gerakan 3: Keluar . . A . . . A . . . P P . . . .

Kasus #3: UCS Atas

Input	Output
4 4 1 K .... .AA. ..P. ..P.	Papan Awal . . . . . A A . . . P . . . P . Gerakan 1: . . . . . A A . . . P . . . P .

	Gerakan 2: Move A left 1 . . . . A A . . . . P . . . P . Gerakan 3: Keluar . . P . A A P . . . .

Kasus #4: UCS Bawah

Input	Output
4 4 1 ..P. ..P. .AA. .... K	Papan Awal . . P . . . P . . A A . . . . . Gerakan 1: . . P . . . P . . A A . . . . . Gerakan 2: Move A left 1 . . P . . . P . A A . . . . . .

	Gerakan 3: Keluar . . . . A A P . . . P .

Kasus #5: GBFS Kiri

Input	Output
4 4 1 .... .A.. K.APP	Papan Awal . . . . . A . . . A P P . . . . Gerakan 1: . . . . . A . . . A P P . . . . Gerakan 2: Move A up 1 . A . . . A . . . . P P . . . . Gerakan 3: Keluar . A . . . A . . P P . . . . . .

Kasus #6: GBFS Kanan

Input	Output
4 4 1 .... ..A. PPA.K	Papan Awal . . . . . . A . P P A . . . . . Gerakan 1: . . . . . . A . P P A . . . . . Gerakan 2: Move A up 1 . . A . . . A . P P . . . . . . Gerakan 3: Keluar . . A . . . A . . . P P . . . .

Kasus #7: GBFS Atas

Input	Output
4 4 1 K .... .AA. ..P. ..P.	Papan Awal . . . . . A A . . . P . . . P . Gerakan 1: . . . .

	. A A . . . P . . . P . Gerakan 2: Move A left 1 . . . . A A . . . . P . . . P . Gerakan 3: Keluar . . P . A A P . . . .

Kasus #8: GBFS Bawah

Input	Output
4 4 1 ..P. ..P. .AA. .... K	Papan Awal . . P . . . P . . A A . . . . . Gerakan 1: . . P . . . P . . A A . . . . . Gerakan 2: Move A left 1 . . P . . . P . A A . .

	. . . . Gerakan 3: Keluar . . . . A A P . . . P .

Kasus #9: A* Kiri

Input	Output
4 4 1 .... .A.. K.APP	Papan Awal . . . . . A . . . A P P . . . . Gerakan 1: . . . . . A . . . A P P . . . . Gerakan 2: Move A up 1 . A . . . A . . . . P P . . . . Gerakan 3: Keluar . A . . . A . . P P . . . . . .

Kasus #10: A* Kanan

Input	Output
4 4 1 .... ..A. PPA.K	Papan Awal . . . . . . A . P P A . . . . . Gerakan 1: . . . . . . A . P P A . . . . . Gerakan 2: Move A up 1 . . A . . . A . P P . . . . . . Gerakan 3: Keluar . . A . . . A . . . P P . . . .

Kasus #11: A* Atas

Input	Output
4 4 1 K .... .AA. ..P.	Papan Awal . . . . . A A . . . P . . . P .

..P.	Gerakan 1: . . . . . A A . . . P . . . P . Gerakan 2: Move A left 1 . . . . A A . . . . P . . . P . Gerakan 3: Keluar . . P . A A P . . . .

Kasus #12: A* Bawah

Input	Output
4 4 1 ..P. ..P. .AA. .... K	Papan Awal . . P . . . P . . A A . . . . . Gerakan 1: . . P . . . P . . A A . . . . . Gerakan 2: Move A left 1 . . P .

	. . P . A A . . . . . . Gerakan 3: Keluar . . . . A A P . . . P .

Kasus #13: IDS Kiri

Input	Output
4 4 1 .... .A.. K.APP	Papan Awal . . . . . A . . . A P P . . . . Gerakan 1: . . . . . A . . . A P P . . . . Gerakan 2: Move A up 1 . A . . . A . . . . P P . . . . Gerakan 3: Keluar . A . . . A . . P P . .

	. . . .

Kasus #14: IDS Kanan

Input	Output
4 4 1 .... ..A. PPA.K	. . . . . . A . P P A . . . . . Gerakan 1: . . . . . . A . P P A . . . . . Gerakan 2: Move A up 1 . . A . . . A . P P . . . . . . Gerakan 3: Keluar . . A . . . A . . . P P . . . .

Kasus #15: IDS Atas

Input	Output
4 4 1 K	Papan Awal . . . . . A A .

.... .AA. ..P. ..P.	. . P . . . P . Gerakan 1: . . . . . A A . . . P . . . P . Gerakan 2: Move A left 1 . . . . A A . . . . P . . . P . Gerakan 3: Keluar . . P . A A P . . . .

Kasus #16: IDS Bawah

Input	Output
4 4 1 ..P. ..P. .AA. .... K	Papan Awal . . P . . . P . . A A . . . . . Gerakan 1: . . P . . . P . . A A . . . . .

	Gerakan 2: Move A left 1 . . P . . . P . A A . . . . . . Gerakan 3: Keluar . . . . A A P . . . P .

BAB V
HASIL ANALISIS PERCOBAAN

Uniform Cost Search (UCS) memiliki kompleksitas waktu dan ruang sebesar $O(b^{1+\frac{C}{\epsilon}})$, dengan b adalah branching factor (jumlah rata-rata anak per node), C adalah biaya solusi optimal, dan ϵ adalah biaya minimum antar edge. Kompleksitas ini muncul karena UCS mengeksplorasi seluruh simpul dengan biaya kumulatif kurang dari C, menjadikannya sangat mahal dalam kasus graf dengan banyak edge berbiaya rendah. UCS menjamin solusi optimal selama semua biaya lintasan tidak negatif, tetapi kelemahannya terletak pada penggunaan memori yang besar akibat penyimpanan semua node dalam antrian prioritas.

Greedy Best-First Search (GBFS) hanya menggunakan fungsi heuristik h(n) untuk menentukan prioritas eksplorasi node, tanpa mempertimbangkan total biaya dari awal. Kompleksitas waktu dari GBFS dalam kasus terburuk adalah $O(b^m)$, dengan m adalah kedalaman maksimum solusi. Karena tidak menyimpan informasi biaya sebelumnya, GBFS bisa sangat cepat dalam menemukan solusi, namun tidak menjamin optimalitas dan cenderung tersesat jika heuristik tidak akurat. Kompleksitas ruangnya juga eksponensial karena penggunaan struktur data seperti priority queue.

A* menggabungkan kelebihan UCS dan GBFS dengan memanfaatkan fungsi evaluasi f(n)=g(n)+h(n). Dengan asumsi bahwa heuristik h(n) adalah *admissible* dan *consistent*, A* menjamin solusi optimal dan cenderung lebih efisien dibanding UCS. Kompleksitas waktu dan ruangnya dalam kasus terburuk masih eksponensial, yaitu $O(b^d)$, dengan d adalah kedalaman solusi. Masalah utama A* adalah kebutuhan memori yang besar karena menyimpan semua node dalam memori untuk menjamin optimalitas.

Iterative Deepening Search (IDS) menggabungkan keuntungan DFS (memori kecil) dan BFS (kelengkapan), dengan melakukan DFS terbatas kedalaman secara berulang. Kompleksitas waktu IDS adalah $O(b^d)$ karena pada akhirnya semua node hingga kedalaman d dieksplorasi. Namun, IDS hanya membutuhkan memori sebesar O(d), karena sifat rekursif dari DFS. IDS sangat efektif pada pencarian solusi yang tidak terlalu dalam dan ketika tidak ada informasi heuristik, tetapi tidak efisien untuk graf dengan cabang lebar atau solusi sangat dalam karena banyaknya eksplorasi ulang.

Secara keseluruhan, pemilihan algoritma sangat bergantung pada struktur graf, ketersediaan heuristik, serta kebutuhan terhadap optimalitas dan efisiensi ruang.

BAB VI (BONUS)
PENJELASAN IMPLEMENTASI BONUS

5.1 GUI

Implementasi bonus GUI dilakukan dengan menggunakan kakas JavaFX. Berikut adalah kode implementasinya.

5.1.1 BoardView.java

```
package gui;

import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.geometry.Pos;
import java.util.*;

public class BoardView {
    private final GridPane grid;
    private final Map<Character, Color> colorMap = new
HashMap<>();
    private int cellSize;

    public BoardView(char[][] initialBoard, int cellSize) {
        this.cellSize = cellSize;
        grid = new GridPane();
        grid.setHgap(2);
        grid.setVgap(2);
        grid.setStyle("-fx-background-color: lightgray;");
        grid.setAlignment(Pos.CENTER);

        drawBoard(initialBoard);
    }

    public void updateBoard(char[][] newBoard) {
        drawBoard(newBoard);
    }

    public void setCellSize(int size) {
```

```
        this.cellSize = size;
    }

    public Pane getView() {
        return grid;
    }

    private void drawBoard(char[][] board) {
        grid.getChildren().clear();
        Random rand = new Random();

        int rows = board.length;
        int cols = board[0].length;

        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                char c = board[row][col];
                Rectangle rect = new Rectangle(cellSize,
cellSize);

                if (c == '.') {
                    rect.setFill(Color.WHITE);
                } else if (c == 'P') {
                    rect.setFill(Color.RED);
                } else {
                    colorMap.putIfAbsent(c,
Color.color(rand.nextDouble(), rand.nextDouble(),
rand.nextDouble()));
                    rect.setFill(colorMap.get(c));
                }

                rect.setStroke(Color.BLACK);
                grid.add(rect, col, row);
            }
        }
    }
}
```

5.1.2 Controller.java

```
package gui;

import javafx.scene.Parent;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.stage.FileChooser;
import model.InputParser;
import javafx.scene.layout.StackPane;
import java.io.*;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import java.util.*;
import model.SolverAstar;
import model.SolverUCS;
import model.SolverGBFS;
import model.SolverIDS;

public class Controller {
    private final BorderPane rootPane;
    private final VBox controlPanel;
    private final Label headerLabel;
    private final ComboBox<String> algorithmSelector;
    private final ComboBox<String> heuristicSelector;
    private final Button startButton;
    private final Button uploadButton;
    private final Label moveCountLabel;
    private File inputFile;
    private String selectedAlgorithm;
    private BoardView boardView;
    private StackPane centerPane;

    private List<char[][]> loadedStates = new ArrayList<>();

    public Controller() {
        rootPane = new BorderPane();
```

```
        controlPanel = new VBox(10);
        controlPanel.setAlignment(Pos.CENTER);
        controlPanel.setPadding(new Insets(20));

        headerLabel = new Label("Rush Hour PathFinder");
        headerLabel.getStyleClass().add("header-label");

        algorithmSelector = new ComboBox<>();
        algorithmSelector.getItems().addAll("UCS", "GBFS",
"A*", "IDS");
        algorithmSelector.setPromptText("Select Algorithm");

        heuristicSelector = new ComboBox<>();
        heuristicSelector.getItems().addAll("Distance",
"Blocking + Distance");
        heuristicSelector.setPromptText("Select Heuristic");

        HBox controlsGroup = new HBox(20);
        controlsGroup.setAlignment(Pos.CENTER);
        controlsGroup.getChildren().addAll(algorithmSelector,
heuristicSelector);

        moveCountLabel = new Label("Total Moves: -");

        controlPanel.getChildren().addAll(headerLabel,
controlsGroup, moveCountLabel);
        rootPane.setTop(controlPanel);
        BorderPane.setAlignment(controlPanel, Pos.CENTER);

        uploadButton = new Button("Upload Input File");
        uploadButton.setOnAction(e → loadInputFile());

        startButton = new Button("Start");
        startButton.setOnAction(e → {
            selectedAlgorithm = algorithmSelector.getValue();
            String selectedHeuristic =
heuristicSelector.getValue();
            if (inputFile ≠ null && selectedAlgorithm ≠ null)
{
                runSolverAndLoadOutput(inputFile.getName(),
```

```
selectedAlgorithm, selectedHeuristic);
            } else {
                moveCountLabel.setText("Please select algorithm
and input file.");
            }
        });

        HBox bottomControls = new HBox(20);
        bottomControls.setAlignment(Pos.CENTER);
        bottomControls.setPadding(new Insets(20, 20, 40, 20));
        bottomControls.getChildren().addAll(uploadButton,
startButton);
        rootPane.setBottom(bottomControls);

        boardView = new BoardView(new char[1][1], 50);
        centerPane = new StackPane(boardView.getView());
        centerPane.setAlignment(Pos.CENTER);
        rootPane.setCenter(centerPane);
    }

    public Parent getView() {
        return rootPane;
    }

    private void loadInputFile() {
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Open Input File");

        fileChooser.setInitialDirectory(new
File("test/input/"));
        fileChooser.getExtensionFilters().add(new
FileChooser.ExtensionFilter("Text Files", "*.txt"));

        inputFile =
fileChooser.showOpenDialog(rootPane.getScene().getWindow());

        if (inputFile ≠ null) {
            moveCountLabel.setText("File loaded: " +
inputFile.getName());
        }
```

```
    }

    private void parseFile(File file) throws IOException {
        loadedStates.clear();
        try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
            String line;
            List<char[]> currentBoard = new ArrayList<>();
            while ((line = reader.readLine()) != null) {
                line = line.trim();
                if (line.isEmpty()) {
                    if (!currentBoard.isEmpty()) {

loadedStates.add(currentBoard.toArray(new char[0][]));
                        currentBoard.clear();
                    }
                } else if (!line.startsWith("Move") &&
!line.startsWith("Total")) {
                    currentBoard.add(line.toCharArray());
                }
            }
            if (!currentBoard.isEmpty()) {
                loadedStates.add(currentBoard.toArray(new
char[0][]));
            }
        }
    }

    private void runSolverAndLoadOutput(String inputFileName,
String algorithm, String heuristic) {
        try {
            String inputFolder = "test/input/";
            String outputFolder = "test/output/";

            String fileName = inputFolder + inputFileName;
            int[] ukuran = new int[2];
            int[] exitPos = new int[2];

            char[][] board = InputParser.readInput(fileName,
ukuran, exitPos);
```

```
            int heuristicType = -1;

            if (heuristic.equals("Distance")) {
                heuristicType = 0;
            } else if (heuristic.equals("Blocking + Distance"))
{
                heuristicType = 1;
            } else {
                moveCountLabel.setText("Heuristic not
supported.");
                return;
            }

            fileName = outputFolder + inputFileName;

            if (algorithm.equals("UCS")) {
                SolverUCS solver = new SolverUCS(board,
ukuran[0], ukuran[1], exitPos[0], exitPos[1], fileName);
                solver.solve();
            } else if (algorithm.equals("GBFS")) {
                SolverGBFS solver = new SolverGBFS(board,
ukuran[0], ukuran[1], exitPos[0], exitPos[1], 1, fileName);
                solver.solve();
            } else if (algorithm.equals("A*")) {
                SolverAstar solver = new SolverAstar(board,
ukuran[0], ukuran[1], exitPos[0], exitPos[1], 1, fileName);
                solver.solve();
            } else if (algorithm.equals("IDS")) {
                SolverIDS solver = new SolverIDS(board,
ukuran[0], ukuran[1], exitPos[0], exitPos[1], fileName);
                solver.solve();
            } else {
                moveCountLabel.setText("Algorithm not
supported.");
                return;
            }

            File outputFile = new File(outputFolder +
inputFileName);
```

```
            if (!outputFile.exists()) {
                moveCountLabel.setText("Output file not found:
" + outputFile.getName());
                return;
            }

            parseFile(outputFile);

            if (!loadedStates.isEmpty()) {
                boardView = new BoardView(loadedStates.get(0),
50);

centerPane.getChildren().setAll(boardView.getView());
                moveCountLabel.setText("Total Moves: " +
(loadedStates.size() - 1));
                playAnimation();
            } else {
                moveCountLabel.setText("No states loaded from
output file.");
            }

        } catch (IOException e) {
            e.printStackTrace();
            moveCountLabel.setText("Failed to read input/output
files.");
        } catch (Exception e) {
            e.printStackTrace();
            moveCountLabel.setText("Error running solver.");
        }
    }


    private void playAnimation() {
        if (loadedStates.isEmpty()) return;

        new Thread(() → {
            try {
                for (char[][] frame : loadedStates) {
                    javafx.application.Platform.runLater(() →
```

```
boardView.updateBoard(frame));
                    Thread.sleep(500);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

5.1.3 MainApp.java

```
package gui;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import java.io.File;
import java.net.URL;

public class MainApp extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
            Controller controller = new Controller();
            Scene scene = new Scene(controller.getView(), 600,
600);

            boolean cssLoaded = false;

            URL cssResource =
getClass().getResource("style.css");
            if (cssResource ≠ null) {

scene.getStylesheets().add(cssResource.toExternalForm());
                cssLoaded = true;
```

```
                System.out.println("CSS loaded via direct
resource");
            }

            if (!cssLoaded) {
                File srcCssFile = new
File("src/gui/style.css");
                if (srcCssFile.exists()) {

scene.getStylesheets().add(srcCssFile.toURI().toURL().toExterna
lForm());
                    cssLoaded = true;
                    System.out.println("CSS loaded from src
directory");
                }
            }

            if (!cssLoaded) {
                File binCssFile = new
File("bin/gui/style.css");
                if (binCssFile.exists()) {

scene.getStylesheets().add(binCssFile.toURI().toURL().toExterna
lForm());
                    cssLoaded = true;
                    System.out.println("CSS loaded from bin
directory");
                }
            }

            if (!cssLoaded) {
                System.out.println("Could not load external
CSS, using inline styles");
                String css =
                    ".root { -fx-font-family: \"Segoe UI\",
sans-serif; -fx-background-color: #f8f9fa; }" +
                    ".header-label { -fx-font-size: 36px;
-fx-font-weight: extrabold; -fx-text-fill: #2c3e50;
-fx-padding: 20 0 10 0; }" +
                    ".combo-box { -fx-pref-width: 200;
```

```
-fx-font-size: 14px; }" +
                    ".button { -fx-background-color: #27ae60;
-fx-text-fill: white; -fx-font-size: 14px; -fx-font-weight:
bold; -fx-background-radius: 8; -fx-padding: 8 16; }";

                scene.getStylesheets().add("data:text/css," +
css);
            }

            primaryStage.setTitle("Rush Hour PathFinder");
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (Exception e) {
            System.err.println("Error starting application: " +
e.getMessage());
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

5.1.4 style.css

```
.root {
    -fx-font-family: "Segoe UI", sans-serif;
    -fx-background-color: #f8f9fa;
}

.header-label {
    -fx-font-size: 36px;
    -fx-font-weight: extrabold;
    -fx-text-fill: #2c3e50;
    -fx-padding: 20 0 10 0;
}
```

```
.combo-box {
    -fx-pref-width: 200;
    -fx-font-size: 14px;
}

.button {
    -fx-background-color: #27ae60;
    -fx-text-fill: white;
    -fx-font-size: 14px;
    -fx-font-weight: bold;
    -fx-background-radius: 8;
    -fx-padding: 8 16;
}
```

5.2 Implementasi Bonus Heuristik

```
package model;

import java.util.HashSet;
import java.util.Set;

public class Heuristic {
    int actualHeight;
    int actualWidth;
    int exitRow;
    int exitCol;
    int mode;

    public Heuristic(int actualHeight, int actualWidth, int
exitRow, int exitCol, int mode) {
        this.actualHeight = actualHeight;
        this.actualWidth = actualWidth;
        this.exitRow = exitRow;
        this.exitCol = exitCol;
        this.mode = mode;
    }
```

```
    public int calculateBlockerDistanceHeuristic(Node node) {
        int minR = actualHeight, maxR = -1, minC = actualWidth,
maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (node.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        boolean horizontal = (minR == maxR);
        int length = horizontal ? (maxC - minC + 1) : (maxR -
minR + 1);
        int distance;
        Set<Character> blockers = new HashSet<>();

        if (horizontal) {
            if(exitRow != minR) {
                return Integer.MAX_VALUE;
            }

            if(exitCol == actualWidth) {
                distance = exitCol - maxC;
                for(int c = maxC + 1; c < exitCol; c++) {
                    char cell = node.getCell(minR, c);
                    if(cell != '.') {
                        blockers.add(cell);
                    }
                }
            }
            else {
                distance = minC - exitCol;
                for(int c = 0; c < minC; c++) {
                    char cell = node.getCell(minR, c);
                    if(cell != '.') {
                        blockers.add(cell);
```

```
                    }
                }
            }
        }
        else {
            if(exitCol != minC) {
                return Integer.MAX_VALUE;
            }

            if(exitRow == actualHeight) {
                distance = exitRow - maxR;
                for (int r = maxR + 1; r < exitRow; r++) {
                    char cell = node.getCell(r, minC);
                    if(cell != '.'){
                        blockers.add(cell);
                    }
                }
            }
            else {
                distance = minR - exitRow;
                for (int r = 0; r < minR; r++) {
                    char cell = node.getCell(r, minC);
                    if(cell != '.') {
                        blockers.add(cell);
                    }
                }
            }
        }

        return distance + blockers.size();
    }

    public int calculateDistanceHeuristic(Node node) {
        int minR = actualHeight, maxR = -1, minC = actualWidth,
maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (node.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
```

```
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        boolean horizontal = (minR == maxR);
        int length = horizontal ? (maxC - minC + 1) : (maxR -
minR + 1);
        int distance;

        if (horizontal) {
            if(exitRow ≠ minR) {
                return Integer.MAX_VALUE;
            }

            if(exitCol == actualWidth) {
                distance = exitCol - maxC;
            }
            else {
                distance = minC - exitCol;
            }
        }
        else {
            if(exitCol ≠ minC) {
                return Integer.MAX_VALUE;
            }

            if(exitRow == actualHeight) {
                distance = exitRow - maxR;
            }
            else {
                distance = minR - exitRow;
            }
        }

        return distance;
    }

    int calculateHeuristic(Node node) {
```

```
        if(mode == 0) {
            return calculateDistanceHeuristic(node);
        }
        return calculateBlockerDistanceHeuristic(node);
    }
}
```

5.3 Implementasi algoritma IDS

```
package model;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.Stack;

public class SolverIDS {
    char[][] startingGrid;
    HashMap<String, Integer> nodeToID;
    ArrayList<Node> nodes;
    int actualWidth;
    int actualHeight;
    int exitRow;
    int exitCol;
    int nodeID;
    int maxDepth;
    String filename;

    public SolverIDS(char[][] startingGrid, int actualWidth,
int actualHeight, int exitRow, int exitCol, String filename) {
        this.startingGrid = new char[startingGrid.length][];
        for (int i = 0; i < startingGrid.length; ++i) {
            this.startingGrid[i] =
```

```
Arrays.copyOf(startingGrid[i], startingGrid[i].length);
        }
        nodeToID = new HashMap<>();
        nodes = new ArrayList<>();
        this.actualWidth = actualWidth;
        this.actualHeight = actualHeight;
        this.exitRow = exitRow;
        this.exitCol = exitCol;
        this.filename = filename;
        nodeID = 0;
        maxDepth = 1000;
    }

    public boolean isSolved(Node b) {
        int minR = actualHeight, maxR = -1, minC = actualWidth,
maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (b.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        boolean horizontal = (minR == maxR);

        if(exitCol == actualWidth) {
            if(!horizontal || exitRow != minR) {
                return false;
            }

            for(int c = maxC + 1; c < actualWidth; ++c) {
                if(b.getCell(minR, c) != '.') {
                    return false;
                }
            }
            return true;
```

```
        }
        if(exitCol == -1) {
            if(!horizontal || exitRow != minR) {
                return false;
            }

            for(int c = minC - 1; c >= 0; --c) {
                if(b.getCell(minR, c) != '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitRow == actualHeight) {
            if(horizontal || exitCol != minC) {
                return false;
            }

            for(int r = maxR + 1; r < actualHeight; ++r) {
                if(b.getCell(r, minC) != '.') {
                    return false;
                }
            }
            return true;
        }
        if(exitRow == -1) {
            if(horizontal || exitCol != minC) {
                return false;
            }

            for(int r = minR - 1; r >= 0; --r) {
                if(b.getCell(r, minC) != '.') {
                    return false;
                }
            }
            return true;
        }

        return false;
    }
```

```
    public void showSolution(Node b) {
        ArrayList<Node> path = new ArrayList<>();
        Node currentNode = b;
        while(currentNode.getParentID() != -1) {
            path.add(currentNode);
            currentNode = nodes.get(currentNode.getParentID());
        }
        path.add(currentNode);

        System.out.println("Papan Awal");
        currentNode.printGrid();

        int moveCount = 0;

        Collections.reverse(path);
        for(Node node : path) {
            moveCount++;
            System.out.print("Gerakan " + moveCount + ": ");
            node.printMove();
            node.printGrid();
            System.out.println();
        }

        Node lastNode = path.get(path.size() - 1);

        int minR = actualHeight, maxR = -1, minC = actualWidth,
maxC = -1;
        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (lastNode.getCell(i, j) == 'P') {
                    minR = Math.min(minR, i);
                    maxR = Math.max(maxR, i);
                    minC = Math.min(minC, j);
                    maxC = Math.max(maxC, j);
                }
            }
        }

        if(exitCol == actualWidth) {
```

```
            int length = maxC - minC + 1;
            for(int c = minC; c ⩽ maxC; ++c) {
                lastNode.setCell(minR, c, '.');
            }
            for(int c = actualWidth - length; c < actualWidth;
++c) {
                lastNode.setCell(minR, c, 'P');
            }
        }
        else if(exitCol == -1) {
            int length = maxC - minC + 1;
            for(int c = minC; c ⩽ maxC; ++c) {
                lastNode.setCell(minR, c, '.');
            }
            for(int c = 0; c < length; ++c) {
                lastNode.setCell(minR, c, 'P');
            }
        }
        else if(exitRow == actualHeight) {
            int length = maxR - minR + 1;
            for(int r = minR; r ⩽ maxR; ++r) {
                lastNode.setCell(r, minC, '.');
            }
            for(int r = actualHeight - length; r <
actualHeight; ++r) {
                lastNode.setCell(r, minC, 'P');
            }
        }
        else if(exitRow == -1) {
            int length = maxR - minR + 1;
            for(int r = minR; r ⩽ maxR; ++r) {
                lastNode.setCell(r, minC, '.');
            }
            for(int r = 0; r < length; ++r) {
                lastNode.setCell(r, minC, 'P');
            }
        }

        moveCount++;
        System.out.println("Gerakan " + moveCount + ":
```

```
Keluar");
        lastNode.printGrid();
        System.out.println();
    }

    public void showSolutionToFile(Node b) {
        try (PrintWriter out = new PrintWriter(new
FileWriter(filename))) {
            ArrayList<Node> path = new ArrayList<>();
            Node currentNode = b;
            while (currentNode.getParentID() != -1) {
                path.add(currentNode);
                currentNode =
nodes.get(currentNode.getParentID());
            }
            path.add(currentNode);

            out.println("Total moves: " + (path.size() + 1));
            out.println();

            Collections.reverse(path);
            for (Node node : path) {
                node.printMove(out);
                node.printGrid(out);
                out.println();
            }

            Node lastNode = path.get(path.size() - 1);

            int minR = actualHeight, maxR = -1, minC =
actualWidth, maxC = -1;
            for (int i = 0; i < actualHeight; ++i) {
                for (int j = 0; j < actualWidth; ++j) {
                    if (lastNode.getCell(i, j) == 'P') {
                        minR = Math.min(minR, i);
                        maxR = Math.max(maxR, i);
                        minC = Math.min(minC, j);
                        maxC = Math.max(maxC, j);
                    }
                }
```

```
            }

            if (exitCol == actualWidth) {
                int length = maxC - minC + 1;
                for (int c = minC; c <= maxC; ++c) {
                    lastNode.setCell(minR, c, '.');
                }
                for (int c = actualWidth - length; c <
actualWidth; ++c) {
                    lastNode.setCell(minR, c, 'P');
                }
            } else if (exitCol == -1) {
                int length = maxC - minC + 1;
                for (int c = minC; c <= maxC; ++c) {
                    lastNode.setCell(minR, c, '.');
                }
                for (int c = 0; c < length; ++c) {
                    lastNode.setCell(minR, c, 'P');
                }
            } else if (exitRow == actualHeight) {
                int length = maxR - minR + 1;
                for (int r = minR; r <= maxR; ++r) {
                    lastNode.setCell(r, minC, '.');
                }
                for (int r = actualHeight - length; r <
actualHeight; ++r) {
                    lastNode.setCell(r, minC, 'P');
                }
            } else if (exitRow == -1) {
                int length = maxR - minR + 1;
                for (int r = minR; r <= maxR; ++r) {
                    lastNode.setCell(r, minC, '.');
                }
                for (int r = 0; r < length; ++r) {
                    lastNode.setCell(r, minC, 'P');
                }
            }

            lastNode.printMove(out);
            lastNode.printGrid(out);
```

```
            out.println();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void solve() {
        int solutionNodeID = -1;

        for (int depthLimit = 0; depthLimit ⩽ maxDepth;
depthLimit++) {
            System.out.println("Searching with depth limit: " +
depthLimit);

            nodeToID.clear();
            nodes.clear();
            nodeID = 0;

            Node startNode = new Node(startingGrid, 0, 0, 0,
-1, "");
            nodeToID.put(startNode.getStringGrid(), 0);
            nodes.add(startNode);

            solutionNodeID = depthLimitedDFS(startNode,
depthLimit);

            if (solutionNodeID ≠ -1) {
                System.out.println("Solution found at depth: "
+ depthLimit);
                break;
            }
        }

        if (solutionNodeID ≠ -1) {
            showSolution(nodes.get(solutionNodeID));
            showSolutionToFile(nodes.get(solutionNodeID));
        } else {
            System.out.println("No solution found within depth
limit " + maxDepth + ".");
```

```
        }
    }

    private int depthLimitedDFS(Node startNode, int depthLimit)
{
        Stack<Node> stack = new Stack<>();
        HashMap<String, Integer> visitedAtDepth = new
HashMap<>();

        stack.push(startNode);
        visitedAtDepth.put(startNode.getStringGrid(), 0);

        while (!stack.isEmpty()) {
            Node currentNode = stack.pop();

            if (isSolved(currentNode)) {
                return currentNode.getID();
            }

            if (currentNode.getG() ⩾ depthLimit) {
                continue;
            }

            expandNode(currentNode, stack, visitedAtDepth);
        }

        return -1;
    }

    private void expandNode(Node currentNode, Stack<Node>
stack, HashMap<String, Integer> visitedAtDepth) {
        boolean[] visited = new boolean[256];

        for(int i = 0; i < actualHeight; ++i) {
            for(int j = 0; j < actualWidth; ++j) {
                if (currentNode.getCell(i, j) == '.') {
                    continue;
                }

                char pieceID = currentNode.getCell(i, j);
```

```
                if (visited[pieceID]) {
                    continue;
                }
                visited[pieceID] = true;

                int vertical = currentNode.getPieceDirection(i,
j);
                int length = currentNode.getPieceLength(i, j);

                if(vertical == 1) {
                    for(int k = 0; k < length; ++k) {
                        currentNode.setCell(i + k, j, '.');
                    }

                    int left = i;
                    while(left >= 0 &&
currentNode.getCell(left, j) == '.') {
                        left--;
                    }
                    left++;
                    int right = i;
                    while(right + length - 1 < actualHeight &&
currentNode.getCell(right + length - 1, j) == '.') {
                        right++;
                    }
                    right--;

                    for(int k = right; k >= left; --k) {
                        if(k == i) {
                            continue;
                        }
                        if (currentNode.canPutPiece(k, j, 1,
length)) {
                            Node newNode = new
Node(currentNode);
                            for(int l = 0; l < length; ++l) {
                                newNode.setCell(k + l, j,
pieceID);
                            }
```

```
                            String newNodeKey =
newNode.getStringGrid();
                            int newDepth = currentNode.getG() +
1;

                            if
(visitedAtDepth.containsKey(newNodeKey) &&
visitedAtDepth.get(newNodeKey) ⩽ newDepth) {
                                continue;
                            }

                            visitedAtDepth.put(newNodeKey,
newDepth);
                            newNode.setG(newDepth);
                            nodeID++;
                            newNode.setID(nodeID);

newNode.setParentID(currentNode.getID());
                            if(k < i) {
                                newNode.setMove("Move " +
pieceID + " up " + (i - k));
                            }
                            else {
                                newNode.setMove("Move " +
pieceID + " down " + (k - i));
                            }

                            nodeToID.put(newNodeKey, nodeID);
                            nodes.add(newNode);

                            stack.push(newNode);
                        }
                    }

                    for(int k = 0; k < length; ++k) {
                        currentNode.setCell(i + k, j, pieceID);
                    }
                }
                else {
                    for(int k = 0; k < length; ++k) {
```

```
                        currentNode.setCell(i, j + k, '.');
                    }

                    int left = j;
                    while(left ⩾ 0 && currentNode.getCell(i,
left) == '.') {
                        left--;
                    }
                    left++;
                    int right = j;
                    while(right + length - 1 < actualWidth &&
currentNode.getCell(i, right + length - 1) == '.') {
                        right++;
                    }
                    right--;

                    for(int k = right; k ⩾ left; --k) {
                        if(k == j) {
                            continue;
                        }
                        if (currentNode.canPutPiece(i, k, 0,
length)) {
                            Node newNode = new
Node(currentNode);
                            for(int l = 0; l < length; ++l) {
                                newNode.setCell(i, k + l,
pieceID);
                            }

                            String newNodeKey =
newNode.getStringGrid();
                            int newDepth = currentNode.getG() +
1;

                            if
(visitedAtDepth.containsKey(newNodeKey) &&
visitedAtDepth.get(newNodeKey) ⩽ newDepth) {
                                continue;
                            }
```

```
                            visitedAtDepth.put(newNodeKey,
newDepth);
                            newNode.setG(newDepth);
                            nodeID++;
                            newNode.setID(nodeID);

newNode.setParentID(currentNode.getID());
                            if(k < j) {
                                newNode.setMove("Move " +
pieceID + " left " + (j - k));
                            }
                            else {
                                newNode.setMove("Move " +
pieceID + " right " + (k - j));
                            }

                            nodeToID.put(newNodeKey, nodeID);
                            nodes.add(newNode);

                            stack.push(newNode);
                        }
                    }

                    for(int k = 0; k < length; ++k) {
                        currentNode.setCell(i, j + k, pieceID);
                    }
                }
            }
        }
    }
}
```

LAMPIRAN

Berikut merupakan pranala repository GitHub tugas kecil ini.

https://github.com/BP04/Tucil3_13523059_13523067

Tabel Checklist:

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