Laporan Tugas Kecil 3
IF2211 Strategi Algoritma
Penyelesaian Puzzle Rush Hour Dengan Algoritma Path Finding

Disusun Oleh
13523065 - Dzaky Aurelia Fawwaz
13523073 - Alfian Hanif Fitria Yustanto

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024/2025

DAFTAR ISI

BAB 1
DESKRIPSI MASALAH DAN ALGORITMA

1.1. Puzzle Rush Hour

Gambar 1. Rush Hour Puzzle
(Sumber: https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582)

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

Komponen penting dari permainan Rush Hour terdiri dari:

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

2. **Piece –** *Piece* adalah sebuah kendaraan di dalam papan. Setiap *piece* memiliki *posisi*, *ukuran*, dan *orientasi*. *Orientasi* sebuah *piece* hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. *Piece* dapat memiliki beragam *ukuran*, yaitu jumlah

cell yang ditempati oleh *piece*. Secara standar, variasi *ukuran* sebuah *piece* adalah *2-piece* (menempati 2 *cell*) atau *3-piece* (menempati 3 *cell*). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

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

4. **Pintu Keluar –** *Pintu keluar* adalah tempat *primary piece* dapat digerakkan keluar untuk menyelesaikan permainan

5. **Gerakan** — *Gerakan* yang dimaksudkan adalah pergeseran *piece* di dalam permainan. *Piece* hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

1.2. Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search termasuk salah satu algoritma yang digunakan dalam pencarian rute dan termasuk ke dalam uninformed search. Algoritma yang termasuk ke dalam uninformed search adalah Breadth First Search, Depth First Search, Uniform Cost Search, Iterative Deepening Search, dan Depth Limited Search. Algoritma ini biasa dipakai dalam pencarian sebuah jalur dengan total bobot yang paling minimum. Dalam implementasinya, algoritma ini memanfaatkan priority queue untuk menyimpan simpul dengan bobotnya masing–masing, bobot ini menentukan urutan/prioritas. Simpul selanjutnya yang diperiksa diambil dari awal priority queue dan dihapus dari priority queue setelah diperiksa. Dalam pembangkitan simpul ekspan, nilai bobot dari simpul tetangganya bernilai bobot jarak dari simpul awal hingga simpul tetangga. Simpul-simpul dalam data yang diberikan akan terus ditelusuri berdasarkan priority queue yang dibuat sampai diperiksa simpul yang dicari.

Konsep utama pada algoritma Uniform Cost Search (UCS) adalah :

1. Priority Queue
   Priority Queue berfungsi untuk menyimpan simpul-simpul yang akan dikunjungi selanjutnya. Penggunaan priority queue memungkinkan simpul yang tersimpan terurut berdasarkan path cost paling rendah.
2. Path Cost
   Path cost merupakan cost yang diperlukan ketika memilih suatu simpul. Pada kasus puzzle rush hour path cost yang digunakan adalah biaya yang diperlukan ketika mengunjungi anak dari sebuah simpul.
3. Exploration
   Eksplorasi oleh UCS akan mengunjungi berdasarkan urutan priority queue dan akan terus mencari sampai tidak ada lagi kemungkinan pencarian atau mencapai simpu tujuan.

4. Termination
   Algoritma dihentikan ketika simpul tujuan berhasil dicapai atau tidak ada lagi simpul yang dapat dikunjungi

Pada penyelesaian Puzzle Rush Hour algoritma UCS dapat diimplementasikan dengan cara berikut :

1. Definisikan Board berukuran m x n dengan kondisi awal yang memiliki piece-piece pada lokasi tertentu di dalam board, sebuah piece primary, dan pintu keluar yang menjadi tujuan dari puzzle.
2. Buat suatu simpul yang berisi kondisi awal dari state board saat ini.
3. Cari semua kemungkinan gerakan piece yang dapat dilakukan berdasarkan kondisi awal. Kemungkinan gerakan dapat berupa gerakan horizontal atau vertikal dengan jarak tertentu selama tidak menyalahi aturan permainan puzzle rush hour.
4. Jadikan seluruh kemungkinan gerakan sebagai *children* dari node. Kemudian untuk setiap kemungkinan hitung nilai fungsi evaluasi. nilai fungsi evaluasi merupakan g(n) pada kasus ini yaitu path-cost , path cost berupa increment yang dimulai dari nilai 0 pada simpul akar dan akan bertambah pada *children.* kemudian masukkan simpul pada priority queue. lakukan kembali proses pada tahap ke-2 dan ke-3 dengan simpul yang terletak pada posisi pertama priority queue.
5. Terminasi terjadi ketika simpul tujuan telah ditemukan, atau tidak ada lagi simpul yang dapat dicari/priority queue kosong.

Pseudocode :

```
function UniformCostSearch(Board initial) -> Result
    startTime ← current system time

    open ← priority queue ordered by g(n)
    closed ← empty set
    nodesExpanded ← 0

    startNode ← Node(initial board, empty path, g = 0)
    insert startNode into open

    while open is not empty:
        current ← remove node with lowest g from open
        nodesExpanded ← nodesExpanded + 1

        if current.board is solved:
            endTime ← current system time
            result ← new Result
            result.moves ← current.path
            result.solvingTime ← endTime - startTime
            result.nodesExpanded ← nodesExpanded
            -> result

        stateKey ← string representation of current.board
        if stateKey in closed:
```

```
            continue
        add stateKey to closed

        for each child in generateChildren(current):
            if string representation of child.board in closed:
                continue

            gNew ← current.g + 1
            newPath ← copy of current.path
            append child.move to newPath

            newNode ← Node(child.board, newPath, gNew)
            insert newNode into open

    -> null  // No solution found
```

1.3. Algoritma Greedy Best First Search (GBFS)

Algoritma Greedy Best-First Search adalah salah satu algoritma informed search di mana fungsi evaluasinya adalah fungsi heuristik. Algoritma ini memanfaatkan representasi graf untuk menyelesaikan permasalahannya. Tujuan utamanya adalah bergerak dari simpul awal menuju simpul tujuan dengan langkah yang paling optimal melalui simpul-simpul yang paling sesuai menurut fungsi heurisitk yang didefinisikan. Pada umumnya, algoritma ini digunakan untuk menyelesaikan permasalahan penentuan rute. Meskipun demikian, domain permasalahan yang mampu diselesaikan dapat melampaui penentuan rute, misalnya untuk memecahkan puzzle.

Permasalahan yang ada dalam algoritma ini diantaranya:

1. Tidak ada jaminan optimum global

   Algoritma Greedy Best-First Search adalah algoritma yang mengutamakan optimum lokal, berarti ia akan menentukan keputusan hanya berdasarkan nilai fungsi heuristik saat itu tanpa mempertimbangkan perspektif global.
2. Tidak ada backtracking

   Pada umumnya, algoritma ini tidak memiliki kemampuan backtracking karena mengutamakan efisiensi komputasi program. Berarti, dia tidak akan mundur untuk menelusuri simpul lain ketika menemui jalan buntu. Hal ini membuat algoritma Greedy Best-First Search menjadi not complete, karena akan ada kasus di mana ia tidak bisa menemukan solusi padahal kenyataannya ada solusi.
3. Sensitif terhadap fungsi heuristik

   Efektivitas algoritma ini sangat bergantung kepada kualitas dan akurasi fungsi heuristik yang didefinisikan. Jika fungsi heuristiknya tidak diformulasikan dengan baik atau tidak mampu memenuhi domain permasalahan, algoritma Greedy Best-First Search bisa saja menyesatkan, mulai dari tidak optimalnya solusi hingga ditemukan solusi yang salah atau bahkan tidak ditemukan solusi.

Pada penyelesaian Puzzle Rush Hour algoritma GBFS dapat diimplementasikan dengan cara berikut :

1. Definisikan Board berukuran m x n dengan kondisi awal yang memiliki piece-piece pada lokasi tertentu di dalam board, sebuah piece primary, dan pintu keluar yang menjadi tujuan dari puzzle.
2. Buat suatu simpul yang berisi kondisi awal dari state board saat ini.
3. Cari semua kemungkinan gerakan piece yang dapat dilakukan berdasarkan kondisi awal. Kemungkinan gerakan dapat berupa gerakan horizontal atau vertikal dengan jarak tertentu selama tidak menyalahi aturan permainan puzzle rush hour.
4. Jadikan seluruh kemungkinan gerakan sebagai *children* dari node. Kemudian untuk setiap kemungkinan hitung nilai fungsi evaluasi. nilai fungsi evaluasi merupakan h(n) pada kasus ini merupakan fungsi heuristic yang digunakan. Fungsi heuristik akan memberikan estimasi nilai berdasarkan kondisi board. Jika proses terminasi belum tercapai, lakukan kembali proses pada tahap ke-2 dan ke-3 dengan simpul yang terletak pada posisi pertama priority queue.
5. Terminasi terjadi ketika simpul tujuan telah ditemukan, atau tidak ada lagi simpul yang dapat dicari/priority queue kosong.

Pseudocode :

```
function GreedyBestFirstSearch(initial, heuristic):
    startTime ← current system time

    open ← priority queue ordered by h(n)
    closed ← empty set
    nodesExpanded ← 0

    h ← heuristic estimate of initial state
    startNode ← Node(initial, empty path, h)
    insert startNode into open

    while open is not empty:
        current ← remove node with lowest h from open
        nodesExpanded ← nodesExpanded + 1

        if current.board is goal state:
            endTime ← current system time
            result ← new Result
            result.moves ← current.path
            result.solvingTime ← endTime - startTime
            result.nodesExpanded ← nodesExpanded
            -> result

        stateKey ← string representation of current.board
        if stateKey in closed:
            continue
        add stateKey to closed

        for each child in generateChildren(current):
```

```
            if string representation of child.board in closed:
                continue

            hNew ← heuristic estimate of child.board
            newPath ← copy of current.path
            append child.move to newPath

            newNode ← Node(child.board, newPath, hNew)
            insert newNode into open

    -> null  // No solution found
```

1.4. Algoritma A*

Algoritma A* (A-star) adalah salah satu algoritma route planning atau penentuan rute dengan adanya informasi tambahan mengenai tujuan pencarian (Informed Search). Algoritma A* adalah pengembangan dari algoritma Uniform Cost Search dan Greedy Best First Search, dengan ide untuk menghindari path (jalur) atau simpul yang sudah mahal selain menggunakan nilai heuristik. Pencarian dilakukan berdasarkan fungsi evaluasi (f(n)) suatu simpul. Fungsi evaluasi ini adalah penjumlahan dari ongkos untuk sampai ke suatu simpul ditambahkan dengan nilai heuristik simpul tersebut untuk sampai ke simpul tujuan. Pada algoritma A*, simpul dengan nilai fungsi evaluasi terbaik (terkecil dalam kasus pencarian ongkos minimum) akan dibangkitkan terlebih dahulu sehingga algoritma diimplementasikan menggunakan Priority Queue. Jika suatu fungsi heuristik yang dipakai dalam pencarian admissible, maka A* dijamin menghasilkan solusi yang optimal.

Pada penyelesaian Puzzle Rush Hour algoritma GBFS dapat diimplementasikan dengan cara berikut :

1. Definisikan Board berukuran m x n dengan kondisi awal yang memiliki piece-piece pada lokasi tertentu di dalam board, sebuah piece primary, dan pintu keluar yang menjadi tujuan dari puzzle.
2. Buat suatu simpul yang berisi kondisi awal dari state board saat ini.
3. Cari semua kemungkinan gerakan piece yang dapat dilakukan berdasarkan kondisi awal. Kemungkinan gerakan dapat berupa gerakan horizontal atau vertikal dengan jarak tertentu selama tidak menyalahi aturan permainan puzzle rush hour.
4. Jadikan seluruh kemungkinan gerakan sebagai *children* dari node. Kemudian untuk setiap kemungkinan hitung nilai fungsi evaluasi. nilai fungsi evaluasi merupakan h(n) pada kasus ini merupakan fungsi heuristic yang digunakan ditambah g(n) yaitu jarak simpul awal ke simpul saat ini. Fungsi heuristik akan memberikan estimasi nilai berdasarkan kondisi board. Jika proses terminasi belum tercapai, lakukan kembali proses pada tahap ke-2 dan ke-3 dengan simpul yang terletak pada posisi pertama priority queue.
5. Terminasi terjadi ketika simpul tujuan telah ditemukan, atau tidak ada lagi simpul yang dapat dicari/priority queue kosong.

Pseudocode :

```
function AStarSearch( Board: initial, Heuristic: heuristic) -> Result
    startTime ← current system time

    open ← priority queue ordered by f(n)
    closed ← empty set
    nodesExpanded ← 0

    initialPath ← empty list
    h ← heuristic estimate of initial state
    startNode ← Node(initial, initialPath, g=0, f=h)
    insert startNode into open

    while open is not empty:
        current ← remove node with lowest f from open
        nodesExpanded ← nodesExpanded + 1

        if current.board is goal state:
            endTime ← current system time
            result ← new Result
            result.moves ← current.path
            result.solvingTime ← endTime - startTime
            result.nodesExpanded ← nodesExpanded
            -> result

        stateKey ← string representation of current.board
        if stateKey in closed:
            continue
        add stateKey to closed

        for each child in generateChildren(current):
            if string representation of child.board in closed:
                continue

            gNew ← current.g + 1
            hNew ← heuristic estimate of child.board
            fNew ← gNew + hNew

            newPath ← copy of current.path
            append child.move to newPath

            newNode ← Node(child.board, newPath, gNew, fNew)
            insert newNode into open

    -> null  // No solution found
```

BAB 2
ANALISIS ALGORITMA

2.1. f(n) dan g(n)

Pada algoritma pathfinding terutama dikenal istilah f(n) dan g(n). f(n) merupakan fungsi evaluasi sedangkan g(n) merupakan fungsi sebenarnya. Fungsi ini menentukan nilai evaluasi suatu simpul untuk menentukan prioritasnya. Perumusan f(n) pada setiap algoritma dapat berbeda. Pada UCS nilai f(n) adalah g(n). g(n) merupakan biaya yang diperlukan dari simpul awal hingga simpul pencarian saat ini. Pada GBFS nilai f(n) merupakan h(n). h(n) adalah estimasi yang diberikan fungsi heuristic pada suatu simpul. Pada algoritma A* nilai f(n) dirumuskan dengan penggabungan antara g(n) dan h(n).

2.2. Heuristik A*

Pada algoritma A* untuk penyelesaian puzzle rush hour, digunakan dua heuristic yaitu :

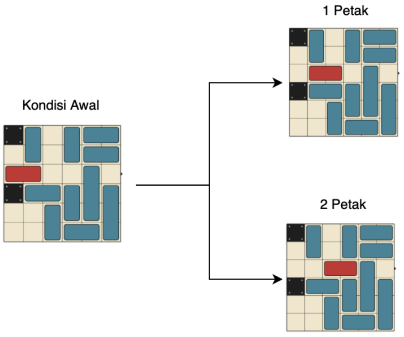
1. Shortest
   Heuristic ini mirip dengan Manhattan Heuristic. Fungsi ini akan menghitung selisih jarak row dan col antara primary piece dan exit.
2. Blocking
   Heuristic ini menghitung jumlah piece yang menghalangi primary piece menuju exit.

Suatu Heuristic dikatakan *admissible* jika nilai h(n) tidak lebih besar dari nilai aktual yang dibutuhkan. Nilai h(n) tidak boleh *overestimate* nilai dari biaya yang diperlukan untuk mencapai node tujuan. Hasil pencarian oleh algoritma A* bisa gagal jika heuristic yang digunakan tidak *admissible*.

Heuristic shortest hanya menghitung selisih jarak antara exit dengan primary piece sehingga nilai perhitungan akan selalu lebih kecil dari biaya sebenarnya. Begitu juga dengan Heuristik Blocking yang hanya menghitung jumlah piece yang menghalangi primary key menuju exit, nilai heuristiknya tidak akan melebihi nilai aktual dari biaya pencarian.

2.3. Analisis Algoritma UCS dan BFS

Algoritma UCS hanya memperhatikan nilai biaya sebenarnya dalam pemilihan simpul. Pada penyelesaian Puzzle Rush Hour, penulis mengasumsikan bahwa biaya setiap gerakan yang dilakukan adalah sama. contohnya, biaya yang diperlukan untuk menggerakkan sebuah piece horizontal ke arah kanan sebanyak 1 petak atau menggerakkan sebuah piece horizontal ke arah kanan sebanyak 2 petak memiliki biaya yang sama karena dianggap 1 gerakan.



gambar 2.3.1 (sumber : dokumentasi pribadi penulis)

Dua bentuk perpindahan piece tersebut diasumsikan hanya terjadi dalam 1 gerakan sehingga biaya yang diperlukan sama untuk setiap gerakan. Akibat dari asumsi bahwa setiap gerakan memiliki biaya yang sama, urutan prioritas pada implementasi priority queue akan terurut berdasarkan kemunculan *children* hal ini sangat serupa dengan algoritma BFS dimana pencarian berurutan berdasarkan level pada graf. Sehingga pada implementasi algoritma pada Puzzle Rush Hour hasil dari algoritma UCS akan serupa dengan algoritma BFS.

2.4. Perbandingan Algoritma A* dan UCS

Secara teori penyelesaian menggunakan algoritma A* akan lebih efisien dibandingkan dengan penyelesaian dengan algoritma UCS. Algoritma UCS hanya menghitung biaya aktual untuk menentukan prioritas simpul yang dipilih sedangkan algoritma A* menghitung biaya aktual dan heuristik untuk penentuan prioritas simpul. Jika heuristik yang digunakan pada algoritma A* *admissible* maka algoritma A* akan menghasilkan solusi optimal global. Dengan ini, Algoritma A* akan lebih efisien dalam penyelesaian puzzle rush hour.

2.5. Analisis Algoritma GBFS

Greedy Best-First Search (GBFS) adalah algoritma pencarian yang hanya mempertimbangkan nilai heuristik (h(n)) dalam menentukan urutan eksplorasi simpul. Algoritma ini selalu memilih simpul yang menurut prediksi heuristik paling dekat dengan tujuan, tanpa memperhitungkan biaya langkah yang sudah ditempuh (g(n)). Karena hanya mengandalkan tebakan heuristik, GBFS bisa saja membawa pencarian ke solusi yang tidak optimal. Hal ini menyebabkan GBFS tidak menjamin hasil solusi yang optimal secara global.

Dalam puzzle Rush Hour, heuristik seperti jarak horizontal primary piece ke pintu keluar mungkin tampak kecil, tetapi jalur tersebut bisa terhalang banyak kendaraan lain yang sulit digeser, sehingga solusi akhirnya memerlukan lebih banyak langkah dibanding jalur lain yang tampak lebih jauh. Dengan demikian, meskipun GBFS bisa menemukan solusi dengan cepat, hasilnya sering kali bukan solusi terbaik dalam hal jumlah langkah atau total biaya.

BAB 3
SOURCE CODE PROGRAM

3.1. Repository Github

Source Code lengkap dapat diakses melalui link github berikut:
https://github.com/WwzFwz/Tucil3_13523065_13523073

3.2. Abstract Class

3.2.1 Abstract Class Solver

```
public abstract class Solver {
   public abstract Result solve(Board initial);

   public Solver() {
   }

   protected static class Node {
       Board board;
       List<Movement> path;
       int g;
       int f;
       int h;

       Node(Board board, List<Movement> path, int g, int f) {
           this.board = board;
           this.path = path;
           this.g = g;
           this.f = f;
       }

       Node(Board board, List<Movement> path, int n, boolean t) {
           this.board = board;
           this.path = path;
           if (t) {
               this.h = n;
           } else {
               this.g = n;
           }

       }

   }

   protected static class ChildNode {
```

```
        Board board;
        Movement move;

        ChildNode(Board board, Movement move) {
            this.board = board;
            this.move = move;
        }
    }

    protected List<ChildNode> generateChildren(Node node) {
        List<ChildNode> children = new ArrayList<>();
        Board board = node.board;

        for (var piece : board.getPieces().values()) {
            for (String dir : piece.isHorizontal() ? new String[] { "L", "R" } :
new String[] { "U", "D" }) {
                for (int distance = 1; distance < Math.max(board.getRow(),
board.getCol()); distance++) {
                    Board newBoard = board.deepCopy();
                    Piece movedPiece = newBoard.getPieces().get(piece.getId());

                    if (canMove(newBoard, movedPiece, dir, distance)) {
                        Movement move = movedPiece.move(dir, distance);
                        children.add(new ChildNode(newBoard, move));
                    } else {

                        break;
                    }
                }
            }

        }

        return children;
    }

    protected boolean canMove(Board board, Piece piece, String direction, int
distance) {
        int r = piece.getRow();
        int c = piece.getCol();

        switch (direction) {
            case "L":
                if (!piece.isHorizontal())
```

```
                    return false;
                for (int step = 1; step <= distance; step++) {
                    if (c - step < 0 || !board.isCellEmpty(r, c - step))
                        return false;
                }
                break;
            case "R":
                if (!piece.isHorizontal())
                    return false;
                for (int step = 1; step <= distance; step++) {
                    int checkCol = c + piece.getLength() - 1 + step;
                    if (checkCol >= board.getCol() || !board.isCellEmpty(r,
checkCol))
                        return false;
                }
                break;
            case "U":
                if (piece.isHorizontal())
                    return false;
                for (int step = 1; step <= distance; step++) {
                    if (r - step < 0 || !board.isCellEmpty(r - step, c))
                        return false;
                }
                break;
            case "D":
                if (piece.isHorizontal())
                    return false;
                for (int step = 1; step <= distance; step++) {
                    int checkRow = r + piece.getLength() - 1 + step;
                    if (checkRow >= board.getRow() || !board.isCellEmpty(checkRow,
c))
                        return false;
                }
                break;
        }

        return true;
    }
}
```

3.2.1 Abstract Class Heuristic

```
public abstract class Heuristic {
    public Heuristic() {
```

```
    }

    public abstract int estimate(Board board);

}
```

3.3. Algorithm

3.3.1 AStarSolver.java

```
public class AStarSolver extends Solver {
    private Heuristic heuristic;

    public AStarSolver(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public Result solve(Board initial) {
        long startTime = System.currentTimeMillis();

        PriorityQueue<Node> open = new PriorityQueue<>(Comparator.comparingInt(n
-> n.f));
        Set<String> closed = new HashSet<>();
        int nodesExpanded = 0;

        List<Movement> initialPath = new ArrayList<>();
        int h = heuristic.estimate(initial);
        Node startNode = new Node(initial, initialPath, 0, h);
        open.add(startNode);

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

            if (current.board.isSolved()) {
                long endTime = System.currentTimeMillis();
                Result result = new Result();
                result.setMoves(current.path);
                result.setSolvingTime(endTime - startTime);
                result.setNodesExpanded(nodesExpanded);
                return result;
            }
```

```
            String stateKey = current.board.toString();
            if (closed.contains(stateKey))
                continue;
            closed.add(stateKey);

            for (var child : generateChildren(current)) {
                if (closed.contains(child.board.toString()))
                    continue;

                int gNew = current.g + 1;
                int hNew = heuristic.estimate(child.board);
                int fNew = gNew + hNew;

                List<Movement> newPath = new ArrayList<>(current.path);
                newPath.add(child.move);

                Node newNode = new Node(child.board, newPath, gNew, fNew);
                open.add(newNode);
            }
        }

        return null;
    }

}
```

3.3.2. GBFSSolver.java

```
public class GBFSSolver extends Solver {
    private Heuristic heuristic;

    public GBFSSolver(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public Result solve(Board initial) {
        long startTime = System.currentTimeMillis();

        PriorityQueue<Node> open = new PriorityQueue<>(Comparator.comparingInt(n
-> n.h));
        Set<String> closed = new HashSet<>();
```

```
        int nodesExpanded = 0;

        int h = heuristic.estimate(initial);
        Node startNode = new Node(initial, new ArrayList<>(), h, true);
        open.add(startNode);

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

            if (current.board.isSolved()) {
                long endTime = System.currentTimeMillis();
                Result result = new Result();
                result.setMoves(current.path);
                result.setSolvingTime(endTime - startTime);
                result.setNodesExpanded(nodesExpanded);
                return result;
            }

            String stateKey = current.board.toString();
            if (closed.contains(stateKey))
                continue;
            closed.add(stateKey);

            for (var child : generateChildren(current)) {
                if (closed.contains(child.board.toString()))
                    continue;

                int hNew = heuristic.estimate(child.board);
                List<Movement> newPath = new ArrayList<>(current.path);
                newPath.add(child.move);
                open.add(new Node(child.board, newPath, hNew, true));
            }
        }

        return null;
    }

}
```

3.4. Heuristic

3.4.1. BlockingHeuristic.java

```
public class BlockingHeuristic extends Heuristic {
    public BlockingHeuristic() {
    }

    @Override
    public int estimate(Board board) {
        Piece primary = board.getPrimaryPiece();
        Position exit = board.getExitPosition();
        if (primary == null || exit == null)
            return Integer.MAX_VALUE;

        char[][] grid = board.getBoardState();
        Set<Character> blockingIds = new HashSet<>();

        int pr = primary.getRow();
        int pc = primary.getCol();
        int len = primary.getLength();

        if (primary.isHorizontal()) {
            int row = pr;
            int startCol = (exit.getCol() > pc) ? pc + len : pc - 1;
            int endCol = exit.getCol();
            int step = (exit.getCol() > pc) ? 1 : -1;

            for (int c = startCol; c != endCol + step && c >= 0 && c <
board.getCol(); c += step) {
                char cell = grid[row][c];
                if (cell != '.' && cell != 'P') {
                    blockingIds.add(cell);
                }
            }
        } else {
            int col = pc;
            int startRow = (exit.getRow() > pr) ? pr + len : pr - 1;
            int endRow = exit.getRow();
            int step = (exit.getRow() > pr) ? 1 : -1;

            for (int r = startRow; r != endRow + step && r >= 0 && r <
board.getRow(); r += step) {
```

```
                char cell = grid[r][col];
                if (cell != '.' && cell != 'P') {
                    blockingIds.add(cell);
                }
            }
        }

        return blockingIds.size();
    }
}
```

3.4.2. ShortestHeuristic.java

```
public class ShortestHeuristic extends Heuristic {
    public ShortestHeuristic() {
    }

    @Override
    public int estimate(Board board) {
        Piece primary = board.getPieces().get("P");
        Position exit = board.getExitPosition();
        if (primary == null || exit == null)
            return Integer.MAX_VALUE;

        int r = primary.getRow();
        int c = primary.getCol();
        if (primary.isHorizontal()) {
            int frontCol = c + primary.getLength() - 1;
            if (r == exit.getRow() && exit.getCol() > frontCol)
                return exit.getCol() - frontCol - 1;

            if (r == exit.getRow() && exit.getCol() < c)
                return c - exit.getCol() - 1;

        } else {
            int frontRow = r + primary.getLength() - 1;
            if (c == exit.getCol() && exit.getRow() > frontRow)
                return exit.getRow() - frontRow - 1;

            if (c == exit.getCol() && exit.getRow() < r)
                return r - exit.getRow() - 1;
        }
```

```
        return Integer.MAX_VALUE;
    }


}
```

3.5 Game Component

3.5.1. Board.java

Board.java

```java
public class Board {
    private int rows;
    private int cols;
    private int countPiece;
    private Map<String, Piece> pieces;
    private Position exitPosition;



    public Board(int rows, int cols, int countPiece) {
        this.rows = rows;
        this.cols = cols;
        this.countPiece = countPiece;
        this.pieces = new HashMap<>();
    }

    public Board(Board other) {
        this.rows = other.rows;
        this.cols = other.cols;
        this.countPiece = other.countPiece;
        this.exitPosition = other.exitPosition != null ? new Position(other.exitPosition) : null;
        this.pieces = new HashMap<>();
        for (Map.Entry<String, Piece> e : other.pieces.entrySet()) {
            this.pieces.put(e.getKey(), new Piece(e.getValue()));
        }
    }

    public void addPiece(Piece p) {
        pieces.put(p.getId(), p);
    }


    public boolean isSolved() {
        Piece primary = pieces.get("P");
        if (primary == null || exitPosition == null)
            return false;

        int exitRow = exitPosition.getRow();
        int exitCol = exitPosition.getCol();
        int pr = primary.getRow();
        int pc = primary.getCol();
        int tailRow = pr;
        int tailCol = pc;

        if (primary.isHorizontal()) {
            tailCol = pc + primary.getLength();
            if (pr == exitRow && tailCol == exitCol) {
                System.err.println("1");
                return true;
            }
            if (pr == exitRow && exitCol == pc + 1) {
                System.err.println("2");
                return true;
            }

        } else {
            tailRow = pr + primary.getLength();
            if (pc == exitCol && tailRow == exitRow) {
                System.err.println("3");
                return true;
            }
            if (pc == exitCol && exitRow == pr) {
                System.err.println("4");
                return true;
            }
        }

        return false;
    }

    public boolean isCellEmpty(int r, int c) {
        for (Piece p : pieces.values()) {
            int rr = p.getRow();
            int cc = p.getCol();
            for (int i = 0; i < p.getLength(); i++) {
                int tr = rr + (!p.isHorizontal() ? i : 0);
                int tc = cc + (p.isHorizontal() ? i : 0);
                if (tr == r && tc == c)
                    return false;
            }
        }
        return true;
    }
```

Board.java

```java
public boolean canMove(Piece p, String direction, int distance) {
        if (distance <= 0)
            return false;

        int dr = 0, dc = 0;
        switch (direction) {
            case "R":
                dc = 1;
                break;
            case "L":
                dc = -1;
                break;
            case "U":
                dr = -1;
                break;
            case "D":
                dr = 1;
                break;
            default:
                return false;
        }

        if ((p.isHorizontal() && (direction.equals("U") || direction.equals("D"))) ||
                (!p.isHorizontal() && (direction.equals("L") || direction.equals("R")))) {
            return false;
        }

        int startRow = p.getRow();
        int startCol = p.getCol();

        for (int step = 1; step <= distance; step++) {
            int checkRow, checkCol;

            if (direction.equals("R")) {
                checkRow = startRow;
                checkCol = startCol + p.getLength() - 1 + step;
            } else if (direction.equals("L")) {
                checkRow = startRow;
                checkCol = startCol - step;
            } else if (direction.equals("U")) {
                checkRow = startRow - step;
                checkCol = startCol;
            } else { // D
                checkRow = startRow + p.getLength() - 1 + step;
                checkCol = startCol;
            }

            if (checkRow < 0 || checkRow >= rows || checkCol < 0 || checkCol >= cols)
                return false;

            if (!isCellEmpty(checkRow, checkCol)) {
                return false;
            }
        }

        return true;
    }

    public Movement movePiece(Piece p, String direction, int distance) {
        if (!canMove(p, direction, distance)) {
            throw new IllegalArgumentException("Invalid move");
        }
        return p.move(direction, distance);
    }

    public List<Movement> getAllPossibleMoves() {
        List<Movement> moves = new ArrayList<>();
        String[] directions = { "R", "L", "U", "D" };

        for (Piece p : pieces.values()) {
            for (String dir : directions) {
                for (int dist = 1; dist < Math.max(rows, cols); dist++) {
                    if (canMove(p, dir, dist)) {
                        moves.add(new Movement(p.getId(), dir, dist));
                    } else {
                        break;
                    }
                }
            }
        }
        return moves;
    }
```

3.5.2 Movement.java

Movement.java

```java
public class Movement {
    private String pieceId;
    private String direction; // "R", "L", "U", "D"
    private int distance; // pindah brp langkah

    public Movement(String pieceId, String direction, int distance) {
        this.pieceId = pieceId;
        this.direction = direction;
        this.distance = distance;
    }

    public String getDirection() {
        return direction;
    }

    public int getDistance() {
        return distance;
    }

    public String getPieceId() {
        return pieceId;
    }

    public void setDirection(String direction) {
        this.direction = direction;
    }

    public void setPieceId(String pieceId) {
        this.pieceId = pieceId;
    }

    public void setDistance(int distance) {
        this.distance = distance;
    }
    @Override
    public String toString() {
        String directionText = "";
        switch (direction) {
            case "R": directionText = "Right"; break;
            case "L": directionText = "Left"; break;
            case "U": directionText = "Up"; break;
            case "D": directionText = "Down"; break;
            default: directionText = direction;
        }

        return "Piece " + pieceId + " moves " + directionText + " " + distance +
            (distance > 1 ? " steps" : " step");
    }
}
```

3.5.3 Piece.java

Piece.java

```java
public class Piece {
    private String id;
    private final Position position;
    private int length;
    private boolean isHorizontal;
    private Color color;
    private boolean isPrimary;

    public Piece(String id, int row, int col, int length, boolean isHorizontal, boolean isPrimary) {
        this.id = id;
        this.position = new Position(row, col);
        this.length = length;
        this.isHorizontal = isHorizontal;
        this.isPrimary = isPrimary;

        if (isPrimary) {
            this.color = Color.PURPLE;
        } else {
            int hue = 0;
            if (id.length() > 0 && id.charAt(0) >= 'A' && id.charAt(0) <= 'Z') {
                int charValue = id.charAt(0) - 'A';
                hue = (charValue * 360 / 26) % 360;
            }
            this.color = Color.hsb(hue, 0.7, 0.8);
        }
    }

    public Piece(Piece other) {
        this.id = other.id;
        this.position = new Position(other.position);
        this.length = other.length;
        this.isHorizontal = other.isHorizontal;
        this.color = other.color;
        this.isPrimary = other.isPrimary;
    }

    public Movement move(String direction, int distance) {
        switch (direction) {
            case "R":
                if (isHorizontal) {
                    setCol(getCol() + distance);
                }
                break;
            case "L":
                if (isHorizontal) {
                    setCol(getCol() - distance);
                }
                break;
            case "U":
                if (!isHorizontal) {
                    setRow(getRow() - distance);
                }
                break;
            case "D":
                if (!isHorizontal) {
                    setRow(getRow() + distance);
                }
                break;
        }
        return new Movement(getId(), direction, distance);
    }
```

Piece.java

```java
public String getId() {
        return id;
    }

    public Color getColor(){
        return color;
    }
    public int getRow() {
        return position.getRow();
    }

    public void setRow(int row) {
        position.setRow(row);
    }

    public int getCol() {
        return position.getCol();
    }

    public void setCol(int col) {
        position.setCol(col);
    }

    public int getLength() {
        return length;
    }

    public Position getPosition() {
        return position;
    }

    public boolean isHorizontal() {
        return isHorizontal;
    }

    public boolean isPrimary() {
        return isPrimary;
    }
```

3.6 Game State

3.6.1. IO.java

```
package com.jawa.model.gameState;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.jawa.model.gameComponent.Board;
import com.jawa.model.gameComponent.Movement;
import com.jawa.model.gameComponent.Piece;
import com.jawa.model.gameComponent.Position;

public class IO {
   public static class InvalidConfigException extends Exception {
       public InvalidConfigException(String message) {
           super(message);
       }
   }

   public static Board loadFromFile(File file) throws IOException,
InvalidConfigException {
       try (BufferedReader br = new BufferedReader(new FileReader(file))) {

           String[] dim = br.readLine().trim().split("\\s+");
           if (dim.length != 2) {
               throw new InvalidConfigException("Error at line 1: Expected <row>
<col>");
           }

           int declaredRows, declaredCols;
           try {
               declaredRows = Integer.parseInt(dim[0]);
               declaredCols = Integer.parseInt(dim[1]);
           } catch (NumberFormatException e) {
```

```
                throw new InvalidConfigException("Error at line 1: Row and column
must be integers");
            }

            String secondLine = br.readLine();
            if (secondLine == null) {
                throw new InvalidConfigException("Error at line 2: Missing piece
count");
            }

            int pieceCount;
            try {
                pieceCount = Integer.parseInt(secondLine.trim());
            } catch (NumberFormatException e) {
                throw new InvalidConfigException("Error at line 2: Piece count
must be an integer");
            }

            List<String> allLines = new ArrayList<>();
            String line;
            while ((line = br.readLine()) != null) {
                allLines.add(line);
            }

            Position exitPos = null;
            int actualRows = allLines.size();
            int offsetX = 0, offsetY = 0;
            int countK = 0;

            outerLoop: for (int r = 0; r < actualRows; r++) {
                String currentLine = allLines.get(r);
                for (int c = 0; c < currentLine.length(); c++) {
                    if (currentLine.charAt(c) == 'K') {
                        countK++;
                        if (countK > 1) {
                            throw new InvalidConfigException(
                                    "Error at line " + r + " : Invalid K (more
than one K on config)");
                        }
                        if (r == 0) {
                            int exitRow = 0;
                            int exitCol = c;
                            offsetX = 0;
```

```
                        if (declaredCols + 1 <= currentLine.length()) {

                            if (c == 0) {
                                offsetX = 1;
                                System.out.println("ll");
                                exitCol = c + 1;

                            } else {
                                offsetX = 0;
                            }
                            offsetY = 0;
                        } else {
                            offsetY = 1;

                        }

                        exitPos = new Position(exitRow, exitCol);
                        System.out.println(exitRow + " " + exitCol);
                    } else if (c == 0) {
                        int exitCol = 1;
                        if (declaredCols + 1 <= currentLine.length()) {
                            offsetX = 1;
                            exitCol = 1;
                        } else {
                            offsetX = 0;
                            exitCol = c;
                        }
                        offsetY = 0;
                        int exitRow = r;
                        exitPos = new Position(exitRow, exitCol);
                        System.out.println(exitRow + " " + exitCol);
                    } else {
                        int exitRow = r;
                        int exitCol = c;
                        exitPos = new Position(exitRow, exitCol);
                        System.out.println(exitRow + " " + exitCol);
                    }
                    break outerLoop;
                }
            }
        }

        List<String> boardLines = new ArrayList<>();
        for (int i = offsetY; i < declaredRows + offsetY; i++) {
```

```
                String rawLine = i < allLines.size() ? allLines.get(i) : "";
                StringBuilder processedLine = new StringBuilder();
                for (int c = offsetX; c < declaredCols + offsetX; c++) {
                    processedLine.append(
                            (c < rawLine.length()) ? (rawLine.charAt(c) == ' ' ?
'.' : rawLine.charAt(c)) : '.');
                }
                boardLines.add(processedLine.toString());
            }

            Board board = new Board(declaredRows, declaredCols, pieceCount);
            if (exitPos != null)
                board.setExitPosition(exitPos);

            Map<Character, List<Position>> pieceMap = new HashMap<>();
            for (int r = 0; r < declaredRows; r++) {
                String boardRow = boardLines.get(r);
                for (int c = 0; c < declaredCols; c++) {
                    char ch = boardRow.charAt(c);
                    if (ch != '.' && ch != 'K') {
                        pieceMap.putIfAbsent(ch, new ArrayList<>());
                        pieceMap.get(ch).add(new Position(r, c));
                    }
                }
            }

            for (Map.Entry<Character, List<Position>> entry : pieceMap.entrySet())
{
                List<Position> positions = entry.getValue();

positions.sort(Comparator.comparingInt(Position::getRow).thenComparingInt(Positio
n::getCol));
                Position first = positions.get(0);
                boolean isHorizontal = positions.size() > 1 &&
                        positions.get(1).getRow() == first.getRow();

                Piece piece = new Piece(
                        String.valueOf(entry.getKey()),
                        first.getRow(), first.getCol(),
                        positions.size(),
                        isHorizontal,
                        entry.getKey() == 'P');
                board.addPiece(piece);
            }
```

```
            if (board.getPrimaryPiece() == null) {
                throw new InvalidConfigException("Invalid config : Primary Pieces
not found");
            }

            if (board.getPrimaryPiece().isHorizontal()) {
                if (board.getPrimaryPiece().getRow() != exitPos.getRow()) {
                    throw new InvalidConfigException("Invalid config : Primary
Piece and exit not alligned");
                }
            } else {
                if (board.getPrimaryPiece().getCol() != exitPos.getCol()) {
                    throw new InvalidConfigException("Invalid config : Primary
Piece and exit not alligned");
                }
            }

            if (board.getPieces().size() - 1 != pieceCount) {
                throw new InvalidConfigException("Invalid config : Pieces count
wrong");
            }

            return board;
        }
    }

    public static void saveResultToFile(Board initialBoard, Result result, File
file) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(file))) {

            Board currentBoard = new Board(initialBoard);

            Board finalBoard = new Board(initialBoard);
            List<Movement> moves = result.getMoves();
            for (Movement move : moves) {
                Piece piece = finalBoard.getPieces().get(move.getPieceId());
                piece.move(move.getDirection(), move.getDistance());
            }

            Piece primaryPiece = finalBoard.getPieces().get("P");
            Position exitPosition = determineExitPosition(finalBoard,
primaryPiece);
```

```
            writer.write("Puzzle Solution\n");
            writer.write("===============\n");
            writer.write(String.format("Solving Time: %d ms\n",
result.getSolvingTime()));
            writer.write(String.format("Nodes Expanded: %d\n",
result.getNodesExpanded()));
            writer.write(String.format("Total Moves: %d\n\n",
result.getMoves().size()));

            writer.write("Initial State\n");
            writer.write(boardToStringWithExit(currentBoard, exitPosition));
            writer.write("\n");

            for (int i = 0; i < moves.size(); i++) {
                Movement move = moves.get(i);
                Piece piece = currentBoard.getPieces().get(move.getPieceId());

                piece.move(move.getDirection(), move.getDistance());

                writer.write(String.format("Move %d: %s-%s %d\n",
                        i + 1,
                        move.getPieceId(),
                        getDirectionName(move.getDirection()),
                        move.getDistance()));
                writer.write(boardToStringWithExit(currentBoard, exitPosition));
                writer.write("\n");
            }
        }
    }

    private static Position determineExitPosition(Board solvedBoard, Piece
primaryPiece) {
        int pr = primaryPiece.getRow();
        int pc = primaryPiece.getCol();

        if (primaryPiece.isHorizontal()) {
            if (pc == 0) {
                return new Position(pr, -1);
            } else if (pc + primaryPiece.getLength() == solvedBoard.getCols()) {
                return new Position(pr, solvedBoard.getCols());
            }
        } else {
            if (pr == 0) {
                return new Position(-1, pc);
```

```
            } else if (pr + primaryPiece.getLength() == solvedBoard.getRows()) {
                return new Position(solvedBoard.getRows(), pc);
            }
        }

        return new Position(2, -1);
    }

    private static String getDirectionName(String direction) {
        switch (direction) {
            case "R":
                return "right";
            case "L":
                return "left";
            case "U":
                return "up";
            case "D":
                return "down";
            default:
                return direction;
        }
    }

    private static String boardToStringWithExit(Board board, Position exitPos) {
        int rows = board.getRows();
        int cols = board.getCols();

        char[][] extendedGrid = new char[rows + 2][cols + 2];
        for (int r = 0; r < rows + 2; r++) {
            for (int c = 0; c < cols + 2; c++) {
                extendedGrid[r][c] = ' ';
            }
        }

        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                extendedGrid[r + 1][c + 1] = '.';
            }
        }

        for (Piece piece : board.getPieces().values()) {
            String id = piece.getId();
            int row = piece.getRow();
            int col = piece.getCol();
```

```
            int length = piece.getLength();
            boolean isHorizontal = piece.isHorizontal();

            for (int i = 0; i < length; i++) {
                int r = row + (isHorizontal ? 0 : i);
                int c = col + (isHorizontal ? i : 0);

                if (r >= 0 && r < rows && c >= 0 && c < cols) {
                    extendedGrid[r + 1][c + 1] = id.charAt(0);
                }
            }
        }

        if (exitPos != null) {
            int exitRow = exitPos.getRow();
            int exitCol = exitPos.getCol();

            if (exitRow < 0) {
                extendedGrid[0][exitCol + 1] = 'K';
            } else if (exitRow >= rows) {
                extendedGrid[rows + 1][exitCol + 1] = 'K';
            } else if (exitCol < 0) {
                extendedGrid[exitRow + 1][0] = 'K';
            } else if (exitCol >= cols) {
                extendedGrid[exitRow + 1][cols + 1] = 'K';
            }
        } else {
            extendedGrid[3][0] = 'K';
        }
        StringBuilder result = new StringBuilder();
        for (int r = 0; r < rows + 2; r++) {
            for (int c = 0; c < cols + 2; c++) {
                result.append(extendedGrid[r][c]);
            }
            result.append('\n');
        }

        return result.toString();
    }
}
```

IO.java

```
List<String> boardLines = new ArrayList<>();
        for (int i = offsetY; i < declaredRows + offsetY; i++) {
            String rawLine = i < allLines.size() ? allLines.get(i) : "";
            StringBuilder processedLine = new StringBuilder();
            for (int c = offsetX; c < declaredCols + offsetX; c++) {
                if (c < rawLine.length()) {
                    char ch = rawLine.charAt(c);
                    processedLine.append(ch == ' ' ? '.' : ch);
                } else {
                    processedLine.append('.');
                }
            }
            boardLines.add(processedLine.toString());
        }

        Board board = new Board(declaredRows, declaredCols, pieceCount);
        if (exitPos != null) {
            board.setExitPosition(exitPos);
        }


        Map<Character, List<Position>> pieceMap = new HashMap<>();
        for (int r = 0; r < declaredRows; r++) {
            String boardRow = boardLines.get(r);
            for (int c = 0; c < declaredCols; c++) {
                char ch = boardRow.charAt(c);
                if (ch != '.' && ch != 'K') {
                    pieceMap.putIfAbsent(ch, new ArrayList<>());
                    pieceMap.get(ch).add(new Position(r, c));
                }
            }
        }

        for (Map.Entry<Character, List<Position>> entry : pieceMap.entrySet()) {
            List<Position> positions = entry.getValue();
            positions.sort(Comparator.comparingInt(Position::getRow).thenComparingInt(Position::getCol));

            Position first = positions.get(0);
            boolean isHorizontal = positions.size() > 1 &&
                    positions.get(1).getRow() == first.getRow();

            Piece piece = new Piece(
                    String.valueOf(entry.getKey()),
                    first.getRow(),
                    first.getCol(),
                    positions.size(),
                    isHorizontal,
                    entry.getKey() == 'P');
            board.addPiece(piece);
        }

        return board;
    }

    public static void saveResultToFile(Board initialBoard, Result result, File file) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(file))) {

            Board currentBoard = new Board(initialBoard);

            Board finalBoard = new Board(initialBoard);
            List<Movement> moves = result.getMoves();
            for (Movement move : moves) {
                Piece piece = finalBoard.getPieces().get(move.getPieceId());
                piece.move(move.getDirection(), move.getDistance());
            }

            Piece primaryPiece = finalBoard.getPieces().get("P");
            Position exitPosition = determineExitPosition(finalBoard, primaryPiece);

            writer.write("Puzzle Solution\n");
            writer.write("═══════════════\n");
            writer.write(String.format("Solving Time: %d ms\n", result.getSolvingTime()));
            writer.write(String.format("Nodes Expanded: %d\n", result.getNodesExpanded()));
            writer.write(String.format("Total Moves: %d\n\n", result.getMoves().size()));

            writer.write("Initial State\n");
            writer.write(boardToStringWithExit(currentBoard, exitPosition));
            writer.write("\n");

            for (int i = 0; i < moves.size(); i++) {
                Movement move = moves.get(i);
                Piece piece = currentBoard.getPieces().get(move.getPieceId());

                piece.move(move.getDirection(), move.getDistance());

                writer.write(String.format("Move %d: %s-%s %d\n",
                        i + 1,
                        move.getPieceId(),
                        getDirectionName(move.getDirection()),
                        move.getDistance()));
                writer.write(boardToStringWithExit(currentBoard, exitPosition));
                writer.write("\n");
            }
        }
    }

    private static Position determineExitPosition(Board solvedBoard, Piece primaryPiece) {
        int pr = primaryPiece.getRow();
        int pc = primaryPiece.getCol();

        if (primaryPiece.isHorizontal()) {
            if (pc == 0) {
                return new Position(pr, -1);
            } else if (pc + primaryPiece.getLength() == solvedBoard.getCols()) {
                return new Position(pr, solvedBoard.getCols());
            }
        } else {
            if (pr == 0) {
                return new Position(-1, pc);
            } else if (pr + primaryPiece.getLength() == solvedBoard.getRows()) {
                return new Position(solvedBoard.getRows(), pc);
            }
        }

        return new Position(2, -1);
    }
```

IO.java

```java
private static String getDirectionName(String direction) {
        switch (direction) {
            case "R":
                return "right";
            case "L":
                return "left";
            case "U":
                return "up";
            case "D":
                return "down";
            default:
                return direction;
        }
    }

    private static String boardToStringWithExit(Board board, Position exitPos) {
        int rows = board.getRows();
        int cols = board.getCols();

        char[][] extendedGrid = new char[rows + 2][cols + 2];
        for (int r = 0; r < rows + 2; r++) {
            for (int c = 0; c < cols + 2; c++) {
                extendedGrid[r][c] = ' ';
            }
        }

        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                extendedGrid[r + 1][c + 1] = '.';
            }
        }

        for (Piece piece : board.getPieces().values()) {
            String id = piece.getId();
            int row = piece.getRow();
            int col = piece.getCol();
            int length = piece.getLength();
            boolean isHorizontal = piece.isHorizontal();

            for (int i = 0; i < length; i++) {
                int r = row + (isHorizontal ? 0 : i);
                int c = col + (isHorizontal ? i : 0);

                if (r >= 0 && r < rows && c >= 0 && c < cols) {
                    extendedGrid[r + 1][c + 1] = id.charAt(0);
                }
            }
        }

        if (exitPos != null) {
            int exitRow = exitPos.getRow();
            int exitCol = exitPos.getCol();

            if (exitRow < 0) {
                extendedGrid[0][exitCol + 1] = 'K';
            } else if (exitRow >= rows) {
                extendedGrid[rows + 1][exitCol + 1] = 'K';
            } else if (exitCol < 0) {
                extendedGrid[exitRow + 1][0] = 'K';
            } else if (exitCol >= cols) {
                extendedGrid[exitRow + 1][cols + 1] = 'K';
            }
        } else {
            extendedGrid[3][0] = 'K';
        }
        StringBuilder result = new StringBuilder();
        for (int r = 0; r < rows + 2; r++) {
            for (int c = 0; c < cols + 2; c++) {
                result.append(extendedGrid[r][c]);
            }
            result.append('\n');
        }

        return result.toString();
    }
```

3.6.2. Result.java

Result.java

```java
public class Result {
    private List<Movement> moves;
    private long solvingTime;
    private int nodesExpanded;

    public Result() {
    }

    public Result(List<Movement> moves, long solvingTime, int nodesExpanded) {
        this.moves = moves;
        this.solvingTime = solvingTime;
        this.nodesExpanded = nodesExpanded;
    }

    // Getter dan Setter
    public List<Movement> getMoves() {
        return moves;
    }

    public void setMoves(List<Movement> moves) {
        this.moves = moves;
    }

    public long getSolvingTime() {
        return solvingTime;
    }

    public void setSolvingTime(long solvingTime) {
        this.solvingTime = solvingTime;
    }

    public int getNodesExpanded() {
        return nodesExpanded;
    }

    public void setNodesExpanded(int nodesExpanded) {
        this.nodesExpanded = nodesExpanded;
    }
    public List<Movement> getMovements(){
        return moves;
    }
}
```

3.6 Controller

3.6.1 MainController.java

MainController.java

```java
public class MainController {
    @FXML
    private Button uploadButton;

    @FXML
    private Label fileNameLabel;

    @FXML
    private ComboBox<String> algorithmComboBox;

    @FXML
    private ComboBox<String> heuristicComboBox;

    @FXML
    private Button solveButton;

    @FXML
    private Button backButton;

    @FXML
    private Button playPauseButton;

    @FXML
    private Button nextButton;

    @FXML
    private ListView<String> stepsListView;

    @FXML
    private Pagination stepsPagination;

    @FXML
    private Label statusLabel;

    @FXML
    private Pane boardPane;

    @FXML
    private Button saveButton;

    private boolean fileUploaded = false;
    private File selectedFile;
    private Board board;
    private Board originalBoard;
    private Result result;
    private final int stepsPerPage = 12;
    private Timeline playbackTimeline;
    private boolean isPlaying = false;
    private int currentStepIndex = -1;
```

MainController.java

```java
@FXML
    private void initialize() {
        initializeBoard(6, 6);

        algorithmComboBox.getItems().addAll(
                "A*",
                "Greedy Best-First",
                "Uniform-Cost Search");

        algorithmComboBox.valueProperty().addListener((observable, oldValue, newValue) -> {
            updateHeuristicOptions(newValue);
        });

        heuristicComboBox.valueProperty().addListener((observable, oldValue, newValue) -> {
            updateSolveButtonState();
        });

        stepsPagination.setPageCount(1);
        stepsPagination.setCurrentPageIndex(0);
        stepsPagination.setMaxPageIndicatorCount(5);

        stepsPagination.currentPageIndexProperty().addListener((obs, oldVal, newVal) -> {
            updateStepsListView(newVal.intValue());
        });

        saveButton.setDisable(true);
        stepsListView.getSelectionModel().selectedIndexProperty().addListener((obs, oldVal, newVal) -> {
            if (newVal != null && newVal.intValue() >= 0 && result != null && result.getMovements() != null) {
                int stepIndex = stepsPagination.getCurrentPageIndex() * stepsPerPage + newVal.intValue();
                if (stepIndex < result.getMovements().size()) {
                    showSolutionStep(stepIndex);
                }
            }
        });

    }

    private void initializeBoard(int rows, int cols) {
        boardPane.getChildren().clear();

        GridPane boardGrid = new GridPane();
        boardGrid.setHgap(2);
        boardGrid.setVgap(2);
        boardGrid.setPadding(new Insets(10));
        double cellSize = calculateCellSize(rows, cols);

        for (int row = 1; row < rows + 1; row++) {
            for (int col = 1; col < cols + 1; col++) {
                StackPane cell = createCell(row, col, cellSize);
                boardGrid.add(cell, col, row);
            }
        }

        boardPane.getChildren().add(boardGrid);

        boardGrid.layoutXProperty().bind(
                boardPane.widthProperty().subtract(boardGrid.widthProperty()).divide(2));
        boardGrid.layoutYProperty().bind(
                boardPane.heightProperty().subtract(boardGrid.heightProperty()).divide(2));
    }

    private void updateHeuristicOptions(String algorithm) {
        heuristicComboBox.getItems().clear();
        heuristicComboBox.setValue(null);

        solveButton.setDisable(true);

        if (!fileUploaded) {
            heuristicComboBox.setDisable(true);
            heuristicComboBox.setPromptText("Upload File First");
            return;
        }

        if (algorithm == null) {
            heuristicComboBox.setDisable(true);
            heuristicComboBox.setPromptText("Select Algorithm First");
            return;
        } else {
            solveButton.setDisable(false);
        }
        switch (algorithm) {
            case "A*":
                heuristicComboBox.setDisable(false);
                heuristicComboBox.setPromptText("Select Heuristic");
                heuristicComboBox.getItems().addAll(
                        "Shortest Distance",
                        "Blocking Pieces");
                break;

            case "Greedy Best-First":
                heuristicComboBox.setDisable(false);
                heuristicComboBox.getItems().addAll(
                        "Shortest Distance",
                        "Blocking Pieces");
                break;

            case "Uniform-Cost Search":
                heuristicComboBox.setDisable(true);
                heuristicComboBox.setPromptText("No Heuristic Needed");
                if (fileUploaded) {
                    solveButton.setDisable(false);
                }
                break;

            default:
                heuristicComboBox.setDisable(true);
                heuristicComboBox.setPromptText("Unknown Algorithm");
        }
    }
```

MainController.java

```java
private void updateSolveButtonState() {

        if (!fileUploaded) {
            solveButton.setDisable(true);
            return;
        }

        String algorithm = algorithmComboBox.getValue();
        if (algorithm == null) {
            solveButton.setDisable(true);
        } else {
            solveButton.setDisable(false);
        }
    }

    private void updateStepsListView(int pageIndex) {
        System.err.println("Updating ListView for page " + pageIndex);
        stepsListView.getItems().clear();

        if (result == null || result.getMovements().isEmpty()) {
            return;
        }

        int startIndex = pageIndex * stepsPerPage;
        int endIndex = Math.min(startIndex + stepsPerPage, result.getMovements().size());

        for (int i = startIndex; i < endIndex; i++) {
            Movement step = result.getMovements().get(i);
            stepsListView.getItems().add("Step " + (i + 1) + ": " + step.toString());
            System.err.println(step);
        }

        if (currentStepIndex >= startIndex && currentStepIndex < endIndex) {
            stepsListView.getSelectionModel().select(currentStepIndex - startIndex);
        }
    }

    @FXML
    private void handleUploadFile() {
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Open Puzzle File");
        fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Text Files", "*.txt"));

        selectedFile = fileChooser.showOpenDialog(uploadButton.getScene().getWindow());
        try {
            board = IO.loadFromFile(selectedFile);
        } catch (Exception e) {
            fileNameLabel.setText("Error Occured : " + e.getMessage());
        }
        initializeBoard(board.getRow(), board.getCol());
        originalBoard = board.deepCopy();

        displayBoard(board);
        if (board != null)
            displayExitGate(board.getExitPosition(), board.getCols(), board.getRows());

        fileUploaded = true;
        fileNameLabel.setText("File: " + selectedFile.getName());
        algorithmComboBox.setDisable(false);
        algorithmComboBox.setPromptText("Select Algorithm");
        updateHeuristicOptions(algorithmComboBox.getValue());
        solveButton.setDisable(true);
    }
```

MainController.java

```java
private void updateSolveButtonState() {

        if (!fileUploaded) {
            solveButton.setDisable(true);
            return;
        }

        String algorithm = algorithmComboBox.getValue();
        if (algorithm == null) {
            solveButton.setDisable(true);
        } else {
            solveButton.setDisable(false);
        }
    }

    private void updateStepsListView(int pageIndex) {
        System.err.println("Updating ListView for page " + pageIndex);
        stepsListView.getItems().clear();

        if (result == null || result.getMovements().isEmpty()) {
            return;
        }

        int startIndex = pageIndex * stepsPerPage;
        int endIndex = Math.min(startIndex + stepsPerPage, result.getMovements().size());

        for (int i = startIndex; i < endIndex; i++) {
            Movement step = result.getMovements().get(i);
            stepsListView.getItems().add("Step " + (i + 1) + ": " + step.toString());
            System.err.println(step);
        }

        if (currentStepIndex >= startIndex && currentStepIndex < endIndex) {
            stepsListView.getSelectionModel().select(currentStepIndex - startIndex);
        }
    }

    @FXML
    private void handleUploadFile() {
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Open Puzzle File");
        fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Text Files", "*.txt"));

        selectedFile = fileChooser.showOpenDialog(uploadButton.getScene().getWindow());
        try {
            board = IO.loadFromFile(selectedFile);
        } catch (Exception e) {
            fileNameLabel.setText("Error Occured : " + e.getMessage());
        }
        initializeBoard(board.getRow(), board.getCol());
        originalBoard = board.deepCopy();

        displayBoard(board);
        if (board != null)
            displayExitGate(board.getExitPosition(), board.getCols(), board.getRows());

        fileUploaded = true;
        fileNameLabel.setText("File: " + selectedFile.getName());
        algorithmComboBox.setDisable(false);
        algorithmComboBox.setPromptText("Select Algorithm");
        updateHeuristicOptions(algorithmComboBox.getValue());
        solveButton.setDisable(true);
    }
```

MainController.java

```java
@FXML
    private void handleBack() {
        if (currentStepIndex > 0) {
            showSolutionStep(currentStepIndex - 1);
        }
    }

    @FXML
    private void handleNext() {
        List<Movement> solutionSteps = result.getMovements();
        if (solutionSteps != null && currentStepIndex < solutionSteps.size() - 1) {
            showSolutionStep(currentStepIndex + 1);
        }
    }

    @FXML
    private void handlePlayPause() {
        List<Movement> solutionSteps = result.getMovements();
        if (solutionSteps == null || solutionSteps.isEmpty()) {
            return;
        }

        if (isPlaying) {

            playPauseButton.setText("▶");
            if (playbackTimeline != null) {
                playbackTimeline.stop();
            }
            isPlaying = false;
        } else {
            isPlaying = true;
            playPauseButton.setText("⏸");
            startPlayback();

        }
    }

    private void startPlayback() {
        if (playbackTimeline != null) {
            playbackTimeline.stop();
        }

        playbackTimeline = new Timeline(
                new KeyFrame(Duration.seconds(1), e -> {
                    List<Movement> movements = result.getMovements();
                    if (movements == null || movements.isEmpty()) {
                        playbackTimeline.stop();
                        isPlaying = false;
                        playPauseButton.setText("▶");
                        return;
                    }

                    if (currentStepIndex < movements.size() - 1) {
                        currentStepIndex++;
                        showSolutionStep(currentStepIndex);
                    } else {
                        playbackTimeline.stop();
                        isPlaying = false;
                        playPauseButton.setText("▶");
                    }
                }));

        playbackTimeline.setCycleCount(Timeline.INDEFINITE);
        playbackTimeline.play();
        isPlaying = true;
        playPauseButton.setText("⏸");
    }

    private StackPane createCell(int row, int col, double size) {
        Rectangle cellBg = new Rectangle(size, size);
        cellBg.setFill(Color.web("#1a1158"));
        cellBg.setStroke(Color.web("#4287f5"));
        cellBg.setStrokeWidth(1.5);
        StackPane cell = new StackPane(cellBg);
        cell.setId("cell_" + row + "_" + col);
        return cell;
    }
```

MainController.java

```java
private double calculateCellSize(int rows, int cols) {
        double availableWidth = boardPane.getWidth();
        double availableHeight = boardPane.getHeight();
        if (availableWidth <= 0)
            availableWidth = boardPane.getPrefWidth();
        if (availableHeight <= 0)
            availableHeight = boardPane.getPrefHeight();

        if (availableWidth <= 0)
            availableWidth = 400;
        if (availableHeight <= 0)
            availableHeight = 400;

        availableWidth -= 40;
        availableHeight -= 40;

        double maxCellWidth = availableWidth / cols;
        double maxCellHeight = availableHeight / rows;

        double cellSize = Math.min(maxCellWidth, maxCellHeight);

        if (cellSize > 60)
            cellSize = 60;

        if (cellSize < 20)
            cellSize = 20;
        return cellSize;
    }

    private void displayBoard(Board board) {
        GridPane boardGrid = (GridPane) boardPane.getChildren().get(0);
        clearPieces(boardGrid);
        double cellSize = calculateCellSize(board.getRows(), board.getCols());
        Map<String, Piece> pieces = board.getPieces();
        for (Piece piece : pieces.values()) {
            displaySinglePiece(boardGrid, piece, cellSize);
        }
        displayExitGate(board.getExitPosition(), board.getCols(), board.getRows());

    }

    private void clearPieces(GridPane boardGrid) {
        boardGrid.getChildren().removeIf(node -> !(node instanceof StackPane) ||
                !((StackPane) node).getId().startsWith("cell_"));
    }
```

MainController.java

```java
private void displayExitGate(Position exitPos, int rows, int cols) {
        try {
            if (exitPos == null)
                return;

            if (boardPane.getChildren().isEmpty()) {
                System.err.println("Warning: boardPane is empty. Cannot display exit gate.");
                return;
            }

            GridPane boardGrid = (GridPane) boardPane.getChildren().get(0);

            double cellSize = calculateCellSize(rows, cols);

            Piece primaryPiece = board.getPieces().get("P");
            if (primaryPiece == null) {
                System.err.println("Warning: Primary piece 'P' not found. Cannot determine exit gate position.");
                return;
            }

            int exitRow = exitPos.getRow();
            int exitCol = exitPos.getCol();
            int pr = primaryPiece.getRow();
            int pc = primaryPiece.getCol();

            int gridRow, gridCol;
            if (primaryPiece.isHorizontal()) {
                int tailCol = pc + primaryPiece.getLength();

                if (pr == exitRow && tailCol == exitCol) {
                    gridRow = pr + 1;
                    gridCol = tailCol + 1;
                }
                else if (pr == exitRow && exitCol <= 1) {
                    gridRow = pr + 1;
                    gridCol = 0;
                } else {
                    gridRow = pr + 1;
                    gridCol = cols + 1;
                }
            } else {
                int tailRow = pr + primaryPiece.getLength();
                if (pc == exitCol && tailRow == exitRow) {
                    gridRow = tailRow + 1;
                    gridCol = pc + 1;
                }

                else if (pc == exitCol && exitRow == 0) {
                    gridRow = 0;
                    gridCol = pc + 1;
                } else {

                    gridRow = rows + 1;
                    gridCol = pc + 1;
                }
            }

            Rectangle exitRect = new Rectangle(cellSize - 4, cellSize - 4);
            exitRect.setFill(Color.RED);
            exitRect.setStroke(Color.BLACK);
            exitRect.setStrokeWidth(2);
            exitRect.setArcWidth(10);
            exitRect.setArcHeight(10);

            Label exitLabel = new Label("EXIT");
            exitLabel.setTextFill(Color.BLACK);
            exitLabel.setFont(Font.font("System", FontWeight.BOLD, 14));

            StackPane exitContainer = new StackPane();
            exitContainer.getChildren().addAll(exitRect, exitLabel);
            exitContainer.setId("exit_gate");

            boardGrid.add(exitContainer, gridCol, gridRow);
        } catch (Exception e) {
            System.err.println("Error displaying exit gate: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

MainController.java

```java
private void displaySinglePiece(GridPane boardGrid, Piece piece, double cellSize) {
        Rectangle pieceRect = new Rectangle();

        if (piece.isHorizontal()) {
            pieceRect.setWidth(cellSize * piece.getLength() - 4);
            pieceRect.setHeight(cellSize - 4);
        } else {
            pieceRect.setWidth(cellSize - 4);
            pieceRect.setHeight(cellSize * piece.getLength() - 4);
        }

        pieceRect.setFill(piece.getColor());
        pieceRect.setStroke(Color.BLACK);
        pieceRect.setStrokeWidth(2);
        pieceRect.setArcWidth(10);
        pieceRect.setArcHeight(10);

        pieceRect.setOnMouseEntered(e -> pieceRect.setOpacity(0.8));
        pieceRect.setOnMouseExited(e -> pieceRect.setOpacity(1.0));

        StackPane pieceContainer = new StackPane();
        pieceContainer.getChildren().add(pieceRect);

        Label idLabel = new Label(piece.getId());
        idLabel.setTextFill(Color.BLACK);
        idLabel.setFont(Font.font("System", FontWeight.BOLD, 14));
        pieceContainer.getChildren().add(idLabel);

        pieceContainer.setId(piece.getId());

        Position pos = piece.getPosition();
        if (piece.isHorizontal()) {
            GridPane.setColumnSpan(pieceContainer, piece.getLength());
            boardGrid.add(pieceContainer, pos.getCol() + 1, pos.getRow() + 1);
        } else {
            GridPane.setRowSpan(pieceContainer, piece.getLength());
            boardGrid.add(pieceContainer, pos.getCol() + 1, pos.getRow() + 1);
        }
    }


    private void showSolutionStep(int stepIndex) {
        List<Movement> solutionSteps = result.getMovements();
        if (solutionSteps == null || stepIndex < 0 || stepIndex >= solutionSteps.size()) {
            return;
        }


        board = originalBoard.deepCopy();
        for (int i = 0; i <= stepIndex; i++) {
            Movement move = solutionSteps.get(i);
            Piece piece = board.getPieces().get(move.getPieceId());
            if (piece != null) {
                piece.move(move.getDirection(), move.getDistance());
            }
        }
        displayBoard(board);

        currentStepIndex = stepIndex;

        int pageIndex = stepIndex / stepsPerPage;
        if (pageIndex != stepsPagination.getCurrentPageIndex()) {
            stepsPagination.setCurrentPageIndex(pageIndex);
        } else {
            int listIndex = stepIndex % stepsPerPage;
            stepsListView.getSelectionModel().select(listIndex);
        }

        backButton.setDisable(stepIndex == 0);
        nextButton.setDisable(stepIndex == solutionSteps.size() - 1);

    }
```

MainController.java

```
@FXML
    private void handleSaveSolution() {
        if (result == null) {
            showAlert(Alert.AlertType.ERROR, "Error", "No solution available to save.");
            return;
        }

        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Save Solution");
        fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Text Files", "*.txt"));

        String initialFileName = "solution.txt";
        fileChooser.setInitialFileName(initialFileName);

        File file = fileChooser.showSaveDialog(saveButton.getScene().getWindow());
        if (file != null) {
            try {
                IO.saveResultToFile(originalBoard, result, file);

                showAlert(Alert.AlertType.INFORMATION, "Success",
                        "Solution saved successfully to: " + file.getAbsolutePath());
            } catch (IOException e) {
                showAlert(Alert.AlertType.ERROR, "Error",
                        "Failed to save solution: " + e.getMessage());
                e.printStackTrace();
            }
        }
    }

    // Add a helper method for showing alerts
    private void showAlert(Alert.AlertType type, String title, String content) {
        Alert alert = new Alert(type);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(content);
        alert.showAndWait();
    }
```

3.7 Lainnya

3.7.1 Primary.fxml

primary.fxml

```
<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.ComboBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.ListView?>
<?import javafx.scene.control.Pagination?>
<?import javafx.scene.control.ScrollPane?>
<?import javafx.scene.control.Separator?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Font?>

<BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="600.0" minWidth="800.0" prefHeight="700.0"
prefWidth="1000.0"  xmlns="http://javafx.com/javafx/17" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.jawa.MainController">
   <left>
      <VBox styleClass="control-panel" BorderPane.alignment="CENTER">
         <children>
            <HBox alignment="CENTER" maxWidth="1.7976931348623157E308">
               <children>
                  <Label styleClass="title-label" text="Control Panel">
                     <VBox.margin>
                        <Insets bottom="15.0" />
                     </VBox.margin>
                  </Label>
               </children>
            </HBox>

            <Separator>
               <VBox.margin>
                  <Insets bottom="10.0" top="10.0" />
               </VBox.margin>
            </Separator>
            <Button fx:id="uploadButton" maxWidth="1.7976931348623157E308" mnemonicParsing="false"
onAction="#handleUploadFile" text="Select File">
               <VBox.margin>
                  <Insets bottom="10.0" top="5.0" />
               </VBox.margin>
            </Button>
            <HBox alignment="CENTER">
               <children>
                  <Label fx:id="fileNameLabel" text="No file selected" />
               </children>
            </HBox>

            <Separator>
               <VBox.margin>
                  <Insets bottom="10.0" top="10.0" />
               </VBox.margin>
            </Separator>
            <ComboBox fx:id="algorithmComboBox" maxWidth="1.7976931348623157E308" promptText="Select Algorithm" disable =
"true">
               <VBox.margin>
                  <Insets bottom="10.0" top="5.0" />
               </VBox.margin>
            </ComboBox>
               <ComboBox fx:id="heuristicComboBox" maxWidth="1.7976931348623157E308" promptText="Select Heuristic" disable=
"true">
                  <VBox.margin>
                     <Insets bottom="10.0" top="5.0" />
                  </VBox.margin>
               </ComboBox>
            <Button fx:id="solveButton" maxWidth="1.7976931348623157E308" mnemonicParsing="false" onAction="#handleSolve"
styleClass="solve-button" text="Solve Puzzle" disable = "true">
               <VBox.margin>
                  <Insets bottom="15.0" top="5.0" />
               </VBox.margin>
            </Button>
            <Button fx:id="saveButton" maxWidth="1.7976931348623157E308" mnemonicParsing="false"
onAction="#handleSaveSolution" styleClass="save-button" text="Save Solution" disable="true">
               <VBox.margin>
                  <Insets bottom="10.0" top="5.0" />
               </VBox.margin>
            </Button>
            <Separator>
               <VBox.margin>
                  <Insets bottom="10.0" />
               </VBox.margin>
            </Separator>

            <HBox alignment="CENTER" spacing="10.0">
               <children>
                  <Button fx:id="backButton" mnemonicParsing="false" onAction="#handleBack" text="⏮" disable="true" />
                  <Button fx:id="playPauseButton" mnemonicParsing="false" onAction="#handlePlayPause" text="▶" disable
="true" />
                  <Button fx:id="nextButton" mnemonicParsing="false" onAction="#handleNext" text="⏭" disable= "true" />
               </children>
               <VBox.margin>
                  <Insets bottom="10.0" top="5.0" />
               </VBox.margin>
            </HBox>
```

primary.fxml

```
<HBox alignment="CENTER">
               <children>
                  <Label text="Solution Steps" />
               </children>
            </HBox>
            <ScrollPane fitToHeight="true" fitToWidth="true" VBox.vgrow="ALWAYS">
               <content>
                  <VBox>
                     <children>
                        <ListView fx:id="stepsListView" VBox.vgrow="ALWAYS" />
                     </children>
                  </VBox>
               </content>
               <VBox.margin>
                  <Insets bottom="3.0" top="5.0" />
               </VBox.margin>
            </ScrollPane>

            <Pagination fx:id="stepsPagination" maxPageIndicatorCount="5" pageCount="5" />

            <Label fx:id="statusLabel" text="Ready" textFill="#006400">
               <VBox.margin>
                  <Insets top="5.0" />
               </VBox.margin>
            </Label>
         </children>
         <padding>
            <Insets bottom="15.0" left="15.0" right="15.0" top="15.0" />
         </padding>
      </VBox>
   </left>
   <center>
      <VBox alignment="CENTER" BorderPane.alignment="CENTER">
         <children>
            <Label styleClass="board-title" text="Game Board">
               <font>
                  <Font name="System Bold" size="16.0" />
               </font>
               <VBox.margin>
                  <Insets bottom="10.0" top="20.0" />
               </VBox.margin>
            </Label>
            <Pane fx:id="boardPane" minHeight="300.0" minWidth="300.0"  maxHeight="1.7976931348623157E308"
maxWidth="1.7976931348623157E308" styleClass="board-pane" VBox.vgrow="ALWAYS">
               <VBox.margin>
                  <Insets bottom="20.0" left="20.0" right="20.0" top="10.0" />
               </VBox.margin>
            </Pane>
         </children>
      </VBox>
   </center>
</BorderPane>
```

3.7.2 Style.css

styles.css

```
.root {
    -fx-primary: #4287f5;
    -fx-secondary: #5a9cf8;
    -fx-primarytext: #FFFFFF;
    -fx-light: #a0c6ff;
    -fx-dark: #2257a7;
    -fx-background: #0a033d;
    -fx-disabled-text: #6b6b6b;
    -fx-font-family: "System";
    -fx-background-color: -fx-background;
}


.control-panel {
    -fx-background-color: #0d0545;
    -fx-padding: 10;
    -fx-spacing: 10;
    -fx-min-width: 250px;
    -fx-pref-width: 300px;
    -fx-max-width: 350px;
    -fx-border-color: #2a1d7a;
    -fx-border-width: 0 1 0 0;
}


.title-label {
    -fx-font-size: 18px;
    -fx-font-weight: bold;
    -fx-text-fill: #ffffff;
    -fx-alignment: CENTER;
}

.board-title {
    -fx-font-size: 16px;
    -fx-font-weight: bold;
    -fx-text-fill: #ffffff;
}


Label {
    -fx-font-size: 12px;
    -fx-font-weight: bold;
    -fx-text-fill: #c0c0c0;
}


#statusLabel {
    -fx-text-fill: #4caf50;
    -fx-font-size: 14px;
    -fx-padding: 5px;
    -fx-background-color: rgba(26, 17, 88, 0.7);
    -fx-background-radius: 3px;
}


.button {
    -fx-background-color: -fx-primary;
    -fx-text-fill: -fx-primarytext;
    -fx-font-weight: bold;
    -fx-cursor: hand;
}

.button:hover {
    -fx-background-color: -fx-secondary;
    -fx-effect: dropshadow(three-pass-box, rgba(255,255,255,0.2), 3, 0, 0, 0);
}

.solve-button {
    -fx-background-color: #5e35b1;
    -fx-font-size: 14px;
}

.solve-button:hover {
    -fx-background-color: #7c4dff;
}


.combo-box {
    -fx-background-color: #1a1158;
    -fx-border-color: -fx-primary;
    -fx-border-radius: 3;
    -fx-text-fill: white;
}

.combo-box .list-cell {
    -fx-background-color: #1a1158;
    -fx-text-fill: white;
}

.combo-box .list-cell:hover {
    -fx-background-color: -fx-primary;
}
```

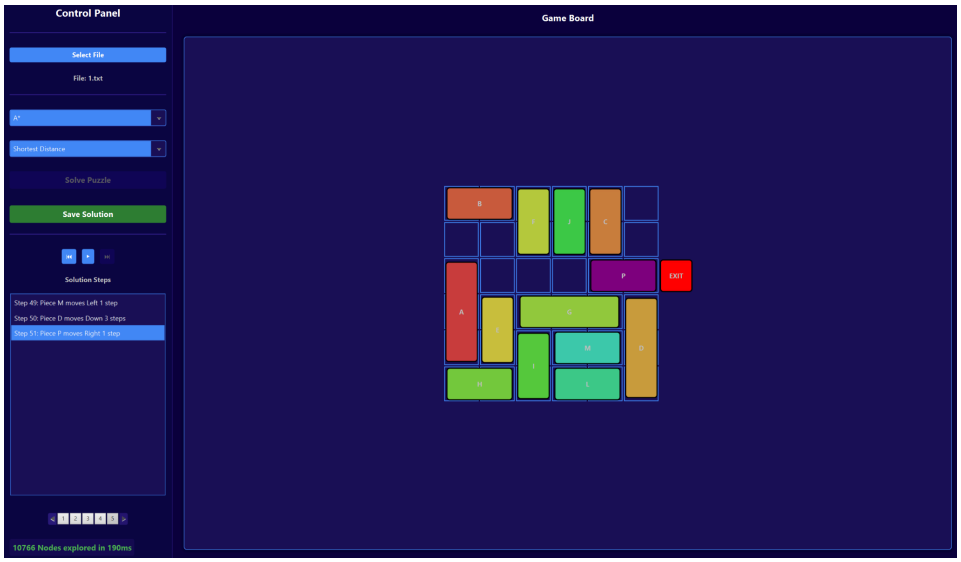
3.7.3 module-info.java

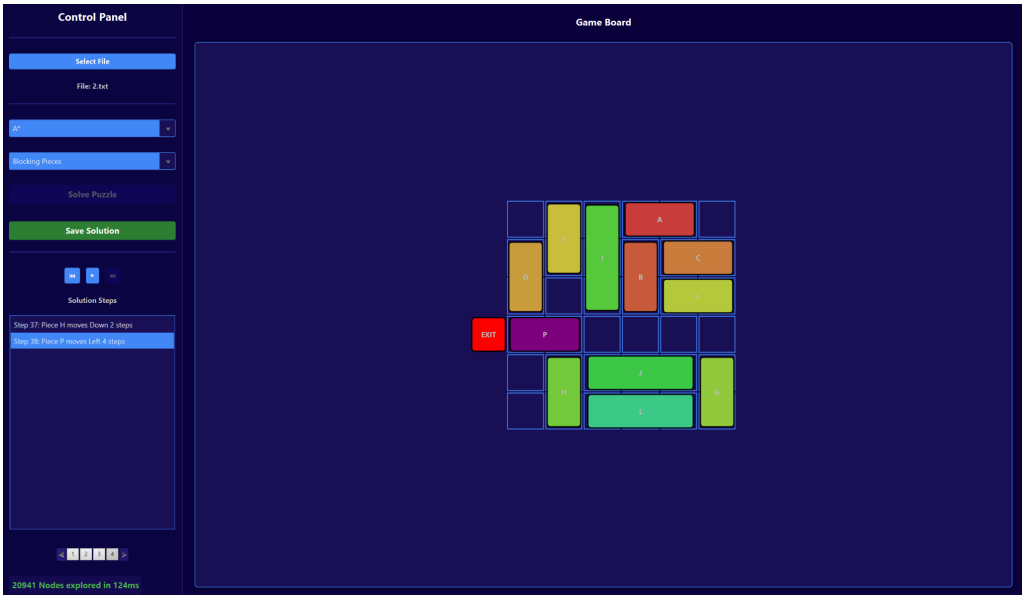
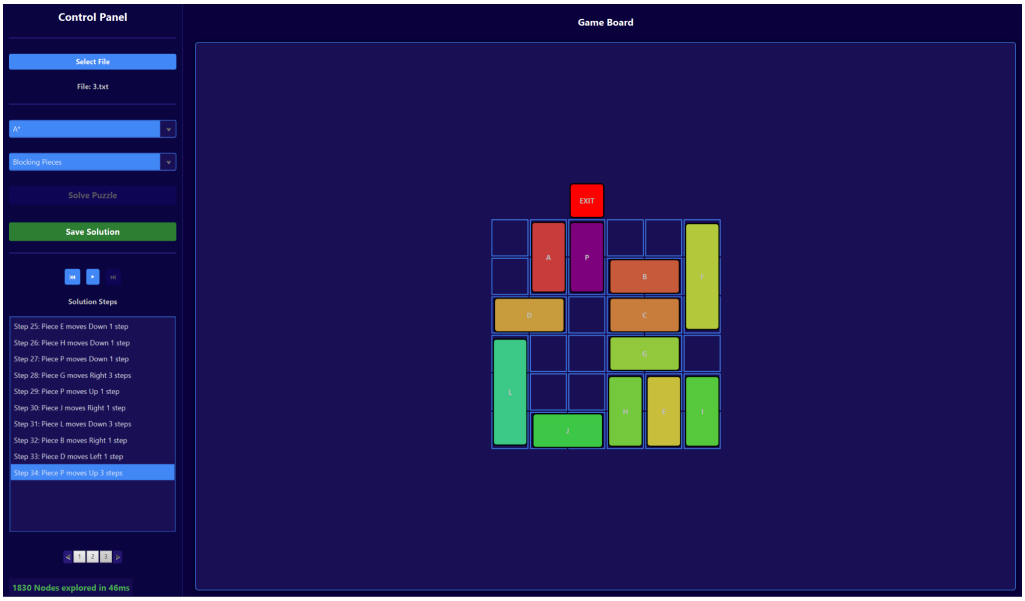
Informasi module untuk memudahkan import satu sama lain

module-info.java

```
module com.jawa {
    requires javafx.controls;
    requires javafx.fxml;

    opens com.jawa to javafx.fxml;
    exports com.jawa;

    opens com.jawa.model.algorithm to javafx.fxml;
    exports com.jawa.model.algorithm;

    opens com.jawa.model.gameComponent to javafx.fxml;
    exports com.jawa.model.gameComponent;

    opens com.jawa.model.gameState to javafx.fxml;
    exports com.jawa.model.gameState;
}
```

BAB 5
UJI COBA PROGRAM
5.1. Test Case A*

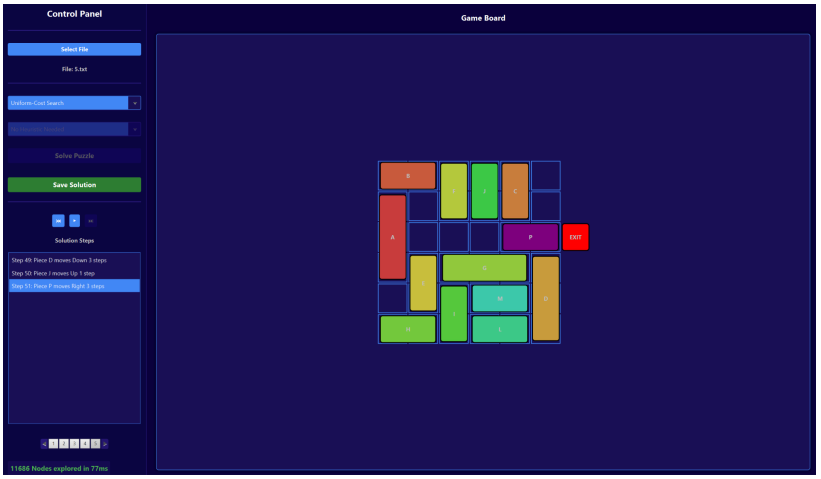
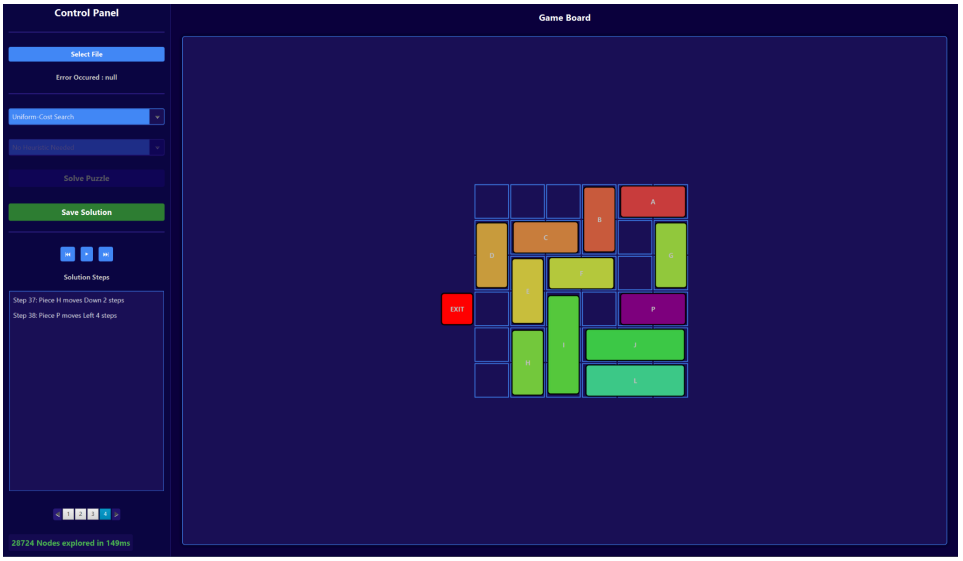
<table>
<tr><th colspan="2">Test Case #1 A* - Shortest</th></tr>
<tr><td>Input</td><td>6 6
12
ABB.C.
AEF.CD
AEFPPDK
GGGJ.D
..IJMM
HHILL.</td></tr>
<tr><td>Output</td><td> 
</td></tr>
<tr><th colspan="2">Test Case #2 A* - Shortest</th></tr>
<tr><td>Input</td><td>6 6
11
...BAA
DCCB.G
DEFF.G
K.EI.PP
.HIJJJ
.HILLL</td></tr>
</table>

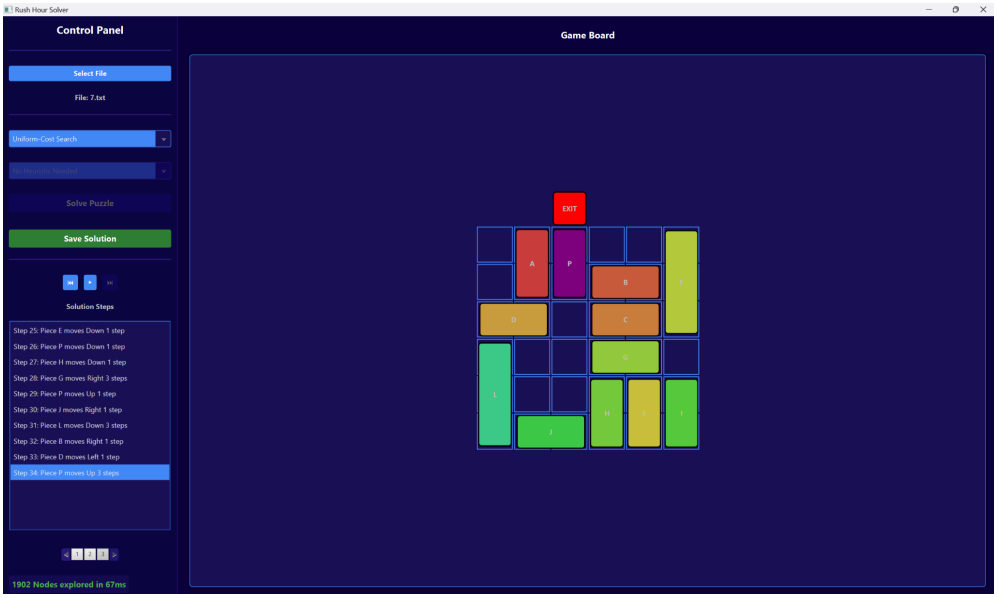
Output	
Test Case #3 A* - Blocking	
Input	6 6 11 K .A...F .ABBEF DDCCEF L..HGG L.PH.I L.PJJI
Output	
Test Case #4 A* - Blocking	
Input	6 6

	11 ABBBCC AL.P.. JL.PDD JL.EEF J...GF IIHHGF K
Output	

5.2. Test Case UCS

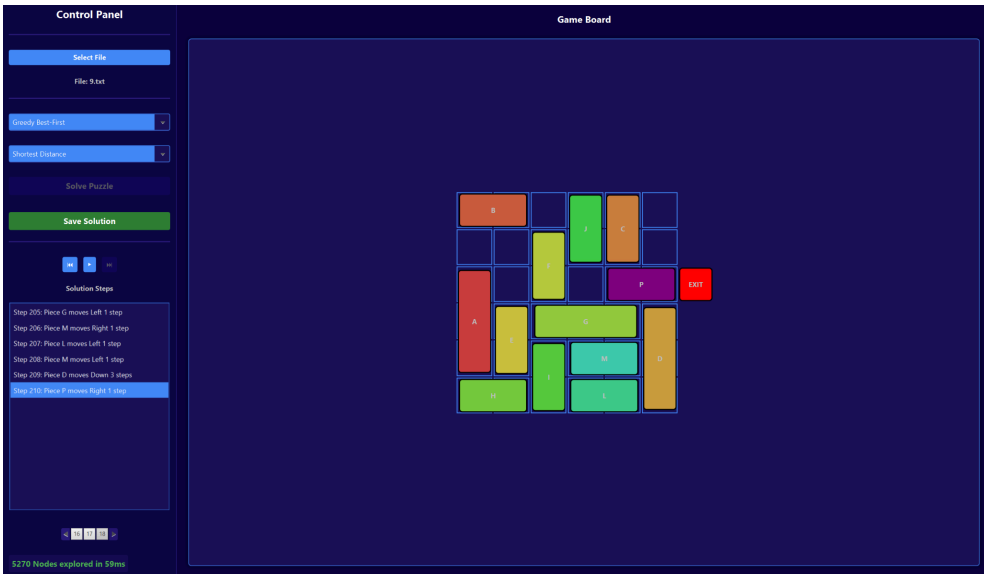
Test Case #1 UCS	
Input	6 6 12 ABB.C. AEF.CD AEFPPDK GGGJ.D ..IJMM HHILL.

Output	
Test Case #2 UCS	
Input	6 6 11 ...BAA DCCB.G DEFF.G K.EI.PP .HIJJJ .HILLL
Output	
Test Case #3 UCS	
Input	6 6 11 K .A...F .ABBEF DDCCEF

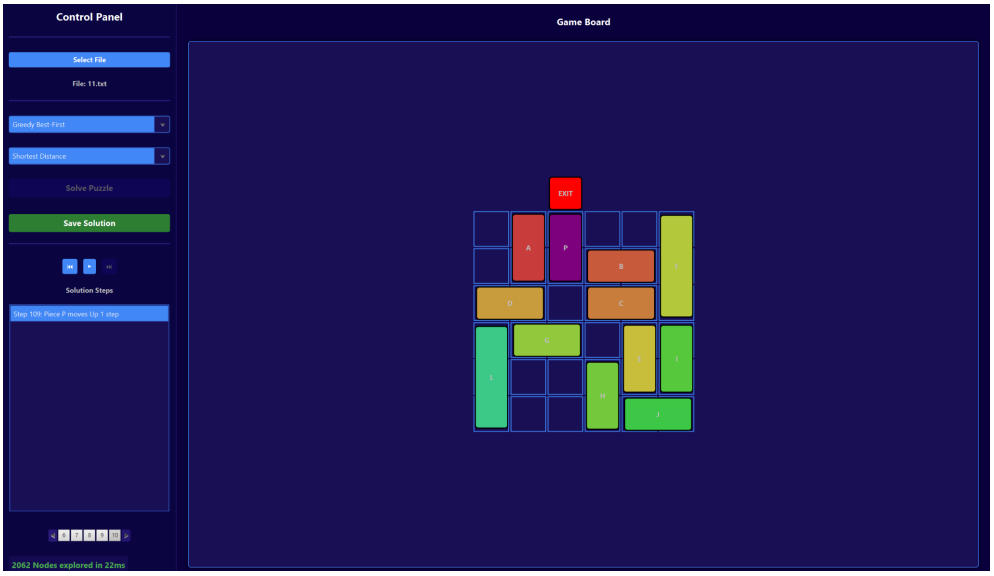
	L..HGG L.PH.I L.PJJI
Output	
Test Case #4 UCS	
Input	6 6 11 ABBBCC AL.P.. JL.PDD JL.EEF J...GF IIHHGF K

Output	

5.1. Test Case GBFS

Test Case #1 GBFS - Shortest	
Input	6 6 12 ABB.C. AEF.CD AEFPPDK GGGJ.D ..IJMM HHILL.
Output	
Test Case #2 GBFS - Shortest	

Input	6 6 11 ...BAA DCCB.G DEFF.G K.EI.PP .HIJJJ .HILLL
Output	
Test Case #3 GBFS - Blocking	
Input	6 6 11 K .A...F .ABBEF DDCCEF L..HGG L.PH.I L.PJJI

Output	
Test Case #4 GBFS - Blocking	
Input	6 6 11 ABBBCC AL.P.. JL.PDD JL.EEF J...GF IIHHGF K
Output	

BAB 6
ANALISIS PERCOBAAN

Algoritma A* melakukan pencarian dengan perhitungan fungsi evaluasi dari penjumlahan h(n) dan g(n). berdasarkan uji coba program penyelesaian puzzle rush hour. Algoritma ini memberi proses menuju titik tujuan dengan step paling sedikit. Hal ini terjadi karena A* mampu menyeimbangkan eksplorasi jalur yang murah g(n) dengan antisipasi terhadap posisi tujuan h(n). Heuristic yang digunakan pada Algoritma ini adalah shortest atau blocking. Ketika dilakukan pengujian dengan dua heuristic berbeda, algoritma ini tetap memberi hasil proses dengan step yang tidak terlalu berbeda.

Algoritma GBFS melakukan pencarian dengan perhitungan fungsi evaluasi dari h(n). Berdasarkan uji coba program Algoritma ini memberi proses menuju titik tujuan dengan step paling banyak hingga berkali-kali lipat jika dibandingkan dengan algoritma A* dan UCS. Heuristic yang digunakan pada algoritma ini adalah shortest dan blocking. Ketika digunakan heuristic shortest , step yang diberikan pada hasil akhir cenderung lebih banyak dibandingkan jumlah step pada penggunaan heuristik blocking. Walau demikian, jumlah step yang dihasilkan algoritma ini tetap lebih besar dibandingkan dua algoritma lainnya.

Algoritma Uniform Cost Search (UCS) melakukan pencarian tak-informatif yang hanya mempertimbangkan biaya dari awal hingga simpul saat ini g(n) sebagai fungsi evaluasi tanpa menggunakan estimasi heuristik. Meskipun demikian, dalam kasus rush hour puzzle yang memiliki ruang pencarian terbatas dan distribusi biaya langkah yang seragam, UCS mampu memberikan solusi dengan jumlah langkah minimum yang sebanding dengan A*. Hal ini menunjukkan bahwa UCS dapat digunakan dengan baik dalam penyelesaian puzzle rush hour.

Secara keseluruhan, ketiga algoritma A*, UCS, dan GBFS terbukti mampu menyelesaikan permasalahan *Rush Hour Puzzle*, namun efektivitasnya berbeda. A* menjadi algoritma yang paling seimbang dan efisien, karena memanfaatkan baik informasi jarak aktual maupun estimasi. UCS juga memberikan hasil optimal dalam jumlah langkah, meskipun kadang lebih lambat dari A* dalam hal waktu. Sebaliknya, GBFS cenderung mengejar tujuan secara agresif tanpa memperhatikan efisiensi jalur, sehingga sering kali memberikan solusi dengan langkah yang jauh lebih panjang. Oleh karena itu, A* dan UCS lebih disarankan untuk digunakan apabila tujuan utamanya adalah meminimalkan jumlah langkah solusi dalam permainan ini.

BAB 7
IMPLEMENTASI BONUS

6.1 GUI

Graphical User Interface yang kami buat memiliki tampilan sebagai berikut

Pada Graphical User Interface kami, user akan diminta memberi masukan sesuai alur input yang tertera pada spek tugas kecil. Suatu button atau combobox akan di disable(lock) jika user belum memasukkan input yang diminta sebelumnya.

Adapun berikut keterangan button, combobox, serta component pada aplikasi kami :

1. Select File Button : Button yang bila ditekan akan menampilkan folder sehingga user dapat memasukkan masukan file
2. Algorithm Combobox : Combobox yang bila ditekan akan menampilkan list algoritma yang tersedia
3. Heuristic Combobox : Combobox yang bila ditekan akan menampilkan list heuristic yang tersedia untuk suatu algoritma yang telah dipilih sebelumnya
4. Solve Button : Button yang bila ditekan akan menjalankan program solver sesuai dengan algoritma dan atau heuristic yang telah dipilih sebelumnya
5. Play Button : Button yang bila ditekan akan menjalankan animasi dari suatu state ke solusi
6. Pause Button : Button yang bila ditekan akan menge-pause
7. Next button : Button yang bila ditekan akan menampilkan step selanjutnya relative dari step sekarang
8. Back Button : Button yang bila ditekan akan menampilka step sebelumnya relative dari step sekarang
9. Solution Steps : Keterangan setiap langkah yang terjadi hingga mencapai solution
10. Pagination : Hasil dari solusion steps dipaginasi, maksimum step list yang ditampilkan berjumlah x dan maksimum keterangan halaman yang ditampilkan berjumlah y ,

11. Next Pagination Button : Lanjut ke halaman selanjutnya
12. Back Pagination Button : Kembali ke halaman sebelumnya
13. Board : visualisasi board, dapat menvisualisasikan board dalam tiap langkah

6.2 Heuristic Shortest

Heuristic tambahan yang diimplementasikan adalah shortest, mirip dengan Manhattan heuristic ini menghitung selisih jarak dari primary piece ke exit.

```
package com.jawa.model.heuristic;

import com.jawa.model.gameComponent.Board;
import com.jawa.model.gameComponent.Piece;
import com.jawa.model.gameComponent.Position;

public class ShortestHeuristic extends Heuristic {
   public ShortestHeuristic() {
   }

   @Override
   public int estimate(Board board) {
       Piece primary = board.getPieces().get("P");
       Position exit = board.getExitPosition();
       if (primary == null || exit == null)
           return Integer.MAX_VALUE;

       int r = primary.getRow();
       int c = primary.getCol();
       if (primary.isHorizontal()) {
           int frontCol = c + primary.getLength() - 1;
           if (r == exit.getRow() && exit.getCol() > frontCol)
               return exit.getCol() - frontCol - 1;

           if (r == exit.getRow() && exit.getCol() < c)
               return c - exit.getCol() - 1;

       } else {
           int frontRow = r + primary.getLength() - 1;
           if (c == exit.getCol() && exit.getRow() > frontRow)
               return exit.getRow() - frontRow - 1;

           if (c == exit.getCol() && exit.getRow() < r)
               return r - exit.getRow() - 1;
```

```
        }

        return Integer.MAX_VALUE;
    }

}
```

LAMPIRAN

Tautan Repository: https://github.com/WwzFwz/Tucil3_13523065_13523073

-

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	☑️	

DAFTAR PUSTAKA

K. N. Mariadi, *Implementasi Algoritma Uniform Cost Search dalam Penentuan Rute Mudik di Sumatera Barat*, Institut Teknologi Bandung, 2023. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/Makalah2023/Makalah-Matdis-2023%20(49).pdf. [Accessed: May 20, 2025].

R. P. Ubaidillah, *Penerapan Algoritma Greedy Best-First Search untuk Memecahkan Permainan Tebak Kata Katla*, Institut Teknologi Bandung, 2023. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Makalah/Makalah-Stima-2023-(107).pdf. [Accessed: May 20, 2025].

A. J. P. Fanjaya, *Perbandingan Algoritma A*, UCS, dan Greedy Best First Search pada Pencarian Solusi Permainan Words of Wonders menggunakan Regex*, Institut Teknologi Bandung, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Makalah/Makalah-IF2211-Stima-2024%20(19).pdf. [Accessed: May 20, 2025].

N. U. Maulidevi, *Penentuan Rute (Route/Path Planning) bagian 1, Bahan Kuliah IF2211 Strategi Algoritma*, Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf. [Accessed: May 20, 2025].

N. U. Maulidevi and R. Munir, *Penentuan Rute (Route/Path Planning), Bahan Kuliah IF2211 Strategi Algoritma*, Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf. [Accessed: May 20, 2025].