Laporan Tugas Kecil 3
IF2211 Strategi Algoritma

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Disusun oleh:

Muhammad Timur Kanigara - 13523055

Kefas Kurnia Jonathan - 13523113

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

2025

DAFTAR ISI

BAB 1
ALGORITMA PATHFINDING

1.1 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian informed yang menemukan jalur dengan biaya terendah dari node awal ke node tujuan. UCS menggunakan priority queue untuk menyimpan node yang akan dieksplorasi, dengan prioritas berdasarkan biaya path (g(n)) dari node awal hingga ke node saat ini.

Algoritma UCS bekerja seperti berikut:

1. Inisialisasi priority queue dengan node awal dan biaya 0
2. Selama priority queue tidak kosong:
    a. Ambil node dengan biaya terendah dari priority queue
    b. Jika node adalah node tujuan, selesai dan kembalikan solusinya
    c. Jika tidak, ekspansi node berikut:
        i. Untuk setiap successor dari node, hitung biaya total untuk mencapai successor
        ii. Jika successor belum pernah dikunjungi atau ditemukan dengan biaya yang lebih rendah:
            1. Perbarui biaya successor
            2. Tambahkan successor ke priority queue dengan prioritas berdasarkan biaya lokal

UCS selalu memperluas node dengan biaya terendah terlebih dahulu, yang menjamin bahwa ketika ada node tujuan yang ditemukan, jalur yang ditemukan adalah jalur dengan biaya terendah. Dalam kontek rush hour, UCS memperluas konfigurasi board degnan jumlah gerakan terkecil terlebih dahulu. Biaya setiap gerakan (memindahkan satu kendaraan) dianggap 1, sehingga UCS akan menghasilkan solusi dengan jumlah gerakan minimum untuk mengeluarkan mobil utama dari board.

1.2 Greedy Best First Search (GBFS)

Greedy Best First Search adalah algoritma pencarian heuristik yang selalu memperluas node yang tampak paling dekat dengan tujuan, berdasarkan fungsi herusitik h(n). Algoritma ini bersifat *greedy* karena hanya mempertimbangkan jarak ke tujuan, tanpa mempertimbangkan biaya yang telah dikeluarkan untuk mencapai node saat ini.

Algoritma GBFS bekerja seperti berikut:

1. Inisialisasi priority queue dengan node awal dan biaya 0
2. Selama priority queue tidak kosong:
    a. Ambil node dengan nilai heuristik terendah dari priority queue
    b. Jika node adalah node tujuan, selesai dan kembalikan solusinya
    c. Jika tidak, ekspansi node berikut:
        i. Hitung nilai heuristik successor h(successor)
        ii. Jika successor belum pernah dikunjungi, tambahkan successor ke priority queue dengan prioritas berdasarkan h(successor)

Dalam konteks Rush Hour, fungsi heuristik h(n) dapat diimplementasikan sebagai:

- Jumlah kendaraan yang menghalangi jalur mobil utama ke pintu keluar
- Jarak mobil utama ke pintu keluar
- Kombinasi dari dua faktor di atas

Greedy Best First Search dapat menemukan solusi dengan cepat, tapi tidak menjamin solusi optimal (jumlah gerakan minimum) karena tidak mempertimbangkan biaya yang telah dikeluarkan untuk mencapai node saat ini.

1.3 A* Search

A* Search adalah algoritma pencarian informed yang menggabungkan kelebihan UCS dan Greedy Best First Search. A* menggunakan fungsi evaluasi f(n) = g(n) + h(n), dimana g(n) adlaah biaya untuk mencapai node n dari node awal (seperti pada UCS), dan h(n) adalah perkiraan biaya dari node n ke node tujuan (seperti pada GBFS).
Algoritma A* bekerja seperti berikut:

1. Inisialisasi priority queue dengan node awal dan biaya 0
2. Selama priority queue tidak kosong:
    a. Ambil node dengan nilai f(n) terendah dari priority queue
    b. Jika node adalah node tujuan, selesai dan kembalikan solusinya
    c. Jika tidak, ekspansi node berikut:
        i. Untuk setiap successor dari node, hitung g(successor) = g(node) + cost(node,successor)
        ii. Kemudian hitung h(successor) menggunakan fungsi heuristik
        iii. Hitung f(successor) = g(successor) + h(successor)
        iv. Jika successor belum pernah dikunjungi atau ditemukan dengan nilai g lebih rendah:
            1. Perbarui nilai g, h, dan f successor
            2. Tambahkan successor ke priority queue dengan prioritas berdasarkan f(successor)

Dalam konteks Rush Hour, A* menggunakan:

- g(n) : jumlah gerakan yang telah dilakukan untuk mencapai konfigurasi papan saat ini
- h(n) : estimasi jumlah gerakan minimal yang diperlukan untuk mengeluarkan mobil utama dari papan

Dengan fungsi heuristik yang admissible, A* menjamin menemukan solusi optimal dengan jumlah node yang diperluas lebih sedikit daripada UCS.

1.4 IDA* Search (Bonus)

Iterative Deepening A* (IDA*) adalah algoritma pencarian informed (sudah mengetahui informasi) yang menggabungkan prinsip pencarian *depth-first* dengan evaluasi seperti A*, sehingga lebih hemat memori namun tetap mempertimbangkan estimasi biaya menuju goal. IDA* menggunakan fungsi evaluasi yang sama dengan A*, yaitu $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah biaya dari node awal ke node saat ini, dan $h(n)$ adalah perkiraan biaya dari node ke tujuan berdasarkan fungsi heuristik.

Algoritma IDA* bekerja seperti berikut:

1. Inisialisasi batas (*threshold*) dengan nilai awal f(*start*) = h(*start*)
2. Lakukan pencarian secara *depth-first*, tetapi hanya menelusuri node-node dengan nilai *f*(n) <= *threshold* saat ini
3. Jika tujuan ditemukan selama pencarian, pencarian dihentikan dan solusi dikembalikan
4. Jika tujuan tidak ditemukan, ambil nilai minimum *f*(n), yang melebihi *threshold* sebagai *threshold* baru dan ulangi prosesnya
5. Proses ini terus diulangi hingga solusi ditemukan atau seluruh ruang pencarian telah habis.

Dalam konteks Rush Hour, IDA* menggunakan:

- g(n) : jumlah gerakan yang telah dilakukan untuk mencapai konfigurasi papan saat ini
- h(n) : estimasi jumlah gerakan minimal yang diperlukan untuk mengeluarkan mobil utama dari papan

IDA* sangat efisien dalam hal penggunaan memori karena tidak menyimpan semua node terbuka seperti A*, namun tetap dapat menemukan solusi optimal selama fungsi heuristik yang digunakan admissible. Meskipun jumlah node yang diperluas bisa lebih banyak dibanding A*, IDA* merupakan alternatif yang baik jika penghematan ruang memori menjadi pertimbangan penting.

1.5 Analisis Perbandingan Algoritma

1.5.1 Definisi f(n) dan g(n)

Fungsi f(n) dan g(n) dalam konteks algoritma pencarian didefinisikan sebagai berikut:

- g(n) : Biaya path dari node awal ke node n. Ini adalah biaya yang telah dikeluarkan untuk mencapai node n
- h(n) : Perkiraan biaya dari node n ke node tujuan. Ini adalah fungsi heuristik yang memperkirakan "jarak" yang tersisa untuk mencapai tujuan
- f(n) : Perkiraan total biaya path dari node awal ke node tujuan melalui node n. Dalam A*, f(n) = g(n) + h(n)

Dalam implementasi algoritma:

- UCS hanya menggunakan g(n) sebagai fungsi evaluasi, di mana f(n) = g(n)
- GBFS hanya menggunakan h(n) sebagai fungsi evaluasi, di mana f(n) = h(n)
- A* menggunakan kombinasi keduanya, di mana f(n) = g(n) + h(n)

1.5.2 Admissibility Heuristik pada A*

Heuristik h(n) dinyatakan admissible jika tidak pernah overestimate biaya sebenarnya untuk mencapai tujuan dari node n. Secara formal, untuk setiap node n, h(n) <= h*(n), di mana h*(n) adalah biaya sebenarnya dari node n ke node tujuan.
Dalam Rush Hour, heuristik yang digunakan adalah:

- Jumlah kendaraan yang menghalangi jalur mobil utama ke pintu keluar
- Jarak mobil utama ke pintu keluar (jumlah sel)

Herustik ini admissible karena:

1. Jumlah kendaraan yang menghalangi: Setiap kendaraan yang menghalangi memerlukan minimal 1 gerakan untuk dipindahkan sehingga tidak overestimate
2. Jarak mobil utama ke pintu keluar : Mobil utama harus bergerak minimal sejumlah sel untuk mencapai pintu keluar sehingga tidak overestimate

Kombinasi kedua heuristik ini juga admissible karena jumlah gerakan minimal yang diperlukan pasti tidak kurang dari perkiraan heuristik.

1.5.3 Perbandingan UCS dan BFS dalam Rush Hour

Pada penyelesaian Rush Hour, algoritma UCS dan BFS menghasilkan urutan node yang dibangkitkan dan path yang sama ketika biaya setiap gerakan adalah sama. Ada beberapa alasan yang menyebabkan hal ini:

1. BFS memperluas node berdasarkan urutan level (kedalaman dari node awal)
2. UCS memperluas node berdasarkan biaya terendah dari node awal
3. Dalam Rush Hour, setiap gerakan memiliki biaya 1, sehingga biaya total sama dengan jumlah gerakan
4. Karena biaya setiap gerakan sama, node dengan level terendah juga memiliki biaya terendah
5. Dengan demikian, UCS dan BFS akan memperluas node dalam urutan yang sama dan menemukan jalur yang sama

Perbandingan akan muncul jika biaya gerakan berbeda-beda, misalnya jika gerakan horizontal dan vertikal memiliki biaya yang sama.

1.5.4 Efisiensi A* dengan UCS

Secara teoritis, algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada penyelesaian Rush Hour, alasannya karena:

1. UCS memperluas node secara “blind” berdasarkan biaya path saja, tanpa mempertimbangkan seberapa dekat node tersebut dengan tujuan
2. A* mengarahkan pencarian menggunakan informasi heuristik tentang perkiraan jarak ke tujuan
3. Dengan heuristik yang admissible, A* menjamin menemukan solusi optimal (sama seperti UCS)
4. A* dapat mengurangi jumlah node yang diperluas dengan memprioritaskan node yang lebih menjanjikan (lebih dekat dengan tujuan)

Dalam kasus Rush Hour yang kompleks dengan banyak kendaraan, A* dapat secara signifikan mengurangi ruang pencarian dengan fokus pada konfigurasi yang lebih mungkin mengarah ke solusi.

1.5.5 Optimalitas Greedy Best First Search

Secara teoritis, algoritma Greedy Best First Search tidak menjamin solusi optimal untuk penyelesaian Rush Hour, alasannya karena:

1. Greedy hanya mempertimbangkan perkiraan jarak ke tujuan (h(n)) tanpa mempertimbangkan biaya yang telah dikeluarkan (g(n))

2. Greedy dapat “tertipu” oleh node yang tampak dekat dengan tujuan tapi sebenarnya memerlukan banyak langkah tambahan
3. Dalam Rush Hour, konfigurasi di mana mobil utama dekat dengan pintu keluar tapi terhalang oleh beberapa mobil dapat menyebabkan Greedy memilih jalur suboptimal

Contohnya:

- Greedy dapat memilih untuk memindahkan primary untuk mendekati pintu keluar terlebih dahulu, meskipun hal itu menciptakan situasi “deadlock” yang memerlukan banyak gerakan tambahan untuk diselesaikan
- UCS atau A* akan mempertimbangkan biaya total dan mungkin memilih jalur yang berbeda yang menghasilkan solusi dengan jumlah gerakan yang lebih sedikit.

Greedy Best First Search mungkin menemukan solusi lebih cepat dalam beberapa kasus, tetapi solusi tersebut kemungkinan bukan yang optimal dari segi jumlah gerakan

BAB 2
SOURCE CODE

2.1 Model

Piece.java

```
package model;

public class Piece {
    private char id;
    private int rowStart;
    private int colStart;
    private int length;
    private boolean isVertical;
    private boolean isPrimary;

    /* Konstruktor Piece */
    public Piece(char id, int rowStart, int colStart, int length, boolean isVertical, boolean
isPrimary) {
        this.id = id;
        this.rowStart = rowStart;
        this.colStart = colStart;
        this.length = length;
        this.isVertical = isVertical;
        this.isPrimary = isPrimary;
    }

    /* CCtor */
    public Piece(Piece other) {
        this.id = other.id;
        this.rowStart = other.rowStart;
        this.colStart = other.colStart;
        this.length = other.length;
        this.isVertical = other.isVertical;
        this.isPrimary = other.isPrimary;
    }

   /* Cek piece nempatin cell atau ga */
    public boolean occupies(int row, int col) {
        if (isVertical) {
            return col == colStart && row >= rowStart && row < rowStart + length;
        } else {
            return row == rowStart && col >= colStart && col < colStart + length;
        }
    }

   /* Move piece */
    public void move(int steps) {
        if (isVertical) {
            rowStart += steps;
        } else {
            colStart += steps;
        }
    }

    /* Getter dan Setter */
    public char getId() {
        return id;
    }
```

```
    public void setId(char id) {
        this.id = id;
    }

    public int getRowStart() {
        return rowStart;
    }

    public void setRowStart(int rowStart) {
        this.rowStart = rowStart;
    }

    public int getColStart() {
        return colStart;
    }

    public void setColStart(int colStart) {
        this.colStart = colStart;
    }

    public int getLength() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public boolean isVertical() {
        return isVertical;
    }

    public void setVertical(boolean isVertical) {
        this.isVertical = isVertical;
    }

    public boolean isHorizontal() {
        return !isVertical;
    }

    public boolean isPrimary() {
        return isPrimary;
    }

    public void setPrimary(boolean isPrimary) {
        this.isPrimary = isPrimary;
    }

    public int getEnd() {
        return isVertical ? rowStart + length - 1 : colStart + length - 1;
    }

    public int getRow() {
        return rowStart;
    }

    public void setRow(int row) {
        this.rowStart = row;
    }

    public int getCol() {
        return colStart;
    }

    public void setCol(int col) {
        this.colStart = col;
```

```
    }

    public char getSymbol() {
        return id;
    }

    @Override
    public String toString() {
        String orientation = isVertical ? "vertical" : "horizontal";
        return "Piece " + id + " at (" + rowStart + "," + colStart + "), " +
               "length: " + length + ", " + orientation +
               (isPrimary ? " [PRIMARY]" : "");
    }
}
```

Board.java

```
package model;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Board {
    private int rows;
    private int cols;
    private char[][] grid;
    private int exitRow;
    private int exitCol;
    private ArrayList<Piece> pieces;

    private static final int BORDER_SIZE = 1;

    /* Konstruktor Board */
    public Board(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        this.grid = new char[rows + 2*BORDER_SIZE][cols + 2*BORDER_SIZE];
        this.pieces = new ArrayList<>();

        for (int i = 0; i < rows + 2*BORDER_SIZE; i++) {
            Arrays.fill(grid[i], '.');
        }

        this.exitRow = -1;
        this.exitCol = -1;
    }

    /* CCtor */
    public Board(Board other) {
        this.rows = other.rows;
        this.cols = other.cols;
        this.exitRow = other.exitRow;
        this.exitCol = other.exitCol;

        this.grid = new char[rows + 2*BORDER_SIZE][cols + 2*BORDER_SIZE];
        for (int i = 0; i < rows + 2*BORDER_SIZE; i++) {
            System.arraycopy(other.grid[i], 0, this.grid[i], 0, cols + 2*BORDER_SIZE);
        }

        this.pieces = new ArrayList<>();
        for (Piece piece : other.pieces) {
            this.pieces.add(new Piece(piece));
        }
```

```
    }


    public void setExit(int row, int col) {
        this.exitRow = row;
        this.exitCol = col;

        int borderRow, borderCol;

        if (row == -1) {                        // exit atas
            borderRow = 0;
            borderCol = col + BORDER_SIZE;
        } else if (row == rows) {               // exit bawah
            borderRow = rows + BORDER_SIZE;
            borderCol = col + BORDER_SIZE;
        } else if (col == -1) {                 // exit kiri
            borderRow = row + BORDER_SIZE;
            borderCol = 0;
        } else if (col == cols) {               // exit kanan
            borderRow = row + BORDER_SIZE;
            borderCol = cols + BORDER_SIZE;
        } else {                                // case di dalem
            borderRow = row + BORDER_SIZE;
            borderCol = col + BORDER_SIZE;
        }

        grid[borderRow][borderCol] = 'K';
    }

    public void addPiece(Piece piece) {
        pieces.add(piece);
        updatePieceInGrid(piece);
    }

    private void updatePieceInGrid(Piece piece) {
        int row = piece.getRow();
        int col = piece.getCol();
        int length = piece.getLength();
        boolean isHorizontal = piece.isHorizontal();
        char symbol = piece.getSymbol();

        int gridRow = row + BORDER_SIZE;
        int gridCol = col + BORDER_SIZE;

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

            if (r >= BORDER_SIZE && r < rows + BORDER_SIZE &&
                c >= BORDER_SIZE && c < cols + BORDER_SIZE) {
                if (grid[r][c] != 'K') {
                    grid[r][c] = symbol;
                }
            }
        }
    }

    public void updateGrid() {
        for (int i = 0; i < rows + 2*BORDER_SIZE; i++) {
            Arrays.fill(grid[i], '.');
        }

        if (exitRow != -1 || exitCol != -1) {
            int borderRow, borderCol;

            if (exitRow == -1) {                    // exit atas
```

```
            borderRow = 0;
            borderCol = exitCol + BORDER_SIZE;
        } else if (exitRow == rows) {           // exit bawah
            borderRow = rows + BORDER_SIZE;
            borderCol = exitCol + BORDER_SIZE;
        } else if (exitCol == -1) {             // exit kiri
            borderRow = exitRow + BORDER_SIZE;
            borderCol = 0;
        } else if (exitCol == cols) {           // exit kanan
            borderRow = exitRow + BORDER_SIZE;
            borderCol = cols + BORDER_SIZE;
        } else {
            borderRow = exitRow + BORDER_SIZE;
            borderCol = exitCol + BORDER_SIZE;
        }

        grid[borderRow][borderCol] = 'K';
    }

    for (Piece piece : pieces) {
        updatePieceInGrid(piece);
    }
}

/* Move piece */
public boolean movePieceTo(Piece piece, int targetRow, int targetCol) {
    if (canPlacePiece(piece, targetRow, targetCol)) {
        // Update piece position
        piece.setRow(targetRow);
        piece.setCol(targetCol);

        // Update the grid
        updateGrid();

        return true;
    }

    return false;
}

public boolean movePiece(Piece piece, int direction) {
    int newRow = piece.getRow();
    int newCol = piece.getCol();

    if (piece.isHorizontal()) {
        newCol += direction;
    } else {
        newRow += direction;
    }

    return movePieceTo(piece, newRow, newCol);
}

/* Ngecek bisa atau ga */
public boolean canPlacePiece(Piece piece, int newRow, int newCol) {
    int length = piece.getLength();
    boolean isHorizontal = piece.isHorizontal();
    boolean isPrimary = piece.isPrimary();

    int gridRow = newRow + BORDER_SIZE;
    int gridCol = newCol + BORDER_SIZE;

    for (int i = 0; i < length; i++) {
        int r = isHorizontal ? gridRow : gridRow + i;
        int c = isHorizontal ? gridCol + i : gridCol;
```

```
            boolean outsideBoard = r < BORDER_SIZE || r >= rows + BORDER_SIZE ||
                                   c < BORDER_SIZE || c >= cols + BORDER_SIZE;

            if (outsideBoard) {
                if (isPrimary) {
                    // Top exit
                    if (exitRow == -1 && r == 0 && c == exitCol + BORDER_SIZE) {
                        continue;
                    }
                    // Bottom exit
                    if (exitRow == rows && r == rows + BORDER_SIZE && c == exitCol +
BORDER_SIZE) {
                        continue;
                    }
                    // Left exit
                    if (exitCol == -1 && c == 0 && r == exitRow + BORDER_SIZE) {
                        continue;
                    }
                    // Right exit
                    if (exitCol == cols && c == cols + BORDER_SIZE && r == exitRow +
BORDER_SIZE) {
                        continue;
                    }
                }
                return false;
            }

            boolean isOwnPosition = false;
            int origRow = piece.getRow() + BORDER_SIZE;
            int origCol = piece.getCol() + BORDER_SIZE;

            for (int j = 0; j < length; j++) {
                int origR = isHorizontal ? origRow : origRow + j;
                int origC = isHorizontal ? origCol + j : origCol;

                if (r == origR && c == origC) {
                    isOwnPosition = true;
                    break;
                }
            }

            if (!isOwnPosition) {
                if (grid[r][c] != '.' && !(isPrimary && grid[r][c] == 'K')) {
                    return false;
                }
            }
        }

        return true;
    }

    /* Ngecek apakah cleat buat bbrp perjalanan sekaligus */
    private boolean isClearPath(Piece piece, int targetRow, int targetCol) {
        int currentRow = piece.getRow();
        int currentCol = piece.getCol();
        boolean isHorizontal = piece.isHorizontal();

        if (isHorizontal) {
            int minCol = Math.min(currentCol, targetCol);
            int maxCol = Math.max(currentCol, targetCol);

            for (int c = minCol + 1; c <= maxCol; c++) {
                if (c == currentCol) continue;

                Board tempBoard = new Board(this);
                Piece tempPiece = tempBoard.getPieces().get(pieces.indexOf(piece));
```

```
            if (!tempBoard.movePieceTo(tempPiece, currentRow, c)) {
                return false;
            }
        }
    }
    else {
        int minRow = Math.min(currentRow, targetRow);
        int maxRow = Math.max(currentRow, targetRow);

        for (int r = minRow + 1; r <= maxRow; r++) {
            if (r == currentRow) continue;

            Board tempBoard = new Board(this);
            Piece tempPiece = tempBoard.getPieces().get(pieces.indexOf(piece));

            if (!tempBoard.movePieceTo(tempPiece, r, currentCol)) {
                return false;
            }
        }
    }

    return true;
}

public boolean isSolved() {
    Piece primaryPiece = null;
    for (Piece piece : pieces) {
        if (piece.isPrimary()) {
            primaryPiece = piece;
            break;
        }
    }

    if (primaryPiece == null) {
        return false;
    }

    int row = primaryPiece.getRow();
    int col = primaryPiece.getCol();
    int length = primaryPiece.getLength();
    boolean isHorizontal = primaryPiece.isHorizontal();

    if (isHorizontal) {
        // kalau exit di kanan
        if (exitCol == cols && row == exitRow && col + length - 1 == cols - 1) {
            return true;
        }
        // kalau exit di kiri
        else if (exitCol == -1 && row == exitRow && col == 0) {
            return true;
        }
    } else {
        // kalau exit di atas
        if (exitRow == -1 && col == exitCol && row == 0) {
            return true;
        }
        // kalau exit di bawah
        else if (exitRow == rows && col == exitCol && row + length - 1 == rows - 1) {
            return true;
        }
    }

    return false;
}
```

```
public List<Board> getNextStates() {
    List<Board> nextStates = new ArrayList<>();

    for (int i = 0; i < pieces.size(); i++) {
        Piece piece = pieces.get(i);
        boolean isHorizontal = piece.isHorizontal();
        int currentRow = piece.getRow();
        int currentCol = piece.getCol();

        if (isHorizontal) {
            for (int newCol = 0; newCol <= cols - piece.getLength(); newCol++) {
                if (newCol == currentCol) continue;

                if (isClearPath(piece, currentRow, newCol)) {
                    Board newBoard = new Board(this);
                    Piece newPiece = newBoard.getPieces().get(i);

                    if (newBoard.movePieceTo(newPiece, currentRow, newCol)) {
                        nextStates.add(newBoard);
                    }
                }
            }
        } else {
            for (int newRow = 0; newRow <= rows - piece.getLength(); newRow++) {
                if (newRow == currentRow) continue;

                if (isClearPath(piece, newRow, currentCol)) {
                    Board newBoard = new Board(this);
                    Piece newPiece = newBoard.getPieces().get(i);

                    if (newBoard.movePieceTo(newPiece, newRow, currentCol)) {
                        nextStates.add(newBoard);
                    }
                }
            }
        }
    }

    return nextStates;
}

public int getRows() {
    return rows;
}

public int getCols() {
    return cols;
}

public int getExitRow() {
    return exitRow;
}

public int getExitCol() {
    return exitCol;
}

public ArrayList<Piece> getPieces() {
    return pieces;
}

public Piece getPrimaryPiece() {
    for (Piece piece : pieces) {
        if (piece.isPrimary()) {
            return piece;
        }
```

```
        }
        return null;
    }

    public char[][] getGrid() {
        char[][] visibleGrid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                visibleGrid[i][j] = grid[i + BORDER_SIZE][j + BORDER_SIZE];
            }
        }
        return visibleGrid;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                sb.append(grid[i + BORDER_SIZE][j + BORDER_SIZE]);
            }
            if (i < rows - 1) {
                sb.append("\n");
            }
        }
        return sb.toString();
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        Board other = (Board) obj;
        if (rows != other.rows || cols != other.cols) return false;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i + BORDER_SIZE][j + BORDER_SIZE] != other.grid[i + BORDER_SIZE][j +
BORDER_SIZE]) {
                    return false;
                }
            }
        }

        return true;
    }

    @Override
    public int hashCode() {
        int result = rows;
        result = 31 * result + cols;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result = 31 * result + grid[i + BORDER_SIZE][j + BORDER_SIZE];
            }
        }

        return result;
    }
}
```

Move.java

```
package model;

/* Kelas sebagai representasi move */
public class Move {
    private int pieceIndex;
    private int targetRow;
    private int targetCol;
    private int fromRow;
    private int fromCol;

    /* Konstruktor move */
    public Move(int pieceIndex, int fromRow, int fromCol, int targetRow, int targetCol) {
        this.pieceIndex = pieceIndex;
        this.fromRow = fromRow;
        this.fromCol = fromCol;
        this.targetRow = targetRow;
        this.targetCol = targetCol;
    }

    public String getDirection(boolean isVertical) {
        if (isVertical) {
            if (targetRow < fromRow) {
                return "atas";
            } else {
                return "bawah";
            }
        } else {
            if (targetCol < fromCol) {
                return "kiri";
            } else {
                return "kanan";
            }
        }
    }

    public int getDistance(boolean isVertical) {
        if (isVertical) {
            return Math.abs(targetRow - fromRow);
        } else {
            return Math.abs(targetCol - fromCol);
        }
    }

    public int getPieceIndex() {
        return pieceIndex;
    }

    public int getTargetRow() {
        return targetRow;
    }

    public int getTargetCol() {
        return targetCol;
    }

    public int getFromRow() {
        return fromRow;
    }

    public int getFromCol() {
        return fromCol;
    }

    @Override
    public String toString() {
        return "Move piece " + pieceIndex + " from (" + fromRow + "," + fromCol + ") to (" +
```

```
                targetRow + "," + targetCol + ")";
    }
}
```

2.2 Algorithm

UCS.java

```
package algorithm;

import java.util.*;
import model.Board;
import model.Move;
import model.Piece;
import util.BoardPrinter;

public class UCS {
    private int nodesVisited = 0;
    private gui.Gui.SolutionCollector collector;

    public UCS(gui.Gui.SolutionCollector collector){
        this.collector = collector;
    }

    /* Fungsi solver buat UCS */
    public void solve(Board initialBoard) {
        long startTime = System.currentTimeMillis();

        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node ->
node.cost));

        Set<String> visited = new HashSet<>();

        queue.add(new Node(initialBoard, null, null, 0));

        BoardPrinter.printInitialBoard(initialBoard);

        boolean solved = false;
        Node solution = null;

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            nodesVisited++;

            String boardString = current.board.toString();
            if (visited.contains(boardString)) {
                continue;
            }

            visited.add(boardString);

            if (current.board.isSolved()) {
                solved = true;
                solution = current;
                break;
            }

            List<Board> nextStates = current.board.getNextStates();

            for (int i = 0; i < nextStates.size(); i++) {
                Board nextBoard = nextStates.get(i);
```

```
            if (visited.contains(nextBoard.toString())) {
                continue;
            }

            Move move = findMove(current.board, nextBoard);

            queue.add(new Node(
                nextBoard,
                current,
                move,
                current.cost + 1
            ));
        }
    }

    long endTime = System.currentTimeMillis();
    double executionTime = (endTime - startTime) / 1000.0;

    if (solved) {
        printSolution(solution);

        System.out.println("Jumlah langkah: " + solution.cost);
        System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
        System.out.println("Waktu eksekusi: " + executionTime + " detik");
    } else {
        System.out.println("Tidak ada solusi yang ditemukan!");
        System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
        System.out.println("Waktu eksekusi: " + executionTime + " detik");
    }

    if (solved && collector != null) {
        List<Node> path = new ArrayList<>();
        Node current = solution;

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

        Collections.reverse(path);

        for (Node node : path) {
            collector.addStep(node.board);
        }
    }
}

private Move findMove(Board from, Board to) {
    List<Piece> fromPieces = from.getPieces();
    List<Piece> toPieces = to.getPieces();

    for (int i = 0; i < fromPieces.size(); i++) {
        Piece fromPiece = fromPieces.get(i);
        Piece toPiece = toPieces.get(i);

        if (fromPiece.getRow() != toPiece.getRow() ||
            fromPiece.getCol() != toPiece.getCol()) {
            return new Move(
                i,
                fromPiece.getRow(),
                fromPiece.getCol(),
                toPiece.getRow(),
                toPiece.getCol()
            );
        }
    }
```

```
        throw new IllegalStateException("Could not find the move between board states");
    }

    /* Fungsi utk print solusi */
    private void printSolution(Node solution) {
        List<Node> path = new ArrayList<>();
        Node current = solution;

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

        Collections.reverse(path);

        for (int i = 0; i < path.size(); i++) {
            Node node = path.get(i);
            Move move = node.move;
            int pieceIndex = move.getPieceIndex();
            Piece piece = node.board.getPieces().get(pieceIndex);
            char pieceId = piece.getId();

            String direction = move.getDirection(piece.isVertical());
            int distance = move.getDistance(piece.isVertical());

            BoardPrinter.printBoardAfterMove(node.board, i + 1, pieceId, direction, distance);
        }
    }

    private static class Node {
        Board board;  // Current board state
        Node parent;  // Parent node
        Move move;    // Move that was applied to reach this state
        int cost;     // Path cost (number of moves from initial state)

        Node(Board board, Node parent, Move move, int cost) {
            this.board = board;
            this.parent = parent;
            this.move = move;
            this.cost = cost;
        }
    }

    public int getNodesVisited(){
        return this.nodesVisited;
    }
}
```

GBFS.java

```
package algorithm;

import java.util.*;
import model.Board;
import model.Move;
import model.Piece;
import util.BoardPrinter;

public class GBFS {
    private int nodesVisited = 0;
    private int heuristicType;
    private gui.Gui.SolutionCollector collector;
```

```
    // Tipe heuristik
    public static final int BLOCKING_PIECES = 1;
    public static final int MANHATTAN_DISTANCE = 2;
    public static final int COMBINED = 3;

    public GBFS(int heuristicType, gui.Gui.SolutionCollector collector) {
        this.heuristicType = heuristicType;
        this.collector = collector;
    }

    public GBFS() {
        this(BLOCKING_PIECES,null);
    }

    /* Fungsi solver buat GBFS */    public void solve(Board initialBoard) {
        long startTime = System.currentTimeMillis();

        System.out.println("Using heuristic: " + getHeuristicName());

        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node ->
node.heuristic));

        Set<String> visited = new HashSet<>();

        int initialHeuristic = calculateHeuristic(initialBoard);
        queue.add(new Node(initialBoard, null, null, 0, initialHeuristic));

        BoardPrinter.printInitialBoard(initialBoard);

        boolean solved = false;
        Node solution = null;

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            nodesVisited++;

            String boardString = current.board.toString();
            if (visited.contains(boardString)) {
                continue;
            }

            visited.add(boardString);

            if (current.board.isSolved()) {
                solved = true;
                solution = current;
                break;
            }

            List<Board> nextStates = current.board.getNextStates();

            for (int i = 0; i < nextStates.size(); i++) {
                Board nextBoard = nextStates.get(i);

                if (visited.contains(nextBoard.toString())) {
                    continue;
                }

                Move move = findMove(current.board, nextBoard);

                int heuristic = calculateHeuristic(nextBoard);

                queue.add(new Node(
                    nextBoard,
                    current,
                    move,
```

```
                    current.cost + 1,
                    heuristic
                ));
            }
        }

        long endTime = System.currentTimeMillis();
        double executionTime = (endTime - startTime) / 1000.0;

        if (solved && collector != null) {
            List<Node> path = new ArrayList<>();
            Node current = solution;

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

            Collections.reverse(path);

            for (Node node : path) {
                collector.addStep(node.board);
            }
        }

        if (solved) {
            printSolution(solution);

            System.out.println("Jumlah langkah: " + solution.cost);
            System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
            System.out.println("Waktu eksekusi: " + executionTime + " detik");
        } else {
            System.out.println("Tidak ada solusi yang ditemukan!");
            System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
            System.out.println("Waktu eksekusi: " + executionTime + " detik");
        }
    }

    private String getHeuristicName() {
        switch (heuristicType) {
            case BLOCKING_PIECES:
                return "Blocking Pieces";
            case MANHATTAN_DISTANCE:
                return "Manhattan Distance";
            case COMBINED:
                return "Combined (Blocking + Manhattan)";
            default:
                return "Unknown";
        }
    }

    /* Kalkulasi heuristik sesuai inputan user */
    private int calculateHeuristic(Board board) {
        switch (heuristicType) {
            case MANHATTAN_DISTANCE:
                return calculateManhattanHeuristic(board);
            case COMBINED:
                return calculateCombinedHeuristic(board);
            case BLOCKING_PIECES:
            default:
                return calculateBlockingPiecesHeuristic(board);
        }
    }

    /* Heuristik mobil yg block */
    private int calculateBlockingPiecesHeuristic(Board board) {
```

```
        Piece primaryPiece = board.getPrimaryPiece();
        if (primaryPiece == null) {
            return Integer.MAX_VALUE;
        }

        int exitRow = board.getExitRow();
        int exitCol = board.getExitCol();
        char[][] grid = board.getGrid();
        int rows = board.getRows();
        int cols = board.getCols();

        int blockingPieces = 0;

        if (primaryPiece.isHorizontal()) {
            if (exitCol == cols) {
                int row = primaryPiece.getRow();
                for (int col = primaryPiece.getCol() + primaryPiece.getLength(); col < cols;
col++) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            } else if (exitCol == -1) {
                int row = primaryPiece.getRow();
                for (int col = primaryPiece.getCol() - 1; col >= 0; col--) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            }
        } else {
            if (exitRow == rows) {
                int col = primaryPiece.getCol();
                for (int row = primaryPiece.getRow() + primaryPiece.getLength(); row < rows;
row++) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            } else if (exitRow == -1) {
                int col = primaryPiece.getCol();
                for (int row = primaryPiece.getRow() - 1; row >= 0; row--) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            }
        }

        if (blockingPieces == 0) {
            if (primaryPiece.isHorizontal()) {
                if (exitCol == cols) {
                    int distanceToExit = cols - (primaryPiece.getCol() +
primaryPiece.getLength());
                    return distanceToExit > 0 ? 1 : 0;
                } else if (exitCol == -1) {
                    int distanceToExit = primaryPiece.getCol();
                    return distanceToExit > 0 ? 1 : 0;
                }
            } else {
                if (exitRow == rows) {
                    int distanceToExit = rows - (primaryPiece.getRow() +
primaryPiece.getLength());
                    return distanceToExit > 0 ? 1 : 0;
                } else if (exitRow == -1) {
                    int distanceToExit = primaryPiece.getRow();
```

```
                    return distanceToExit > 0 ? 1 : 0;
                }
            }
        }

        return blockingPieces;
    }

    /* Heuristik Manhattan */
    private int calculateManhattanHeuristic(Board board) {
        Piece primaryPiece = board.getPrimaryPiece();
        if (primaryPiece == null) {
            return Integer.MAX_VALUE;
        }

        int exitRow = board.getExitRow();
        int exitCol = board.getExitCol();
        int rows = board.getRows();
        int cols = board.getCols();

        int manhattanDistance = 0;

        if (primaryPiece.isHorizontal()) {
            if (exitCol == cols) {
                manhattanDistance = cols - (primaryPiece.getCol() + primaryPiece.getLength());
            } else if (exitCol == -1) {
                manhattanDistance = primaryPiece.getCol();
            }
        } else {
            if (exitRow == rows) {
                manhattanDistance = rows - (primaryPiece.getRow() + primaryPiece.getLength());
            } else if (exitRow == -1) {
                manhattanDistance = primaryPiece.getRow();
            }
        }

        char[][] grid = board.getGrid();
        boolean hasObstacles = false;

        if (primaryPiece.isHorizontal()) {
            if (exitCol == cols) {
                int row = primaryPiece.getRow();
                for (int col = primaryPiece.getCol() + primaryPiece.getLength(); col < cols;
col++) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        hasObstacles = true;
                        break;
                    }
                }
            } else if (exitCol == -1) {
                int row = primaryPiece.getRow();
                for (int col = primaryPiece.getCol() - 1; col >= 0; col--) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        hasObstacles = true;
                        break;
                    }
                }
            }
        } else {
            if (exitRow == rows) {
                int col = primaryPiece.getCol();
                for (int row = primaryPiece.getRow() + primaryPiece.getLength(); row < rows;
row++) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        hasObstacles = true;
                        break;
```

```
                }
            }
        } else if (exitRow == -1) {
            int col = primaryPiece.getCol();
            for (int row = primaryPiece.getRow() - 1; row >= 0; row--) {
                if (grid[row][col] != '.' && grid[row][col] != 'K') {
                    hasObstacles = true;
                    break;
                }
            }
        }
    }

    return manhattanDistance + (hasObstacles ? 10 : 0);
}

/* Kalkulasi heuristik yg digabung (ketiga) */
private int calculateCombinedHeuristic(Board board) {
    int blockingPieces = calculateBlockingPiecesHeuristic(board);
    int manhattanDistance = calculateManhattanHeuristic(board);

    if (blockingPieces == 0) {
        return manhattanDistance;
    }

    return blockingPieces * 10 + manhattanDistance;
}

private Move findMove(Board from, Board to) {
    List<Piece> fromPieces = from.getPieces();
    List<Piece> toPieces = to.getPieces();

    for (int i = 0; i < fromPieces.size(); i++) {
        Piece fromPiece = fromPieces.get(i);
        Piece toPiece = toPieces.get(i);

        if (fromPiece.getRow() != toPiece.getRow() ||
            fromPiece.getCol() != toPiece.getCol()) {
            return new Move(
                i,
                fromPiece.getRow(),
                fromPiece.getCol(),
                toPiece.getRow(),
                toPiece.getCol()
            );
        }
    }

    throw new IllegalStateException("Could not find the move between board states");
}

private void printSolution(Node solution) {
    List<Node> path = new ArrayList<>();
    Node current = solution;

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

    Collections.reverse(path);

    for (int i = 0; i < path.size(); i++) {
        Node node = path.get(i);
        Move move = node.move;
        int pieceIndex = move.getPieceIndex();
```

```
            Piece piece = node.board.getPieces().get(pieceIndex);
            char pieceId = piece.getId();

            String direction = move.getDirection(piece.isVertical());
            int distance = move.getDistance(piece.isVertical());

            BoardPrinter.printBoardAfterMove(node.board, i + 1, pieceId, direction, distance);
        }
    }

    private static class Node {
        Board board;
        Node parent;
        Move move;
        int cost;
        int heuristic;

        Node(Board board, Node parent, Move move, int cost, int heuristic) {
            this.board = board;
            this.parent = parent;
            this.move = move;
            this.cost = cost;
            this.heuristic = heuristic;
        }
    }

    public int getNodesVisited(){
        return this.nodesVisited;
    }
}
```

AStar.java

```
package algorithm;

import java.util.*;
import model.Board;
import model.Move;
import model.Piece;
import util.BoardPrinter;

public class AStar {
    private int nodesVisited = 0;
    private int heuristicType;
    private gui.Gui.SolutionCollector collector;

    public static final int BLOCKING_PIECES = 1;
    public static final int MANHATTAN_DISTANCE = 2;
    public static final int COMBINED = 3;

    public AStar(int heuristicType, gui.Gui.SolutionCollector collector) {
        this.heuristicType = heuristicType;
        this.collector = collector;
    }

    public AStar() {
        this(BLOCKING_PIECES, null);
    }

    /* Fungsi solver AStar */
    public void solve(Board initialBoard) {
        long startTime = System.currentTimeMillis();

        System.out.println("Using heuristic: " + getHeuristicName());
```

```
PriorityQueue<Node> queue = new PriorityQueue<>(
    Comparator.comparingInt(node -> node.cost + node.heuristic)
);

Set<String> visited = new HashSet<>();

int initialHeuristic = calculateHeuristic(initialBoard);
queue.add(new Node(initialBoard, null, null, 0, initialHeuristic));

BoardPrinter.printInitialBoard(initialBoard);

boolean solved = false;
Node solution = null;

while (!queue.isEmpty()) {
    Node current = queue.poll();
    nodesVisited++;

    String boardString = current.board.toString();
    if (visited.contains(boardString)) {
        continue;
    }

    visited.add(boardString);

    if (current.board.isSolved()) {
        solved = true;
        solution = current;
        break;
    }

    List<Board> nextStates = current.board.getNextStates();

    for (int i = 0; i < nextStates.size(); i++) {
        Board nextBoard = nextStates.get(i);

        if (visited.contains(nextBoard.toString())) {
            continue;
        }

        Move move = findMove(current.board, nextBoard);

        int heuristic = calculateHeuristic(nextBoard);

        queue.add(new Node(
            nextBoard,
            current,
            move,
            current.cost + 1,
            heuristic
        ));
    }
}

long endTime = System.currentTimeMillis();
double executionTime = (endTime - startTime) / 1000.0;

if (solved && collector != null) {
    List<Node> path = new ArrayList<>();
    Node current = solution;

    collector.addStep(initialBoard);

    while (current != null) {
        path.add(current);
```

```
            current = current.parent;
        }

        Collections.reverse(path);

        for (int i = 1; i < path.size(); i++) {
            collector.addStep(path.get(i).board);
        }
    }

    if (solved) {
        printSolution(solution);

        System.out.println("Jumlah langkah: " + solution.cost);
        System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
        System.out.println("Waktu eksekusi: " + executionTime + " detik");
    } else {
        System.out.println("Tidak ada solusi yang ditemukan!");
        System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
        System.out.println("Waktu eksekusi: " + executionTime + " detik");
    }
}

private String getHeuristicName() {
    switch (heuristicType) {
        case BLOCKING_PIECES:
            return "Blocking Pieces";
        case MANHATTAN_DISTANCE:
            return "Manhattan Distance";
        case COMBINED:
            return "Combined (Blocking + Manhattan)";
        default:
            return "Unknown";
    }
}

private int calculateHeuristic(Board board) {
    switch (heuristicType) {
        case MANHATTAN_DISTANCE:
            return calculateManhattanHeuristic(board);
        case COMBINED:
            return calculateCombinedHeuristic(board);
        case BLOCKING_PIECES:
        default:
            return calculateBlockingPiecesHeuristic(board);
    }
}

/* Heuristik mobil yg ngeblok */
private int calculateBlockingPiecesHeuristic(Board board) {
    Piece primaryPiece = board.getPrimaryPiece();
    if (primaryPiece == null) {
        return Integer.MAX_VALUE;
    }

    int exitRow = board.getExitRow();
    int exitCol = board.getExitCol();
    char[][] grid = board.getGrid();
    int rows = board.getRows();
    int cols = board.getCols();

    int blockingPieces = 0;

    if (primaryPiece.isHorizontal()) {
        if (exitCol == cols) {
            int row = primaryPiece.getRow();
```

```
                for (int col = primaryPiece.getCol() + primaryPiece.getLength(); col < cols;
col++) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            } else if (exitCol == -1) {
                int row = primaryPiece.getRow();
                for (int col = primaryPiece.getCol() - 1; col >= 0; col--) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            }
        } else {
            if (exitRow == rows) {
                int col = primaryPiece.getCol();
                for (int row = primaryPiece.getRow() + primaryPiece.getLength(); row < rows;
row++) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            } else if (exitRow == -1) {
                int col = primaryPiece.getCol();
                for (int row = primaryPiece.getRow() - 1; row >= 0; row--) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            }
        }

        return blockingPieces;
    }

    /* Heuristik Manhattan */
    private int calculateManhattanHeuristic(Board board) {
        Piece primaryPiece = board.getPrimaryPiece();
        if (primaryPiece == null) {
            return Integer.MAX_VALUE;
        }

        int exitRow = board.getExitRow();
        int exitCol = board.getExitCol();
        int rows = board.getRows();
        int cols = board.getCols();

        int manhattanDistance = 0;

        if (primaryPiece.isHorizontal()) {
            if (exitCol == cols) {
                manhattanDistance = cols - (primaryPiece.getCol() + primaryPiece.getLength());
            } else if (exitCol == -1) {
                manhattanDistance = primaryPiece.getCol();
            }
        } else {
            if (exitRow == rows) {
                manhattanDistance = rows - (primaryPiece.getRow() + primaryPiece.getLength());
            } else if (exitRow == -1) {
                manhattanDistance = primaryPiece.getRow();
            }
        }

        return manhattanDistance;
    }
```

```
/* Kombinasi heursitik */
private int calculateCombinedHeuristic(Board board) {
    int blockingPieces = calculateBlockingPiecesHeuristic(board);
    int manhattanDistance = calculateManhattanHeuristic(board);

    return blockingPieces * 2 + manhattanDistance;
}

private Move findMove(Board from, Board to) {
    List<Piece> fromPieces = from.getPieces();
    List<Piece> toPieces = to.getPieces();

    for (int i = 0; i < fromPieces.size(); i++) {
        Piece fromPiece = fromPieces.get(i);
        Piece toPiece = toPieces.get(i);

        if (fromPiece.getRow() != toPiece.getRow() ||
            fromPiece.getCol() != toPiece.getCol()) {
            return new Move(
                i,
                fromPiece.getRow(),
                fromPiece.getCol(),
                toPiece.getRow(),
                toPiece.getCol()
            );
        }
    }


    throw new IllegalStateException("Could not find the move between board states");
}

private void printSolution(Node solution) {

    List<Node> path = new ArrayList<>();
    Node current = solution;

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

    Collections.reverse(path);

    for (int i = 0; i < path.size(); i++) {
        Node node = path.get(i);
        Move move = node.move;
        int pieceIndex = move.getPieceIndex();
        Piece piece = node.board.getPieces().get(pieceIndex);
        char pieceId = piece.getId();

        String direction = move.getDirection(piece.isVertical());
        int distance = move.getDistance(piece.isVertical());

        BoardPrinter.printBoardAfterMove(node.board, i + 1, pieceId, direction, distance);
    }
}

private static class Node {
    Board board;
    Node parent;
    Move move;
    int cost;
    int heuristic;
```

```
        Node(Board board, Node parent, Move move, int cost, int heuristic) {
            this.board = board;
            this.parent = parent;
            this.move = move;
            this.cost = cost;
            this.heuristic = heuristic;
        }
    }

    public int getNodesVisited(){
        return this.nodesVisited;
    }
}
```

IDAStar.java

```
package algorithm;

import java.util.*;
import model.Board;
import model.Move;
import model.Piece;
import util.BoardPrinter;

public class IDAStar {
    private int nodesVisited = 0;
    private int heuristicType;
    private List<Node> solutionPath;
    private gui.Gui.SolutionCollector collector;

    public static final int BLOCKING_PIECES = 1;
    public static final int MANHATTAN_DISTANCE = 2;
    public static final int COMBINED = 3;

    public IDAStar(int heuristicType, gui.Gui.SolutionCollector collector) {
        this.heuristicType = heuristicType;
        this.collector = collector;
    }

    public IDAStar() {
        this(BLOCKING_PIECES, null);
    }

    public void solve(Board initialBoard) {
        long startTime = System.currentTimeMillis();

        System.out.println("Using heuristic: " + getHeuristicName());

        BoardPrinter.printInitialBoard(initialBoard);

        int initialHeuristic = calculateHeuristic(initialBoard);
        Node root = new Node(initialBoard, null, null, 0, initialHeuristic);

        int threshold = initialHeuristic;

        boolean solved = false;
        int solutionCost = 0;

        Set<String> globalVisited = new HashSet<>();

        while (!solved) {
            Set<String> visited = new HashSet<>();
            visited.add(initialBoard.toString());
            globalVisited.add(initialBoard.toString());
            nodesVisited++;
```

```
            solutionPath = new ArrayList<>();

            DFSResult result = depthFirstSearch(root, 0, threshold, visited, globalVisited);

            if (result.solved) {
                solved = true;
                solutionCost = result.cost;
            } else {
                threshold = result.nextThreshold;

                if (threshold == Integer.MAX_VALUE) {
                    break;
                }

                System.out.println("Increasing threshold to: " + threshold);
            }
        }

        long endTime = System.currentTimeMillis();
        double executionTime = (endTime - startTime) / 1000.0;

        if (solved && collector != null) {
            collector.addStep(initialBoard);

            for (Node node : solutionPath) {
                collector.addStep(node.board);
            }
        }

        if (solved) {
            Collections.reverse(solutionPath);
            printSolution(solutionPath);

            System.out.println("Jumlah langkah: " + solutionCost);
            System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
            System.out.println("Waktu eksekusi: " + executionTime + " detik");
        } else {
            System.out.println("Tidak ada solusi yang ditemukan!");
            System.out.println("Jumlah node yang diperiksa: " + nodesVisited);
            System.out.println("Waktu eksekusi: " + executionTime + " detik");
        }
    }

    private DFSResult depthFirstSearch(Node node, int cost, int threshold,
                                       Set<String> visited, Set<String> globalVisited) {
        int f = cost + node.heuristic;

        if (f > threshold) {
            return new DFSResult(false, 0, f);
        }

        if (node.board.isSolved()) {
            Node current = node;
            while (current.parent != null) {
                solutionPath.add(current);
                current = current.parent;
            }

            return new DFSResult(true, cost, f);
        }

        int min = Integer.MAX_VALUE;

        List<Board> nextStates = node.board.getNextStates();

        for (Board nextBoard : nextStates) {
            String boardString = nextBoard.toString();

            if (visited.contains(boardString)) {
```

```
                continue;
            }

            boolean isNewGlobal = !globalVisited.contains(boardString);

            visited.add(boardString);
            globalVisited.add(boardString);
            nodesVisited++;

            Move move = findMove(node.board, nextBoard);

            int heuristic = calculateHeuristic(nextBoard);

            Node childNode = new Node(nextBoard, node, move, cost + 1, heuristic);

            DFSResult result = depthFirstSearch(childNode, cost + 1, threshold, visited,
globalVisited);

            if (result.solved) {
                return result;
            }

            min = Math.min(min, result.nextThreshold);

            visited.remove(boardString);

            if (isNewGlobal && !result.solved) {
                globalVisited.remove(boardString);
            }
        }

        return new DFSResult(false, 0, min);
    }

    private String getHeuristicName() {
        switch (heuristicType) {
            case BLOCKING_PIECES:
                return "Blocking Pieces";
            case MANHATTAN_DISTANCE:
                return "Manhattan Distance";
            case COMBINED:
                return "Combined (Blocking + Manhattan)";
            default:
                return "Unknown";
        }
    }

    private int calculateHeuristic(Board board) {
        switch (heuristicType) {
            case MANHATTAN_DISTANCE:
                return calculateManhattanHeuristic(board);
            case COMBINED:
                return calculateCombinedHeuristic(board);
            case BLOCKING_PIECES:
            default:
                return calculateBlockingPiecesHeuristic(board);
        }
    }

    private int calculateBlockingPiecesHeuristic(Board board) {
        Piece primaryPiece = board.getPrimaryPiece();
        if (primaryPiece == null) {
            return Integer.MAX_VALUE;
        }

        int exitRow = board.getExitRow();
        int exitCol = board.getExitCol();
        char[][] grid = board.getGrid();
        int rows = board.getRows();
        int cols = board.getCols();
```

```
        int blockingPieces = 0;

        if (primaryPiece.isHorizontal()) {
            if (exitCol == cols) {
                int row = primaryPiece.getRow();
                for (int col = primaryPiece.getCol() + primaryPiece.getLength(); col < cols;
col++) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            } else if (exitCol == -1) {
                int row = primaryPiece.getRow();
                for (int col = primaryPiece.getCol() - 1; col >= 0; col--) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            }
        } else {
            if (exitRow == rows) {
                int col = primaryPiece.getCol();
                for (int row = primaryPiece.getRow() + primaryPiece.getLength(); row < rows;
row++) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            } else if (exitRow == -1) {
                int col = primaryPiece.getCol();
                for (int row = primaryPiece.getRow() - 1; row >= 0; row--) {
                    if (grid[row][col] != '.' && grid[row][col] != 'K') {
                        blockingPieces++;
                    }
                }
            }
        }

        return blockingPieces;
    }

    private int calculateManhattanHeuristic(Board board) {
        Piece primaryPiece = board.getPrimaryPiece();
        if (primaryPiece == null) {
            return Integer.MAX_VALUE;
        }

        int exitRow = board.getExitRow();
        int exitCol = board.getExitCol();
        int rows = board.getRows();
        int cols = board.getCols();

        int manhattanDistance = 0;

        if (primaryPiece.isHorizontal()) {
            if (exitCol == cols) {
                manhattanDistance = cols - (primaryPiece.getCol() +
primaryPiece.getLength());
            } else if (exitCol == -1) {
                manhattanDistance = primaryPiece.getCol();
            }
        } else {
            if (exitRow == rows) {
                manhattanDistance = rows - (primaryPiece.getRow() +
primaryPiece.getLength());
            } else if (exitRow == -1) {
                manhattanDistance = primaryPiece.getRow();
            }
        }
```

```
        return manhattanDistance;
    }

    private int calculateCombinedHeuristic(Board board) {
        int blockingPieces = calculateBlockingPiecesHeuristic(board);
        int manhattanDistance = calculateManhattanHeuristic(board);

        return blockingPieces + manhattanDistance;
    }

    private Move findMove(Board from, Board to) {
        List<Piece> fromPieces = from.getPieces();
        List<Piece> toPieces = to.getPieces();

        for (int i = 0; i < fromPieces.size(); i++) {
            Piece fromPiece = fromPieces.get(i);
            Piece toPiece = toPieces.get(i);

            if (fromPiece.getRow() != toPiece.getRow() ||
                fromPiece.getCol() != toPiece.getCol()) {
                return new Move(
                    i,
                    fromPiece.getRow(),
                    fromPiece.getCol(),
                    toPiece.getRow(),
                    toPiece.getCol()
                );
            }
        }

        throw new IllegalStateException("Could not find the move between board states");
    }

    private void printSolution(List<Node> path) {
        for (int i = 0; i < path.size(); i++) {
            Node node = path.get(i);
            Move move = node.move;
            int pieceIndex = move.getPieceIndex();
            Piece piece = node.board.getPieces().get(pieceIndex);
            char pieceId = piece.getId();

            String direction = move.getDirection(piece.isVertical());
            int distance = move.getDistance(piece.isVertical());

            BoardPrinter.printBoardAfterMove(node.board, i + 1, pieceId, direction,
distance);
        }
    }

    private static class Node {
        Board board;
        Node parent;
        Move move;
        int cost;
        int heuristic;

        Node(Board board, Node parent, Move move, int cost, int heuristic) {
            this.board = board;
            this.parent = parent;
            this.move = move;
            this.cost = cost;
            this.heuristic = heuristic;
        }
    }

    private static class DFSResult {
        boolean solved;
        int cost;
        int nextThreshold;
```

```
        DFSResult(boolean solved, int cost, int nextThreshold) {
            this.solved = solved;
            this.cost = cost;
            this.nextThreshold = nextThreshold;
        }
    }

    public int getNodesVisited(){
        return this.nodesVisited;
    }
}
```

2.3 Utility

FileParser.java

```
package util;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import model.Board;
import model.Piece;

public class FileParser {
    /* File parser */
    public Board parseFile(String filename) throws IOException, IllegalArgumentException {
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader(filename));

            String dimensionLine = reader.readLine();
            if (dimensionLine == null || dimensionLine.trim().isEmpty()) {
                throw new IllegalArgumentException("Input file is empty.");
            }

            String[] dimensions = dimensionLine.split("\\s+");
            if (dimensions.length < 2) {
                throw new IllegalArgumentException("Invalid board dimensions format. Expected:
rows cols");
            }

            int rows, cols;
            try {
                rows = Integer.parseInt(dimensions[0]);
                cols = Integer.parseInt(dimensions[1]);
            } catch (NumberFormatException e) {
                throw new IllegalArgumentException("Invalid board dimensions. Expected numbers,
got: " + dimensionLine);
            }

            if (rows <= 0 || cols <= 0) {
                throw new IllegalArgumentException("Board dimensions must be positive. Got: " +
rows + "x" + cols);
            }

            if (rows < 2 || cols < 2) {
                throw new IllegalArgumentException("Board is too small. Minimum size is 2x2.");
```

```
            }

            String numPiecesLine = reader.readLine();
            if (numPiecesLine == null || numPiecesLine.trim().isEmpty()) {
                throw new IllegalArgumentException("Missing number of pieces.");
            }

            int numPieces;
            try {
                numPieces = Integer.parseInt(numPiecesLine);
            } catch (NumberFormatException e) {
                throw new IllegalArgumentException("Invalid number of pieces. Expected a
number, got: " + numPiecesLine);
            }

            if (numPieces <= 0) {
                throw new IllegalArgumentException("Number of pieces must be positive.");
            }

            int maxPieces = (rows * cols) / 2;
            if (numPieces > maxPieces) {
                throw new IllegalArgumentException("Too many pieces (" + numPieces + ") for
this board size.");
            }

            Board board = new Board(rows, cols);

            ArrayList<String> allLines = new ArrayList<>();
            String line;
            while ((line = reader.readLine()) != null) {
                if (!line.trim().isEmpty()) {
                    allLines.add(line);
                }
            }

            if (allLines.size() < rows) {
                throw new IllegalArgumentException("Not enough rows in the puzzle. Expected at
least " + rows + " rows.");
            }

            int exitRow = -1;
            int exitCol = -1;
            boolean exitFound = false;
            char[][] grid = new char[rows][cols];

            int exitCount = 0;

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

            if (allLines.size() > rows) {
                String firstLine = allLines.get(0);
                if (firstLine.contains("K")) {
                    exitRow = -1;
                    exitCol = firstLine.indexOf('K');
                    exitFound = true;
                    exitCount++;
                    board.setExit(exitRow, exitCol);

                    if (exitCol < 0 || exitCol >= cols) {
                        throw new IllegalArgumentException("Top exit position is outside the
board boundary.");
                    }
```

```
                    allLines.remove(0);
                }
                else if (allLines.size() > rows) {
                    String lastLine = allLines.get(rows);
                    if (lastLine.contains("K")) {
                        exitRow = rows;
                        exitCol = lastLine.indexOf('K');
                        exitFound = true;
                        exitCount++;
                        board.setExit(exitRow, exitCol);

                        if (exitCol < 0 || exitCol >= cols) {
                            throw new IllegalArgumentException("Bottom exit position is outside
the board boundary.");
                        }
                    }
                }
            }

            for (int r = 0; r < rows && r < allLines.size(); r++) {
                line = allLines.get(r);

                if (line.length() > 0 && line.charAt(0) == 'K') {
                    exitRow = r;
                    exitCol = -1;
                    exitFound = true;
                    exitCount++;
                    board.setExit(exitRow, exitCol);

                    if (exitRow < 0 || exitRow >= rows) {
                        throw new IllegalArgumentException("Left exit position is outside the
board boundary.");
                    }

                    line = line.substring(1);
                }

                if (line.length() > cols && line.charAt(cols) == 'K') {
                    exitRow = r;
                    exitCol = cols;
                    exitFound = true;
                    exitCount++;
                    board.setExit(exitRow, exitCol);

                    if (exitRow < 0 || exitRow >= rows) {
                        throw new IllegalArgumentException("Right exit position is outside the
board boundary.");
                    }
                }

                int startIdx = (line.length() > 0 && line.charAt(0) == ' ') ? 1 : 0;
                for (int c = 0; c < cols && startIdx + c < line.length(); c++) {
                    grid[r][c] = line.charAt(startIdx + c);

                    if (grid[r][c] == 'K' && !exitFound) {
                        exitRow = r;
                        exitCol = c;
                        exitFound = true;
                        exitCount++;
                        board.setExit(exitRow, exitCol);
                        grid[r][c] = '.';
                    }
                }
            }
```

```
            if (!exitFound) {
                throw new IllegalArgumentException("No exit (K) found in the puzzle.");
            }

            if (exitCount>1){
                throw new IllegalArgumentException("Multiple exits (" + exitCount + ") found in
puzzle. Only one exit is allowed.");
            }

            Set<Character> processedChars = new HashSet<>();
            processedChars.add('.');
            processedChars.add('K');
            processedChars.add(' ');

            boolean foundPrimaryPiece = false;

            for (int r = 0; r < rows; r++) {
                for (int c = 0; c < cols; c++) {
                    char pieceChar = grid[r][c];

                    if (pieceChar == '.' || pieceChar == ' ' ||
processedChars.contains(pieceChar)) {
                        continue;
                    }

                    processedChars.add(pieceChar);
                    boolean isPrimary = (pieceChar == 'P');

                    if (isPrimary) {
                        foundPrimaryPiece = true;
                    }

                    int length = 1;
                    boolean isHorizontal = false;

                    for (int i = c + 1; i < cols; i++) {
                        if (grid[r][i] == pieceChar) {
                            length++;
                            isHorizontal = true;
                        } else {
                            break;
                        }
                    }

                    if (length == 1) {
                        for (int i = r + 1; i < rows; i++) {
                            if (grid[i][c] == pieceChar) {
                                length++;
                            } else {
                                break;
                            }
                        }
                    }

                    if (length < 2) {
                        throw new IllegalArgumentException("Piece '" + pieceChar + "' has
invalid length. Minimum length is 2.");
                    }

                    Piece piece = new Piece(pieceChar, r, c, length, !isHorizontal, isPrimary);
                    board.addPiece(piece);
                }
            }

            if (!foundPrimaryPiece) {
                throw new IllegalArgumentException("No primary piece (P) found in the
```

```
puzzle.");
            }

            int nonPrimaryPieceCount = 0;
            for (Piece piece : board.getPieces()) {
                if (piece.getId() != 'P') {
                    nonPrimaryPieceCount++;
                }
            }

            if (nonPrimaryPieceCount != numPieces) {
                throw new IllegalArgumentException("Expected " + numPieces + " non-primary
pieces, but found " +
                                                   nonPrimaryPieceCount + ".");
            }

            validatePrimaryPieceExitAlignment(board, exitRow, exitCol, rows, cols);

            return board;

        } finally {
            if (reader != null) {
                reader.close();
            }
        }
    }

    /* Validasi primary dan exit */
    private void validatePrimaryPieceExitAlignment(Board board, int exitRow, int exitCol, int
rows, int cols) {
        Piece primaryPiece = null;
        for (Piece piece : board.getPieces()) {
            if (piece.getId() == 'P') {
                primaryPiece = piece;
                break;
            }
        }

        if (primaryPiece == null) {
            return;
        }

        boolean isAligned = false;
        String exitPosition = "";

        boolean isHorizontal = !primaryPiece.isVertical();
        int pRow = primaryPiece.getRow();
        int pCol = primaryPiece.getCol();

        if (exitRow == -1) {
            exitPosition = "top";
            isAligned = !isHorizontal;
        } else if (exitRow == rows) {
            exitPosition = "bottom";
            isAligned = !isHorizontal;
        } else if (exitCol == -1) {
            exitPosition = "left";
            isAligned = isHorizontal;
        } else if (exitCol == cols) {
            exitPosition = "right";
            isAligned = isHorizontal;
        }

        if (!isAligned) {
            throw new IllegalArgumentException(
                "Primary piece has incorrect orientation for " + exitPosition + " exit. " +
```

```
                "For " + exitPosition + " exit, primary piece must be " +
                (!isHorizontal ? "horizontal." : "vertical.")
            );
        }

        if ((exitRow == -1 || exitRow == rows) && !isHorizontal) {
            if (exitCol != pCol) {
                throw new IllegalArgumentException(
                    "Primary piece is not aligned with the " + exitPosition + " exit. " +
                    "Exit column (" + exitCol + ") must match primary piece's column (" + pCol
+ ")."
                );
            }
        } else if ((exitCol == -1 || exitCol == cols) && isHorizontal) {
            if (exitRow != pRow) {
                throw new IllegalArgumentException(
                    "Primary piece is not aligned with the " + exitPosition + " exit. " +
                    "Exit row (" + exitRow + ") must match primary piece's row (" + pRow + ")."
                );
            }
        }
    }
}
```

BoardPrinter.java

```
package util;

import model.Board;

public class BoardPrinter {
    private static final String RESET = "\u001B[0m";
    private static final String RED = "\u001B[31m";
    private static final String GREEN = "\u001B[32m";
    private static final String YELLOW = "\u001B[33m";

    /* Print board awal */
    public static void printInitialBoard(Board board) {
        System.out.println("\nInitial Board:");
        printBoardWithBorders(board, ' ');
    }

    public static void printBoardAfterMove(Board board, int moveCount, char pieceId, String
direction) {
        System.out.println("\nMove " + moveCount + ": " + pieceId + " " + direction);
        printBoardWithBorders(board, pieceId);
    }

    public static void printBoardAfterMove(Board board, int moveCount, char pieceId, String
direction, int distance) {
        if (distance > 1) {
            System.out.println("\nMove " + moveCount + ": " + pieceId + " " + direction + " " +
distance + " cells");
        } else {
            System.out.println("\nMove " + moveCount + ": " + pieceId + " " + direction);
        }
        printBoardWithBorders(board, pieceId);
    }

    private static void printBoardWithBorders(Board board, char movedPiece) {
        int rows = board.getRows();
        int cols = board.getCols();
        char[][] grid = board.getGrid();
        int exitRow = board.getExitRow();
```

```
        int exitCol = board.getExitCol();

        boolean needTopBorder = (exitRow == -1);
        boolean needBottomBorder = (exitRow == rows);
        boolean needLeftBorder = (exitCol == -1);
        boolean needRightBorder = (exitCol == cols);

        System.out.print("+");
        for (int c = 0; c < cols; c++) {
            if (needTopBorder && c == exitCol) {
                System.out.print(GREEN + "K" + RESET);
            } else {
                System.out.print("-");
            }
        }
        System.out.println("+");

        for (int r = 0; r < rows; r++) {
            if (needLeftBorder && r == exitRow) {
                System.out.print(GREEN + "K" + RESET);
            } else {
                System.out.print("|");
            }

            for (int c = 0; c < cols; c++) {
                char cell = grid[r][c];

                if (cell == 'P') {
                    // Primary car warna merah
                    System.out.print(RED + cell + RESET);
                } else if (cell != '.' && cell == movedPiece) {
                    // Mobil yg gerak warna kuning
                    System.out.print(YELLOW + cell + RESET);
                } else {
                    System.out.print(cell);
                }
            }

            if (needRightBorder && r == exitRow) {
                System.out.print(GREEN + "K" + RESET);
            } else {
                System.out.print("|");
            }

            System.out.println();
        }

        System.out.print("+");
        for (int c = 0; c < cols; c++) {
            if (needBottomBorder && c == exitCol) {
                System.out.print(GREEN + "K" + RESET);
            } else {
                System.out.print("-");
            }
        }
        System.out.println("+");
    }
}
```

2.4 Gui

```
Gui.java
```

```
package gui;

import javax.swing.*;
import javax.swing.filechooser.FileNameExtensionFilter;
import java.awt.*;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicBoolean;

import algorithm.*;
import model.Board;
import model.Move;
import model.Piece;
import util.FileParser;

public class Gui extends JFrame {
    private JPanel boardPanel;
    private JButton loadButton;
    private JButton solveButton;
    private JButton createBoardButton;
    private JButton saveButton;
    private JComboBox<String> algorithmSelector;
    private JComboBox<String> heuristicSelector;
    private JLabel statusLabel;
    private JSlider animationSpeedSlider;
    private JButton playPauseButton;
    private JButton stopButton;
    private JButton stepButton;

    private Board currentBoard;
    private List<Board> solutionSteps;
    private List<Move> solutionMoves;
    private int currentStep = 0;
    private Timer animationTimer;
    private AtomicBoolean animationRunning = new AtomicBoolean(false);

    private boolean showExitAnimation = false;
    private boolean showingFinalState = false;

    private int nodesVisited = 0;
    private double executionTime = 0;

    private Map<Character, Color> pieceColors = new HashMap<>();

    public Gui() {
        super("Rush Hour Puzzle Solver");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(800, 600);
        setLayout(new BorderLayout());

        createTopPanel();
        createBoardPanel();
        createBottomPanel();

        animationTimer = new Timer(500, e -> {
            if (currentStep < solutionSteps.size() - 1) {
                currentStep++;

                if (currentStep == solutionSteps.size() - 1) {
                    showingFinalState = true;
```

```
                    showExitAnimation = false;
                    updateBoardDisplay();

                    Timer exitAnimationTimer = new Timer(800, evt -> {
                        ((Timer)evt.getSource()).stop();
                        showingFinalState = false;
                        showExitAnimation = true;
                        updateBoardDisplay();

                        Timer completionTimer = new Timer(1000, evt2 -> {
                            ((Timer)evt2.getSource()).stop();
                            animationTimer.stop();
                            animationRunning.set(false);
                            playPauseButton.setText("Play");
                            statusLabel.setText("Solution complete in " + (solutionSteps.size()
- 1) + " steps!");
                        });
                        completionTimer.setRepeats(false);
                        completionTimer.start();
                    });
                    exitAnimationTimer.setRepeats(false);
                    exitAnimationTimer.start();
                } else {
                    showExitAnimation = false;
                    showingFinalState = false;
                    updateBoardDisplay();
                }

                statusLabel.setText("Step " + currentStep + " of " + (solutionSteps.size() -
1));
            }
        });

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    private void createTopPanel() {
        JPanel topPanel = new JPanel(new BorderLayout());

        JPanel filePanel = new JPanel();
        loadButton = new JButton("Load Puzzle");
        loadButton.addActionListener(e -> loadPuzzleFromFile());

        createBoardButton = new JButton("Create Puzzle");
        createBoardButton.addActionListener(e -> openBoardEditor());

        saveButton = new JButton("Save Solution");
        saveButton.addActionListener(e -> saveSolutionToFile());
        saveButton.setEnabled(false);

        filePanel.add(loadButton);
        filePanel.add(createBoardButton);
        filePanel.add(saveButton);

        JPanel algoPanel = new JPanel();
        algorithmSelector = new JComboBox<>(new String[] {
            "Uniform Cost Search (UCS)",
            "Greedy Best-First Search (GBFS)",
            "A* Search",
            "IDA* Search"
        });

        heuristicSelector = new JComboBox<>(new String[] {
            "Blocking Pieces",
```

```
            "Manhattan Distance",
            "Combined"
        });
        heuristicSelector.setEnabled(false);

        algorithmSelector.addActionListener(e -> {
            int selectedIndex = algorithmSelector.getSelectedIndex();
            heuristicSelector.setEnabled(selectedIndex != 0);
        });

        solveButton = new JButton("Solve");
        solveButton.addActionListener(e -> solvePuzzle());
        solveButton.setEnabled(false);

        algoPanel.add(new JLabel("Algorithm:"));
        algoPanel.add(algorithmSelector);
        algoPanel.add(new JLabel("Heuristic:"));
        algoPanel.add(heuristicSelector);
        algoPanel.add(solveButton);

        topPanel.add(filePanel, BorderLayout.NORTH);
        topPanel.add(algoPanel, BorderLayout.CENTER);

        add(topPanel, BorderLayout.NORTH);
    }

    private void createBoardPanel() {
        boardPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                drawBoard(g);
            }
        };
        boardPanel.setPreferredSize(new Dimension(600, 600));
        boardPanel.setBackground(Color.WHITE);

        add(boardPanel, BorderLayout.CENTER);
    }

    private void createBottomPanel() {
        JPanel bottomPanel = new JPanel(new BorderLayout());

        JPanel controlPanel = new JPanel();
        playPauseButton = new JButton("Play");
        playPauseButton.addActionListener(e -> togglePlayPause());
        playPauseButton.setEnabled(false);

        stopButton = new JButton("Stop");
        stopButton.addActionListener(e -> stopAnimation());
        stopButton.setEnabled(false);

        stepButton = new JButton("Step");
        stepButton.addActionListener(e -> stepForward());
        stepButton.setEnabled(false);

        animationSpeedSlider = new JSlider(JSlider.HORIZONTAL, 1, 10, 5);
        animationSpeedSlider.setMajorTickSpacing(1);
        animationSpeedSlider.setPaintTicks(true);
        animationSpeedSlider.setPaintLabels(true);
        animationSpeedSlider.setSnapToTicks(true);
        animationSpeedSlider.addChangeListener(e -> {
            if (!animationSpeedSlider.getValueIsAdjusting()) {
                updateAnimationSpeed();
            }
        });
```

```
        controlPanel.add(new JLabel("Speed:"));
        controlPanel.add(animationSpeedSlider);
        controlPanel.add(playPauseButton);
        controlPanel.add(stepButton);
        controlPanel.add(stopButton);

        statusLabel = new JLabel("Ready to load or create a puzzle");
        JPanel statusPanel = new JPanel();
        statusPanel.add(statusLabel);

        bottomPanel.add(controlPanel, BorderLayout.CENTER);
        bottomPanel.add(statusPanel, BorderLayout.SOUTH);

        add(bottomPanel, BorderLayout.SOUTH);
    }

    // Load puzzle dari file
    private void loadPuzzleFromFile() {
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setCurrentDirectory(new File("test/input"));
        fileChooser.setFileFilter(new FileNameExtensionFilter("Text Files", "txt"));

        int result = fileChooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fileChooser.getSelectedFile();
            try {
                FileParser parser = new FileParser();
                currentBoard = parser.parseFile(selectedFile.getAbsolutePath());

                solutionSteps = null;
                solutionMoves = null;
                currentStep = 0;
                showExitAnimation = false;
                showingFinalState = false;

                updateBoardDisplay();
                solveButton.setEnabled(true);
                playPauseButton.setEnabled(false);
                stopButton.setEnabled(false);
                stepButton.setEnabled(false);
                saveButton.setEnabled(false);

                statusLabel.setText("Puzzle loaded from " + selectedFile.getName());

                assignColorsToNewPieces();

            } catch (IOException ex) {
                showErrorDialog("File Error",
                    "Could not read the file: " + ex.getMessage());
            } catch (IllegalArgumentException ex) {
                showErrorDialog("File Format Error", ex.getMessage());
            } catch (Exception ex) {
                showErrorDialog("Unexpected Error",
                    "An unexpected error occurred: " + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }

    private void showErrorDialog(String title, String message) {
        JOptionPane.showMessageDialog(this,
            message,
            title,
            JOptionPane.ERROR_MESSAGE);
    }
```

```
private void openBoardEditor() {
    BoardEditor editor = new BoardEditor(this);
    editor.setVisible(true);
}

// Save ke file txt
private void saveSolutionToFile() {
    if (solutionSteps == null || solutionSteps.size() <= 1) {
        JOptionPane.showMessageDialog(this,
            "No solution to save.",
            "Save Error",
            JOptionPane.ERROR_MESSAGE);
        return;
    }

    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setCurrentDirectory(new File("test/output"));
    fileChooser.setFileFilter(new FileNameExtensionFilter("Text Files", "txt"));

    int result = fileChooser.showSaveDialog(this);
    if (result == JFileChooser.APPROVE_OPTION) {
        File selectedFile = fileChooser.getSelectedFile();

        if (!selectedFile.getName().toLowerCase().endsWith(".txt")) {
            selectedFile = new File(selectedFile.getAbsolutePath() + ".txt");
        }

        try (PrintWriter writer = new PrintWriter(new FileWriter(selectedFile))) {
            writer.println("Rush Hour Puzzle Solution");
            writer.println("========================");
            writer.println("Algorithm: " + algorithmSelector.getSelectedItem());
            if (algorithmSelector.getSelectedIndex() != 0) {
                writer.println("Heuristic: " + heuristicSelector.getSelectedItem());
            }
            writer.println("Total steps: " + (solutionSteps.size() - 1));
            writer.println("Nodes visited: " + nodesVisited);
            writer.println("Execution time: " + executionTime + " seconds");
            writer.println();

            writer.println("Initial Board:");
            writer.println(solutionSteps.get(0).toString());
            writer.println();

            for (int i = 1; i < solutionSteps.size(); i++) {
                Move move = solutionMoves.get(i-1);
                Board board = solutionSteps.get(i);
                Piece piece = board.getPieces().get(move.getPieceIndex());
                char pieceId = piece.getId();

                String direction = move.getDirection(piece.isVertical());
                int distance = move.getDistance(piece.isVertical());

                writer.println("Step " + i + ": Move piece " + pieceId +
                               " " + distance + " cell(s) " + direction);
                writer.println(board.toString());
                writer.println();
            }

            statusLabel.setText("Solution saved to " + selectedFile.getName());
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this,
                "Error saving file: " + ex.getMessage(),
                "Save Error",
                JOptionPane.ERROR_MESSAGE);
            ex.printStackTrace();
```

```
        }
    }
}

private void solvePuzzle() {
if (currentBoard == null) {
    return;
}

int algorithmIndex = algorithmSelector.getSelectedIndex();
int heuristicIndex = heuristicSelector.getSelectedIndex() + 1;

solutionSteps = null;
solutionMoves = null;
currentStep = 0;
nodesVisited = 0;
executionTime = 0;
showExitAnimation = false;
showingFinalState = false;

setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
solveButton.setEnabled(false);
statusLabel.setText("Solving puzzle...");

new SwingWorker<List<Board>, Void>() {
    @Override
    protected List<Board> doInBackground() throws Exception {
        SolutionCollector collector = new SolutionCollector();

        long startTime = System.currentTimeMillis();

        switch (algorithmIndex) {
            case 0: // UCS
                UCS ucs = new UCS(collector);
                ucs.solve(currentBoard);
                nodesVisited = ucs.getNodesVisited();
                break;
            case 1: // GBFS
                GBFS gbfs = new GBFS(heuristicIndex, collector);
                gbfs.solve(currentBoard);
                nodesVisited = gbfs.getNodesVisited();
                break;
            case 2: // A*
                AStar aStar = new AStar(heuristicIndex, collector);
                aStar.solve(currentBoard);
                nodesVisited = aStar.getNodesVisited();
                break;
            case 3: // IDA*
                IDAStar idaStar = new IDAStar(heuristicIndex, collector);
                idaStar.solve(currentBoard);
                nodesVisited = idaStar.getNodesVisited();
                break;
        }

        long endTime = System.currentTimeMillis();
        executionTime = (endTime - startTime) / 1000.0;

        return collector.getSolutionSteps();
    }

        @Override
        protected void done() {
            try {
                solutionSteps = get();

                if (solutionSteps != null && solutionSteps.size() > 1) {
```

```
                    solutionMoves = extractMoves(solutionSteps);
                    currentStep = 0;
                    updateBoardDisplay();

                    playPauseButton.setEnabled(true);
                    stopButton.setEnabled(true);
                    stepButton.setEnabled(true);
                    saveButton.setEnabled(true);

                    String algorithmName = (String) algorithmSelector.getSelectedItem();
                    statusLabel.setText("Solution found in " + (solutionSteps.size() - 1) +
                                        " steps using " + algorithmName +
                                        " | Nodes visited: " + nodesVisited +
                                        " | Time: " + executionTime + " sec");
                } else {
                    statusLabel.setText("No solution found!");
                    saveButton.setEnabled(false);
                }

            } catch (Exception ex) {
                JOptionPane.showMessageDialog(Gui.this,
                    "Error solving puzzle: " + ex.getMessage(),
                    "Solver Error",
                    JOptionPane.ERROR_MESSAGE);
                statusLabel.setText("Error solving puzzle");
                ex.printStackTrace();
            } finally {
                setCursor(Cursor.getDefaultCursor());
                solveButton.setEnabled(true);
            }
        }
    }.execute();
}

// buat play pause
private void togglePlayPause() {
    if (solutionSteps == null || solutionSteps.size() <= 1) {
        return;
    }

    if (currentStep >= solutionSteps.size() - 1 && showExitAnimation) {
        showExitAnimation = false;
        showingFinalState = false;
        currentStep = 0;
        updateBoardDisplay();
    }

    if (animationRunning.get()) {
        animationTimer.stop();
        animationRunning.set(false);
        playPauseButton.setText("Play");
    } else {
        updateAnimationSpeed();
        animationTimer.start();
        animationRunning.set(true);
        playPauseButton.setText("Pause");
    }
}

// Memberhentikan animasi
private void stopAnimation() {
    if (animationRunning.get()) {
        animationTimer.stop();
        animationRunning.set(false);
        playPauseButton.setText("Play");
    }
```

```
        showExitAnimation = false;
        showingFinalState = false;
        currentStep = 0;
        updateBoardDisplay();
        statusLabel.setText("Animation stopped");
    }

    private void stepForward() {
        if (solutionSteps != null && currentStep < solutionSteps.size() - 1) {
            currentStep++;

            if (currentStep == solutionSteps.size() - 1) {
                showingFinalState = true;
                showExitAnimation = false;
                updateBoardDisplay();

                Timer exitAnimTimer = new Timer(800, e -> {
                    ((Timer)e.getSource()).stop();
                    showingFinalState = false;
                    showExitAnimation = true;
                    updateBoardDisplay();
                });
                exitAnimTimer.setRepeats(false);
                exitAnimTimer.start();
            } else {
                showExitAnimation = false;
                showingFinalState = false;
                updateBoardDisplay();
            }

            statusLabel.setText("Step " + currentStep + " of " + (solutionSteps.size() - 1));
        }
    }

    // Kecepatan animasi dgn slider
    private void updateAnimationSpeed() {
        int delay = 1100 - (animationSpeedSlider.getValue() * 100);
        animationTimer.setDelay(delay);
    }

    private void updateBoardDisplay() {
        if (currentBoard == null) {
            return;
        }

        boardPanel.repaint();
    }

    // Gambar papan
    private void drawBoard(Graphics g) {
        if (currentBoard == null) {
            return;
        }

        Board boardToDraw = (solutionSteps != null && currentStep < solutionSteps.size()) ?
                            solutionSteps.get(currentStep) : currentBoard;

        int rows = boardToDraw.getRows();
        int cols = boardToDraw.getCols();

        int cellSize = Math.min(
            (boardPanel.getWidth() - 120) / cols,
            (boardPanel.getHeight() - 120) / rows
        );
```

```
        int boardWidth = cols * cellSize;
        int boardHeight = rows * cellSize;
        int startX = (boardPanel.getWidth() - boardWidth) / 2;
        int startY = (boardPanel.getHeight() - boardHeight) / 2;

        g.setColor(Color.LIGHT_GRAY);
        for (int r = 0; r <= rows; r++) {
            g.drawLine(startX, startY + r * cellSize, startX + cols * cellSize, startY + r *
cellSize);
        }
        for (int c = 0; c <= cols; c++) {
            g.drawLine(startX + c * cellSize, startY, startX + c * cellSize, startY + rows *
cellSize);
        }

        int exitRow = boardToDraw.getExitRow();
        int exitCol = boardToDraw.getExitCol();

        g.setColor(Color.GREEN);
        if (exitRow == -1) {
            g.fillRect(startX + exitCol * cellSize, startY - cellSize, cellSize, cellSize);
            int[] xPoints = {startX + exitCol * cellSize + cellSize/2, startX + exitCol *
cellSize + cellSize/4, startX + exitCol * cellSize + 3*cellSize/4};
            int[] yPoints = {startY - cellSize - 10, startY - cellSize, startY - cellSize};
            g.fillPolygon(xPoints, yPoints, 3);
        } else if (exitRow == rows) {
            g.fillRect(startX + exitCol * cellSize, startY + rows * cellSize, cellSize,
cellSize);
            int[] xPoints = {startX + exitCol * cellSize + cellSize/2, startX + exitCol *
cellSize + cellSize/4, startX + exitCol * cellSize + 3*cellSize/4};
            int[] yPoints = {startY + rows * cellSize + cellSize + 10, startY + rows * cellSize
+ cellSize, startY + rows * cellSize + cellSize};
            g.fillPolygon(xPoints, yPoints, 3);
        } else if (exitCol == -1) {
            g.fillRect(startX - cellSize, startY + exitRow * cellSize, cellSize, cellSize);
            int[] xPoints = {startX - cellSize - 10, startX - cellSize, startX - cellSize};
            int[] yPoints = {startY + exitRow * cellSize + cellSize/2, startY + exitRow *
cellSize + cellSize/4, startY + exitRow * cellSize + 3*cellSize/4};
            g.fillPolygon(xPoints, yPoints, 3);
        } else if (exitCol == cols) {
            g.fillRect(startX + cols * cellSize, startY + exitRow * cellSize, cellSize,
cellSize);
            int[] xPoints = {startX + cols * cellSize + cellSize + 10, startX + cols * cellSize
+ cellSize, startX + cols * cellSize + cellSize};
            int[] yPoints = {startY + exitRow * cellSize + cellSize/2, startY + exitRow *
cellSize + cellSize/4, startY + exitRow * cellSize + 3*cellSize/4};
            g.fillPolygon(xPoints, yPoints, 3);
        }

        for (Piece piece : boardToDraw.getPieces()) {
            int row = piece.getRow();
            int col = piece.getCol();
            int length = piece.getLength();
            boolean isVertical = piece.isVertical();
            char id = piece.getId();

            if (showExitAnimation && !showingFinalState && id == 'P' &&
                solutionSteps != null && currentStep == solutionSteps.size() - 1) {
                System.out.println("Skipping primary piece (P) for vanishing effect!");
                continue;
            }

            Color pieceColor = getPieceColor(id);

            boolean isPrimary = (id == 'P');
            boolean isLastStep = (solutionSteps != null && currentStep == solutionSteps.size()
```

```
- 1);

            boolean showExiting = isPrimary && isLastStep && showingFinalState;
            int exitingOffset = cellSize / 3;

            g.setColor(pieceColor);
            if (isVertical) {
                if (showExiting) {
                    if (exitCol == -1) {
                        g.fillRect(startX + col * cellSize - exitingOffset + 2,
                                startY + row * cellSize + 2,
                                cellSize - 4,
                                length * cellSize - 4);
                    } else if (exitCol == cols) {
                        g.fillRect(startX + col * cellSize + exitingOffset + 2,
                                startY + row * cellSize + 2,
                                cellSize - 4,
                                length * cellSize - 4);
                    } else {
                        g.fillRect(startX + col * cellSize + 2,
                                startY + row * cellSize + 2,
                                cellSize - 4,
                                length * cellSize - 4);
                    }
                } else {
                    g.fillRect(startX + col * cellSize + 2,
                            startY + row * cellSize + 2,
                            cellSize - 4,
                            length * cellSize - 4);
                }
            } else {
                if (showExiting) {
                    if (exitRow == -1) {
                        g.fillRect(startX + col * cellSize + 2,
                                startY + row * cellSize - exitingOffset + 2,
                                length * cellSize - 4,
                                cellSize - 4);
                    } else if (exitRow == rows) {
                        g.fillRect(startX + col * cellSize + 2,
                                startY + row * cellSize + exitingOffset + 2,
                                length * cellSize - 4,
                                cellSize - 4);
                    } else if (exitCol == -1) {
                        g.fillRect(startX + col * cellSize - exitingOffset + 2,
                                startY + row * cellSize + 2,
                                length * cellSize - 4,
                                cellSize - 4);
                    } else if (exitCol == cols) {
                        g.fillRect(startX + col * cellSize + exitingOffset + 2,
                                startY + row * cellSize + 2,
                                length * cellSize - 4,
                                cellSize - 4);
                    } else {
                        g.fillRect(startX + col * cellSize + 2,
                                startY + row * cellSize + 2,
                                length * cellSize - 4,
                                cellSize - 4);
                    }
                } else {
                    g.fillRect(startX + col * cellSize + 2,
                            startY + row * cellSize + 2,
                            length * cellSize - 4,
                            cellSize - 4);
                }
            }
```

```
            g.setColor(Color.BLACK);
            Font font = new Font("Arial", Font.BOLD, cellSize / 3);
            g.setFont(font);

            String text = String.valueOf(id);
            FontMetrics metrics = g.getFontMetrics(font);
            int textX, textY;

            if (isVertical) {
                if (showExiting) {
                    if (exitCol == -1) {
                        textX = startX + col * cellSize - exitingOffset + (cellSize -
metrics.stringWidth(text)) / 2;
                    } else if (exitCol == cols) {
                        textX = startX + col * cellSize + exitingOffset + (cellSize -
metrics.stringWidth(text)) / 2;
                    } else {
                        textX = startX + col * cellSize + (cellSize -
metrics.stringWidth(text)) / 2;
                    }
                } else {
                    textX = startX + col * cellSize + (cellSize - metrics.stringWidth(text)) /
2;
                }
                textY = startY + row * cellSize + cellSize / 2 + metrics.getAscent() / 2;
            } else {
                textX = startX + col * cellSize + (cellSize - metrics.stringWidth(text)) / 2;
                if (showExiting) {
                    if (exitRow == -1) {
                        textY = startY + row * cellSize - exitingOffset + (cellSize +
metrics.getAscent()) / 2;
                    } else if (exitRow == rows) {
                        textY = startY + row * cellSize + exitingOffset + (cellSize +
metrics.getAscent()) / 2;
                    } else if (exitCol == -1) {
                        textY = startY + row * cellSize + (cellSize + metrics.getAscent()) / 2;
                    } else if (exitCol == cols) {
                        textY = startY + row * cellSize + (cellSize + metrics.getAscent()) / 2;
                    } else {
                        textY = startY + row * cellSize + (cellSize + metrics.getAscent()) / 2;
                    }
                } else {
                    textY = startY + row * cellSize + (cellSize + metrics.getAscent()) / 2;
                }
            }

            g.drawString(text, textX, textY);
        }

        if (solutionSteps != null && currentStep > 0 && currentStep < solutionMoves.size() + 1
&& !showExitAnimation) {
            Move lastMove = solutionMoves.get(currentStep - 1);
            int pieceIndex = lastMove.getPieceIndex();
            Piece movedPiece = boardToDraw.getPieces().get(pieceIndex);

            int row = movedPiece.getRow();
            int col = movedPiece.getCol();
            int length = movedPiece.getLength();
            boolean isVertical = movedPiece.isVertical();

            g.setColor(Color.YELLOW);
            Graphics2D g2d = (Graphics2D) g;
            g2d.setStroke(new BasicStroke(3));

            if (isVertical) {
                g2d.drawRect(startX + col * cellSize + 1, startY + row * cellSize + 1,
```

```
                        cellSize - 2, length * cellSize - 2);
            } else {
                g2d.drawRect(startX + col * cellSize + 1, startY + row * cellSize + 1,
                        length * cellSize - 2, cellSize - 2);
            }
        }
    }

    private Color getPieceColor(char id) {
        if (id == 'P') {
            return Color.RED;
        }

        if (!pieceColors.containsKey(id)) {
            pieceColors.put(id, generatePastelColor());
        }

        return pieceColors.get(id);
    }

    private Color generatePastelColor() {
        float hue = (float) Math.random();
        float saturation = 0.5f;
        float brightness = 0.9f;

        return Color.getHSBColor(hue, saturation, brightness);
    }

    private void assignColorsToNewPieces() {
        pieceColors.clear();

        for (Piece piece : currentBoard.getPieces()) {
            char id = piece.getId();

            if (id == 'P') {
                pieceColors.put(id, Color.RED);
            } else {
                pieceColors.put(id, generatePastelColor());
            }
        }
    }

    private List<Move> extractMoves(List<Board> steps) {
        List<Move> moves = new ArrayList<>();

        if (steps == null || steps.size() < 2) {
            return moves;
        }

        for (int i = 1; i < steps.size(); i++) {
            Board prev = steps.get(i-1);
            Board curr = steps.get(i);

            List<Piece> prevPieces = prev.getPieces();
            List<Piece> currPieces = curr.getPieces();

            for (int j = 0; j < prevPieces.size(); j++) {
                Piece prevPiece = prevPieces.get(j);
                Piece currPiece = currPieces.get(j);

                if (prevPiece.getRow() != currPiece.getRow() ||
                    prevPiece.getCol() != currPiece.getCol()) {
                    Move move = new Move(
                        j,
                        prevPiece.getRow(),
                        prevPiece.getCol(),
```

```
                        currPiece.getRow(),
                        currPiece.getCol()
                    );
                    moves.add(move);
                    break;
                }
            }
        }

        return moves;
    }

    public static class SolutionCollector {
        private List<Board> solutionSteps = new ArrayList<>();

        public void addStep(Board board) {
            solutionSteps.add(new Board(board));
        }

        public List<Board> getSolutionSteps() {
            return solutionSteps;
        }
    }

    // Gui starter
    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            e.printStackTrace();
        }

        SwingUtilities.invokeLater(() -> new Gui());
    }
}
```

BoardEditor.java

```
package gui;

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

public class BoardEditor extends JFrame {
    private int rows = 6;
    private int cols = 6;
    private EditorPanel boardPanel;

    private char nextPieceId = 'A';
    private boolean placingPrimaryPiece = false;
    private boolean placingExit = false;
    private boolean isVertical = false;
    private int pieceLength = 2;
    private int exitRow = -1;
    private int exitCol = -1;

    private JButton createButton;
    private JButton cancelButton;
    private JButton primaryPieceButton;
```

```
    private JButton exitButton;
    private JComboBox<String> orientationSelector;
    private JSpinner lengthSpinner;
    private JButton clearButton;
    private JButton undoButton;
    private JLabel statusLabel;

    private List<EditorPiece> placedPieces = new ArrayList<>();
    private Map<Character, Color> pieceColors = new HashMap<>();

    public BoardEditor(JFrame parent) {
        super("Rush Hour Board Editor");
        setModal(parent);
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setSize(800, 700);
        setLayout(new BorderLayout());

        createControlPanel();
        createBoardPanel();
        createStatusBar();

        setLocationRelativeTo(parent);
    }

    private void setModal(JFrame parent) {
        if (parent != null) {
            addWindowListener(new WindowAdapter() {
                @Override
                public void windowOpened(WindowEvent e) {
                    parent.setEnabled(false);
                }

                @Override
                public void windowClosed(WindowEvent e) {
                    parent.setEnabled(true);
                    parent.toFront();
                }
            });
        }
    }

    private void createControlPanel() {
        JPanel controlPanel = new JPanel(new BorderLayout());

        JPanel sizePanel = new JPanel();
        JLabel rowsLabel = new JLabel("Rows:");
        JSpinner rowsSpinner = new JSpinner(new SpinnerNumberModel(6, 3, 20, 1));
        rowsSpinner.addChangeListener(e -> {
            rows = (int) rowsSpinner.getValue();
            resetBoard();
        });

        JLabel colsLabel = new JLabel("Columns:");
        JSpinner colsSpinner = new JSpinner(new SpinnerNumberModel(6, 3, 20, 1));
        colsSpinner.addChangeListener(e -> {
            cols = (int) colsSpinner.getValue();
            resetBoard();
        });

        sizePanel.add(rowsLabel);
        sizePanel.add(rowsSpinner);
        sizePanel.add(colsLabel);
        sizePanel.add(colsSpinner);

        JPanel piecePanel = new JPanel();

        primaryPieceButton = new JButton("Place Primary Piece (P)");
        primaryPieceButton.addActionListener(e -> togglePrimaryPiecePlacement());

        exitButton = new JButton("Place Exit");
```

```
        exitButton.addActionListener(e -> toggleExitPlacement());

        orientationSelector = new JComboBox<>(new String[] {"Horizontal", "Vertical"});
        orientationSelector.addActionListener(e -> {
            isVertical = orientationSelector.getSelectedIndex() == 1;
            updateButtonStates();
        });

        JLabel lengthLabel = new JLabel("Length:");
        lengthSpinner = new JSpinner(new SpinnerNumberModel(2, 2, Math.max(rows, cols), 1));
        lengthSpinner.addChangeListener(e -> {
            pieceLength = (Integer) lengthSpinner.getValue();
            updateButtonStates();
        });

        clearButton = new JButton("Clear Board");
        clearButton.addActionListener(e -> clearBoard());

        undoButton = new JButton("Undo Last");
        undoButton.addActionListener(e -> undoLastPlacement());
        undoButton.setEnabled(false);

        piecePanel.add(primaryPieceButton);
        piecePanel.add(exitButton);
        piecePanel.add(new JLabel("Orientation:"));
        piecePanel.add(orientationSelector);
        piecePanel.add(lengthLabel);
        piecePanel.add(lengthSpinner);
        piecePanel.add(clearButton);
        piecePanel.add(undoButton);

        JPanel buttonPanel = new JPanel();
        createButton = new JButton("Create Puzzle");
        createButton.addActionListener(e -> createPuzzle());
        createButton.setEnabled(false);

        cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(e -> dispose());

        buttonPanel.add(createButton);
        buttonPanel.add(cancelButton);

        controlPanel.add(sizePanel, BorderLayout.NORTH);
        controlPanel.add(piecePanel, BorderLayout.CENTER);
        controlPanel.add(buttonPanel, BorderLayout.SOUTH);

        add(controlPanel, BorderLayout.NORTH);
    }

    private void createBoardPanel() {
        boardPanel = new EditorPanel();
        boardPanel.setPreferredSize(new Dimension(600, 600));
        JPanel centeringPanel = new JPanel(new GridBagLayout());
        centeringPanel.add(boardPanel);
        add(new JScrollPane(centeringPanel), BorderLayout.CENTER);
    }

    private void createStatusBar() {
        JPanel statusPanel = new JPanel();
        statusLabel = new JLabel("Click on the board to place pieces");
        statusPanel.add(statusLabel);
        add(statusPanel, BorderLayout.SOUTH);
    }

    private void togglePrimaryPiecePlacement() {
        if (hasPrimaryPiece()) {
            JOptionPane.showMessageDialog(this,
                "Primary piece already placed. Clear board or remove it first.",
                "Primary Piece Exists",
                JOptionPane.WARNING_MESSAGE);
```

```
            return;
        }

        placingPrimaryPiece = !placingPrimaryPiece;
        placingExit = false;
        updateButtonStates();
    }

    private void toggleExitPlacement() {
        placingExit = !placingExit;
        placingPrimaryPiece = false;
        updateButtonStates();
    }

    private void updateButtonStates() {
        primaryPieceButton.setBackground(placingPrimaryPiece ? Color.YELLOW : null);
        exitButton.setBackground(placingExit ? Color.YELLOW : null);

        int maxLength = isVertical ? rows : cols;
        SpinnerNumberModel model = (SpinnerNumberModel) lengthSpinner.getModel();
        model.setMaximum(maxLength);

        if (pieceLength > maxLength) {
            lengthSpinner.setValue(maxLength);
            pieceLength = maxLength;
        }

        if (placingPrimaryPiece) {
            statusLabel.setText("Click on the board to place the primary piece (P)");
        } else if (placingExit) {
            statusLabel.setText("Click on a board edge to place the exit");
        } else {
            statusLabel.setText("Click on the board to place piece '" + nextPieceId + "'");
        }

        createButton.setEnabled(hasPrimaryPiece() && exitRow != -1 && exitCol != -1);
    }

    private boolean hasPrimaryPiece() {
        for (EditorPiece piece : placedPieces) {
            if (piece.id == 'P') {
                return true;
            }
        }
        return false;
    }

    private void clearBoard() {
        placedPieces.clear();
        exitRow = -1;
        exitCol = -1;
        nextPieceId = 'A';
        undoButton.setEnabled(false);
        createButton.setEnabled(false);
        placingPrimaryPiece = false;
        placingExit = false;
        pieceColors.clear();
        updateButtonStates();
        boardPanel.repaint();
    }

    // Reset board
    private void resetBoard() {
        clearBoard();
        updateButtonStates();
        boardPanel.repaint();
    }

    // buat undo
    private void undoLastPlacement() {
```

```
        if (!placedPieces.isEmpty()) {
            EditorPiece removed = placedPieces.remove(placedPieces.size() - 1);

            if (removed.id == 'P') {
                createButton.setEnabled(false);
            }

            if (removed.id != 'P' && removed.id > 'A') {
                nextPieceId = (char) (removed.id - 1);
            }

            undoButton.setEnabled(!placedPieces.isEmpty());
            updateButtonStates();
            boardPanel.repaint();
        }
    }

    // Buat dan simpan puzzle
    private void createPuzzle() {
        if (!hasPrimaryPiece()) {
            JOptionPane.showMessageDialog(this,
                "You must place a primary piece (P) on the board.",
                "Missing Primary Piece",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        if (exitRow == -1 && exitCol == -1) {
            JOptionPane.showMessageDialog(this,
                "You must place an exit on the board.",
                "Missing Exit",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        JFileChooser fileChooser = new JFileChooser(new File("test/input"));
        fileChooser.setDialogTitle("Save Puzzle");
        fileChooser.setSelectedFile(new File("custom_puzzle.txt"));

        int result = fileChooser.showSaveDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            File file = fileChooser.getSelectedFile();

            if (!file.getName().toLowerCase().endsWith(".txt")) {
                file = new File(file.getAbsolutePath() + ".txt");
            }

            try {
                savePuzzleToFile(file);
                JOptionPane.showMessageDialog(this,
                    "Puzzle saved successfully to " + file.getName(),
                    "Save Successful",
                    JOptionPane.INFORMATION_MESSAGE);
                dispose();
            } catch (IOException e) {
                JOptionPane.showMessageDialog(this,
                    "Error saving puzzle: " + e.getMessage(),
                    "Save Error",
                    JOptionPane.ERROR_MESSAGE);
                e.printStackTrace();
            }
        }
    }

    // Simpan puzzle ke file
    private void savePuzzleToFile(File file) throws IOException {
        FileWriter writer = new FileWriter(file);

        writer.write(rows + " " + cols + "\n");
```

```
        int regularPieceCount = 0;
        for (EditorPiece piece : placedPieces) {
            if (piece.id != 'P') {
                regularPieceCount++;
            }
        }

        writer.write(regularPieceCount + "\n");

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

        for (EditorPiece piece : placedPieces) {
            if (piece.isVertical) {
                for (int r = piece.row; r < piece.row + piece.length; r++) {
                    grid[r][piece.col] = piece.id;
                }
            } else {
                for (int c = piece.col; c < piece.col + piece.length; c++) {
                    grid[piece.row][c] = piece.id;
                }
            }
        }

        for (int r = 0; r < rows; r++) {
            if (exitRow == r && exitCol == -1) {
                writer.write('K');
            }

            for (int c = 0; c < cols; c++) {
                writer.write(grid[r][c]);
            }

            if (exitRow == r && exitCol == cols) {
                writer.write('K');
            }

            writer.write("\n");
        }

        if (exitRow == -1) {
            for (int c = 0; c < cols; c++) {
                if (c == exitCol) {
                    writer.write('K');
                } else {
                    writer.write('.');
                }
            }
            writer.write("\n");
        } else if (exitRow == rows) {
            for (int c = 0; c < cols; c++) {
                if (c == exitCol) {
                    writer.write('K');
                } else {
                    writer.write('.');
                }
            }
            writer.write("\n");
        }

        writer.close();
    }

    private Color getPieceColor(char id) {
        if (id == 'P') {
            return Color.RED;
```

```
        }

        if (!pieceColors.containsKey(id)) {
            pieceColors.put(id, generatePastelColor());
        }

        return pieceColors.get(id);
    }

    private Color generatePastelColor() {
        float hue = (float) Math.random();
        float saturation = 0.5f;
        float brightness = 0.9f;

        return Color.getHSBColor(hue, saturation, brightness);
    }

    // menampilkan papan dan menangani input mouse
    private class EditorPanel extends JPanel {
        private static final int CELL_SIZE = 40;
        private static final int BORDER_SIZE = 20;
        private Point hoverPoint = null;

        public EditorPanel() {
            addMouseListener(new MouseAdapter() {
                @Override
                public void mouseClicked(MouseEvent e) {
                    handleMouseClick(e.getX(), e.getY());
                }
            });

            addMouseMotionListener(new MouseMotionAdapter() {
                @Override
                public void mouseMoved(MouseEvent e) {
                    hoverPoint = e.getPoint();
                    repaint();
                }
            });

            addMouseListener(new MouseAdapter() {
                @Override
                public void mouseExited(MouseEvent e) {
                    hoverPoint = null;
                    repaint();
                }
            });
        }

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            drawBoard(g);
        }

        // Menggambar papan dan semua elemennya
        private void drawBoard(Graphics g) {
            int startX = (getWidth() - cols * CELL_SIZE) / 2;
            int startY = (getHeight() - rows * CELL_SIZE) / 2;
            int boardWidth = cols * CELL_SIZE;
            int boardHeight = rows * CELL_SIZE;

            g.setColor(Color.WHITE);
            g.fillRect(startX, startY, boardWidth, boardHeight);

            g.setColor(Color.LIGHT_GRAY);
            for (int r = 0; r <= rows; r++) {
                g.drawLine(startX, startY + r * CELL_SIZE, startX + boardWidth, startY + r *
CELL_SIZE);
            }
            for (int c = 0; c <= cols; c++) {
```

```
                g.drawLine(startX + c * CELL_SIZE, startY, startX + c * CELL_SIZE, startY +
boardHeight);
            }

            g.setColor(Color.GREEN);
            if (exitRow != -1 || exitCol != -1) {
                if (exitRow == -1) {
                    g.fillRect(startX + exitCol * CELL_SIZE, startY - CELL_SIZE/2,
CELL_SIZE, CELL_SIZE/2);
                } else if (exitRow == rows) {
                    g.fillRect(startX + exitCol * CELL_SIZE, startY + rows * CELL_SIZE,
CELL_SIZE, CELL_SIZE/2);
                } else if (exitCol == -1) {
                    g.fillRect(startX - CELL_SIZE/2, startY + exitRow * CELL_SIZE,
CELL_SIZE/2, CELL_SIZE);
                } else if (exitCol == cols) {
                    g.fillRect(startX + cols * CELL_SIZE, startY + exitRow * CELL_SIZE,
CELL_SIZE/2, CELL_SIZE);
                }
            }

            for (EditorPiece piece : placedPieces) {
                Color pieceColor = getPieceColor(piece.id);

                g.setColor(pieceColor);
                if (piece.isVertical) {
                    g.fillRect(startX + piece.col * CELL_SIZE + 2,
                               startY + piece.row * CELL_SIZE + 2,
                               CELL_SIZE - 4,
                               piece.length * CELL_SIZE - 4);
                } else {
                    g.fillRect(startX + piece.col * CELL_SIZE + 2,
                               startY + piece.row * CELL_SIZE + 2,
                               piece.length * CELL_SIZE - 4,
                               CELL_SIZE - 4);
                }

                g.setColor(Color.BLACK);
                Font font = new Font("Arial", Font.BOLD, CELL_SIZE / 3);
                g.setFont(font);

                FontMetrics metrics = g.getFontMetrics(font);
                String text = String.valueOf(piece.id);
                int textX, textY;

                if (piece.isVertical) {
                    textX = startX + piece.col * CELL_SIZE + (CELL_SIZE -
metrics.stringWidth(text)) / 2;
                    textY = startY + piece.row * CELL_SIZE + CELL_SIZE / 2;
                } else {
                    textX = startX + piece.col * CELL_SIZE + (CELL_SIZE -
metrics.stringWidth(text)) / 2;
                    textY = startY + piece.row * CELL_SIZE + (CELL_SIZE +
metrics.getAscent()) / 2;
                }

                g.drawString(text, textX, textY);
            }

            if ((placingPrimaryPiece || !placingExit) && hoverPoint != null) {
                int mouseX = hoverPoint.x;
                int mouseY = hoverPoint.y;

                int gridCol = (mouseX - startX) / CELL_SIZE;
                int gridRow = (mouseY - startY) / CELL_SIZE;

                if (gridCol >= 0 && gridCol < cols && gridRow >= 0 && gridRow < rows) {
                    boolean fits = true;
                    if (isVertical) {
                        fits = gridRow + pieceLength <= rows;
```

```
                    } else {
                        fits = gridCol + pieceLength <= cols;
                    }

                    if (fits) {
                        boolean overlaps = false;
                        for (int i = 0; i < pieceLength; i++) {
                            int checkRow = isVertical ? gridRow + i : gridRow;
                            int checkCol = isVertical ? gridCol : gridCol + i;

                            for (EditorPiece piece : placedPieces) {
                                if (piece.isVertical) {
                                    for (int r = piece.row; r < piece.row + piece.length;
r++) {
                                        if (r == checkRow && piece.col == checkCol) {
                                            overlaps = true;
                                            break;
                                        }
                                    }
                                } else {
                                    for (int c = piece.col; c < piece.col + piece.length;
c++) {
                                        if (piece.row == checkRow && c == checkCol) {
                                            overlaps = true;
                                            break;
                                        }
                                    }
                                }
                                if (overlaps) break;
                            }
                            if (overlaps) break;
                        }

                        Color hoverColor;
                        if (placingPrimaryPiece) {
                            hoverColor = overlaps ?
                                new Color(255, 0, 0, 128) :
                                new Color(255, 0, 0, 64);
                        } else {
                            hoverColor = overlaps ?
                                new Color(255, 0, 0, 128) :
                                new Color(0, 150, 0, 64);
                        }

                        g.setColor(hoverColor);

                        if (isVertical) {
                            g.fillRect(startX + gridCol * CELL_SIZE + 2,
                                       startY + gridRow * CELL_SIZE + 2,
                                       CELL_SIZE - 4,
                                       pieceLength * CELL_SIZE - 4);
                        } else {
                            g.fillRect(startX + gridCol * CELL_SIZE + 2,
                                       startY + gridRow * CELL_SIZE + 2,
                                       pieceLength * CELL_SIZE - 4,
                                       CELL_SIZE - 4);
                        }

                        g.setColor(Color.BLACK);
                        Font font = new Font("Arial", Font.BOLD, CELL_SIZE / 3);
                        g.setFont(font);

                        FontMetrics metrics = g.getFontMetrics(font);
                        String text = String.valueOf(placingPrimaryPiece ? 'P' :
nextPieceId);
                        int textX, textY;

                        if (isVertical) {
                            textX = startX + gridCol * CELL_SIZE + (CELL_SIZE -
metrics.stringWidth(text)) / 2;
```

```
                            textY = startY + gridRow * CELL_SIZE + CELL_SIZE / 2;
                        } else {
                            textX = startX + gridCol * CELL_SIZE + (CELL_SIZE -
metrics.stringWidth(text)) / 2;
                            textY = startY + gridRow * CELL_SIZE + (CELL_SIZE +
metrics.getAscent()) / 2;
                        }

                        g.drawString(text, textX, textY);
                    }
                }
            }

            if (placingExit && hoverPoint != null) {
                int mouseX = hoverPoint.x;
                int mouseY = hoverPoint.y;

                boolean validHover = false;

                if (mouseY < startY && mouseY > startY - CELL_SIZE/2) {
                    int col = (mouseX - startX) / CELL_SIZE;
                    if (col >= 0 && col < cols) {
                        g.setColor(new Color(0, 200, 0, 128));
                        g.fillRect(startX + col * CELL_SIZE, startY - CELL_SIZE/2,
CELL_SIZE, CELL_SIZE/2);
                        validHover = true;
                    }
                }

                if (!validHover && mouseY > startY + boardHeight && mouseY < startY +
boardHeight + CELL_SIZE/2) {
                    int col = (mouseX - startX) / CELL_SIZE;
                    if (col >= 0 && col < cols) {
                        g.setColor(new Color(0, 200, 0, 128));
                        g.fillRect(startX + col * CELL_SIZE, startY + rows * CELL_SIZE,
CELL_SIZE, CELL_SIZE/2);
                        validHover = true;
                    }
                }

                if (!validHover && mouseX < startX && mouseX > startX - CELL_SIZE/2) {
                    int row = (mouseY - startY) / CELL_SIZE;
                    if (row >= 0 && row < rows) {
                        g.setColor(new Color(0, 200, 0, 128));
                        g.fillRect(startX - CELL_SIZE/2, startY + row * CELL_SIZE,
CELL_SIZE/2, CELL_SIZE);
                        validHover = true;
                    }
                }

                if (!validHover && mouseX > startX + boardWidth && mouseX < startX +
boardWidth + CELL_SIZE/2) {
                    int row = (mouseY - startY) / CELL_SIZE;
                    if (row >= 0 && row < rows) {
                        g.setColor(new Color(0, 200, 0, 128));
                        g.fillRect(startX + cols * CELL_SIZE, startY + row * CELL_SIZE,
CELL_SIZE/2, CELL_SIZE);
                    }
                }
            }
        }

        private void handleMouseClick(int x, int y) {
            int startX = (getWidth() - cols * CELL_SIZE) / 2;
            int startY = (getHeight() - rows * CELL_SIZE) / 2;
            int boardWidth = cols * CELL_SIZE;
            int boardHeight = rows * CELL_SIZE;

            if (placingExit) {
                if (y < startY && y > startY - CELL_SIZE/2) {
```

```
                    int col = (x - startX) / CELL_SIZE;
                    if (col >= 0 && col < cols) {
                        exitRow = -1;
                        exitCol = col;
                        placingExit = false;
                        updateButtonStates();
                        repaint();
                        return;
                    }
                }

                if (y > startY + boardHeight && y < startY + boardHeight + CELL_SIZE/2) {
                    int col = (x - startX) / CELL_SIZE;
                    if (col >= 0 && col < cols) {
                        exitRow = rows;
                        exitCol = col;
                        placingExit = false;
                        updateButtonStates();
                        repaint();
                        return;
                    }
                }

                if (x < startX && x > startX - CELL_SIZE/2) {
                    int row = (y - startY) / CELL_SIZE;
                    if (row >= 0 && row < rows) {
                        exitRow = row;
                        exitCol = -1;
                        placingExit = false;
                        updateButtonStates();
                        repaint();
                        return;
                    }
                }

                if (x > startX + boardWidth && x < startX + boardWidth + CELL_SIZE/2) {
                    int row = (y - startY) / CELL_SIZE;
                    if (row >= 0 && row < rows) {
                        exitRow = row;
                        exitCol = cols;
                        placingExit = false;
                        updateButtonStates();
                        repaint();
                        return;
                    }
                }

                return;
            }

            int gridCol = (x - startX) / CELL_SIZE;
            int gridRow = (y - startY) / CELL_SIZE;

            if (gridCol < 0 || gridCol >= cols || gridRow < 0 || gridRow >= rows) {
                return;
            }

            if (isVertical && gridRow + pieceLength > rows) {
                statusLabel.setText("Piece doesn't fit! Try a different position or
orientation.");
                return;
            }
            if (!isVertical && gridCol + pieceLength > cols) {
                statusLabel.setText("Piece doesn't fit! Try a different position or
orientation.");
                return;
            }

            for (int i = 0; i < pieceLength; i++) {
                int checkRow = isVertical ? gridRow + i : gridRow;
```

```
                int checkCol = isVertical ? gridCol : gridCol + i;

                for (EditorPiece piece : placedPieces) {
                    if (piece.isVertical) {
                        for (int r = piece.row; r < piece.row + piece.length; r++) {
                            if (r == checkRow && piece.col == checkCol) {
                                statusLabel.setText("Space is occupied! Try a different
position.");
                                return;
                            }
                        }
                    } else {
                        for (int c = piece.col; c < piece.col + piece.length; c++) {
                            if (piece.row == checkRow && c == checkCol) {
                                statusLabel.setText("Space is occupied! Try a different
position.");
                                return;
                            }
                        }
                    }
                }
            }

            char pieceId = placingPrimaryPiece ? 'P' : nextPieceId;
            EditorPiece newPiece = new EditorPiece(pieceId, gridRow, gridCol, pieceLength,
isVertical);
            placedPieces.add(newPiece);

            if (placingPrimaryPiece) {
                placingPrimaryPiece = false;
            } else {
                nextPieceId = (char) (nextPieceId + 1);
            }

            undoButton.setEnabled(true);
            updateButtonStates();
            repaint();
        }

        @Override
        public Dimension getPreferredSize() {
            return new Dimension(
                cols * CELL_SIZE + 2 * BORDER_SIZE,
                rows * CELL_SIZE + 2 * BORDER_SIZE
            );
        }
    }

    private static class EditorPiece {
        char id;
        int row;
        int col;
        int length;
        boolean isVertical;

        EditorPiece(char id, int row, int col, int length, boolean isVertical) {
            this.id = id;
            this.row = row;
            this.col = col;
            this.length = length;
            this.isVertical = isVertical;
        }
    }
}
```

2.5 Main

Main.java

```
import algorithm.*;
import gui.Gui;
import java.util.Scanner;
import model.Board;
import util.FileParser;

public class Main {
    public static void main(String[] args) {

        if (args.length > 0 && args[0].equalsIgnoreCase("--gui")) {
            // buat launch GUI
            javax.swing.SwingUtilities.invokeLater(() -> new Gui());
            return;
        }

        Scanner scanner = new Scanner(System.in);

        System.out.println("Rush Hour Puzzle Solver");
        System.out.println("======================");

        // Input path kalau CLI
        System.out.print("Enter the path to the input file: ");
        String filePath = scanner.nextLine();

        try {
            // Parse input
            FileParser parser = new FileParser();
            Board initialBoard = parser.parseFile(filePath);

            // Pilih algoritma
            System.out.println("\nChoose the algorithm:");
            System.out.println("1. Uniform Cost Search (UCS)");
            System.out.println("2. Greedy Best-First Search (GBFS)");
            System.out.println("3. A* Search");
            System.out.println("4. IDA* Search");
            System.out.print("Enter your choice: ");
            int choice = 1;
            String input = scanner.nextLine();
            if (!input.isEmpty()) {
                choice = Integer.parseInt(input);
            }

            int heuristicChoice = 1; // default
            if (choice == 2 || choice == 3 || choice == 4) {
                System.out.println("\nChoose the heuristic:");
                System.out.println("1. Blocking Pieces (count blocking vehicles)");
                System.out.println("2. Manhattan Distance (distance to exit)");
                System.out.println("3. Combined (blocking pieces + weighted Manhattan)");
                System.out.print("Enter your choice (1): ");
                input = scanner.nextLine();
                if (!input.isEmpty()) {
                    heuristicChoice = Integer.parseInt(input);
                }
            }

            switch (choice) {
                case 1:
                    System.out.println("\nSolving with Uniform Cost Search (UCS)...");
                    UCS ucs = new UCS(null);
```

```
                    ucs.solve(initialBoard);
                    break;
                case 2:
                    System.out.println("\nSolving with Greedy Best-First Search (GBFS)...");
                    GBFS gbfs = new GBFS(heuristicChoice,null);
                    gbfs.solve(initialBoard);
                    break;
                case 3:
                    System.out.println("\nSolving with A* Search...");
                    AStar aStar = new AStar(heuristicChoice, null);
                    aStar.solve(initialBoard);
                    break;
                case 4:
                    System.out.println("Solving with Iterative Deepening A* (IDA*) Search...");
                    IDAStar idaStar = new IDAStar(heuristicChoice, null);
                    idaStar.solve(initialBoard);
                    break;
                default:
                    System.out.println("Invalid choice. Using UCS by default.");
                    UCS defaultUcs = new UCS(null);
                    defaultUcs.solve(initialBoard);
            }

        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        } finally {
            scanner.close();
        }
    }
}
```

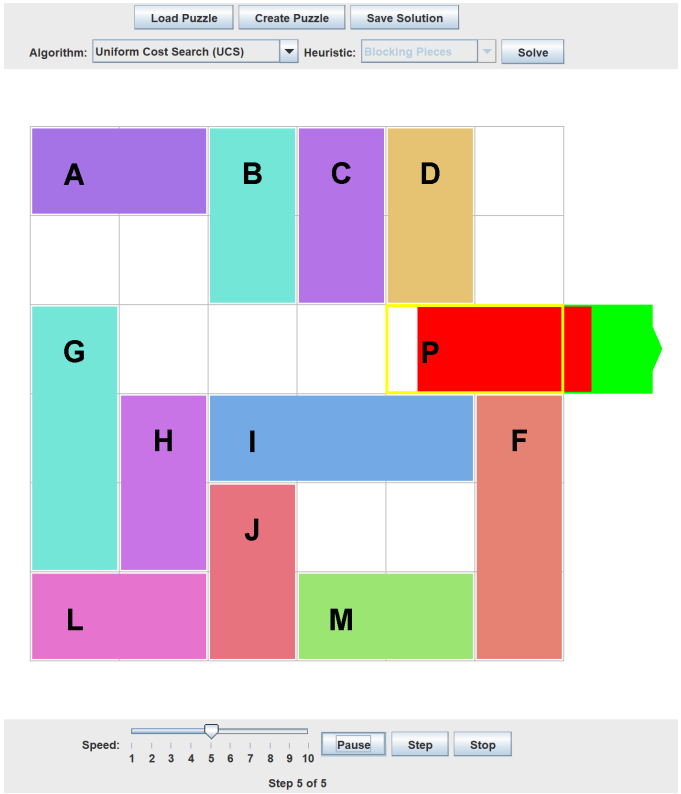
BAB 3
TEST CASE

3.1 Kasus 1 (Test Case Spesifikasi)

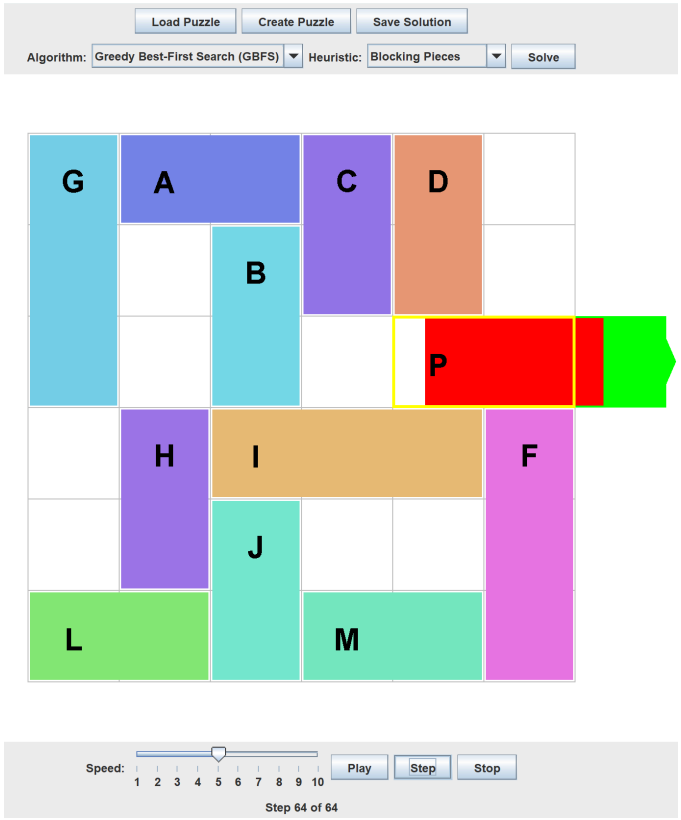
Input:

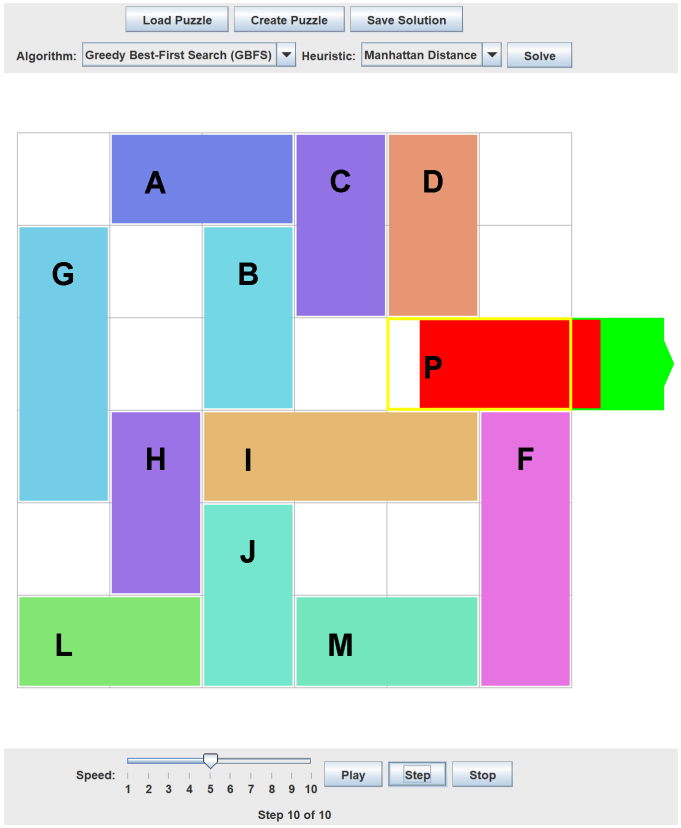
```
6 6
11
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

Algoritma (heuristik)	Output CLI	Output GUI

UCS	Initial Board: +------+ |AAB..F| |..BCDF| |GPPCDFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 1: D atas +------+ |AAB.DF| |..BCDF| |GPPC.FK |GH.III| |GHJ...| |LLJMM.| +------+ Move 2: I kiri +------+ |AAB.DF| |..BCDF| |GPPC.FK |GHIII.| |GHJ...| |LLJMM.| +------+ Move 3: C atas +------+ |AABCDF| |..BCDF| |GPP..FK |GHIII.| |GHJ...| |LLJMM.| +------+	

	<pre>Move 4: F bawah 3 cells
	+------+
	|AABCD.|
	|..BCD.|
	|GPP...K
	|GHIIIF|
	|GHJ..F|
	|LLJMMF|
	+------+
	
	Move 5: P kanan 3 cells
	+------+
	|AABCD.|
	|..BCD.|
	|G...PPK
	|GHIIIF|
	|GHJ..F|
	|LLJMMF|
	+------+
	Jumlah langkah: 5
	Jumlah node yang diperiksa: 426
	Waktu eksekusi: 0.045 detik</pre>	

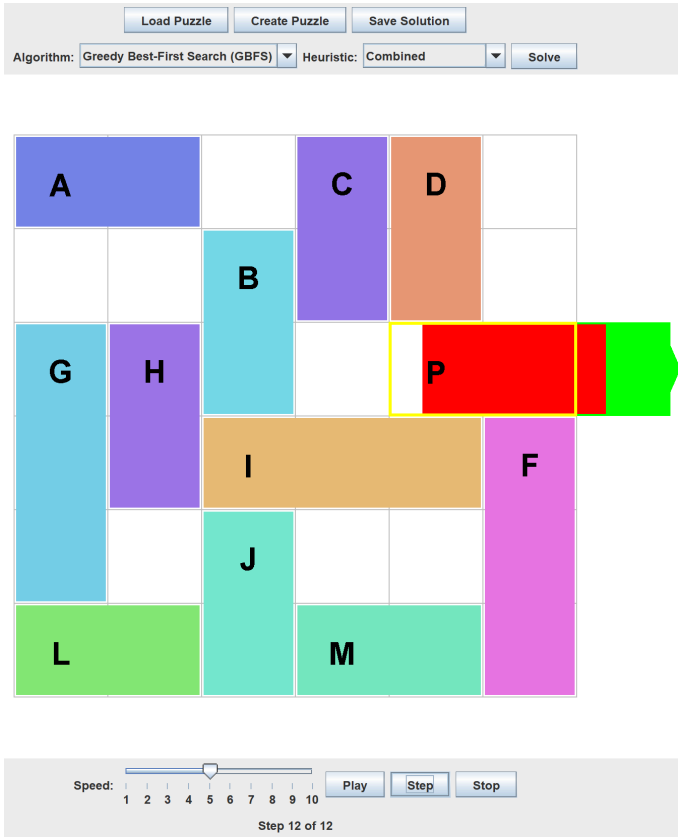
GBFS (blocking)	<pre>Using heuristic: Blocking Pieces
	
	Initial Board:
	+------+
	|AAB..F|
	|..BCDF|
	|GPPCDFK
	|GH.III|
	|GHJ...|
	|LLJMM.|
	+------+
	
	Move 1: C atas
	+------+
	|AABC.F|
	|..BCDF|
	|GPP.DFK
	|GH.III|
	|GHJ...|
	|LLJMM.|
	+------+</pre>
	<pre>Move 63: F bawah
	+------+
	|GAACD.|
	|G.BCD.|
	|G.BPP.K
	|.HIIIF|
	|.HJ..F|
	|LLJMMF|
	+------+
	
	Move 64: P kanan
	+------+
	|GAACD.|
	|G.BCD.|
	|G.B.PPK
	|.HIIIF|
	|.HJ..F|
	|LLJMMF|
	+------+
	Jumlah langkah: 64
	Jumlah node yang diperiksa: 380
	Waktu eksekusi: 0.034 detik</pre>
		

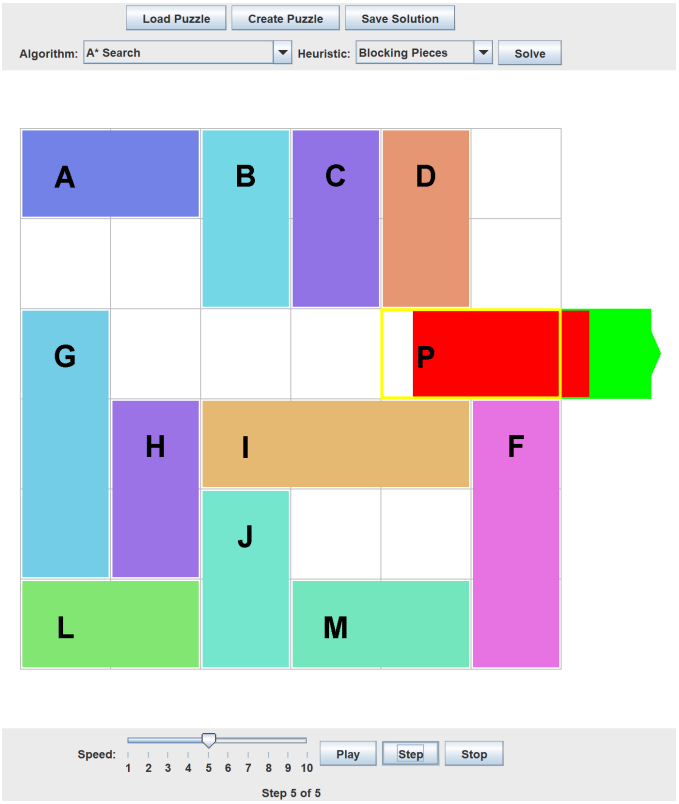
GBFS (Manhattan Distance)	Using heuristic: Manhattan Distance Initial Board: +------+ |AAB..F| |..BCDF| |GPPCDFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 1: C atas +------+ |AABC.F| |..BCDF| |GPP.DFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 2: P kanan +------+ |AABC.F| |..BCDF| |G.PPDFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 8: I kiri +------+ |.AACDF| |G.BCDF| |G.BPPFK |GHIII.| |.HJ...| |LLJMM.| +------+ Move 9: F bawah 3 cells +------+ |.AACD.| |G.BCD.| |G.BPP.K |GHIIIF| |.HJ..F| |LLJMMF| +------+ Move 10: P kanan +------+ |.AACD.| |G.BCD.| |G.B.PPK |GHIIIF| |.HJ..F| |LLJMMF| +------+ Jumlah langkah: 10 Jumlah node yang diperiksa: 53 Waktu eksekusi: 0.005 detik	

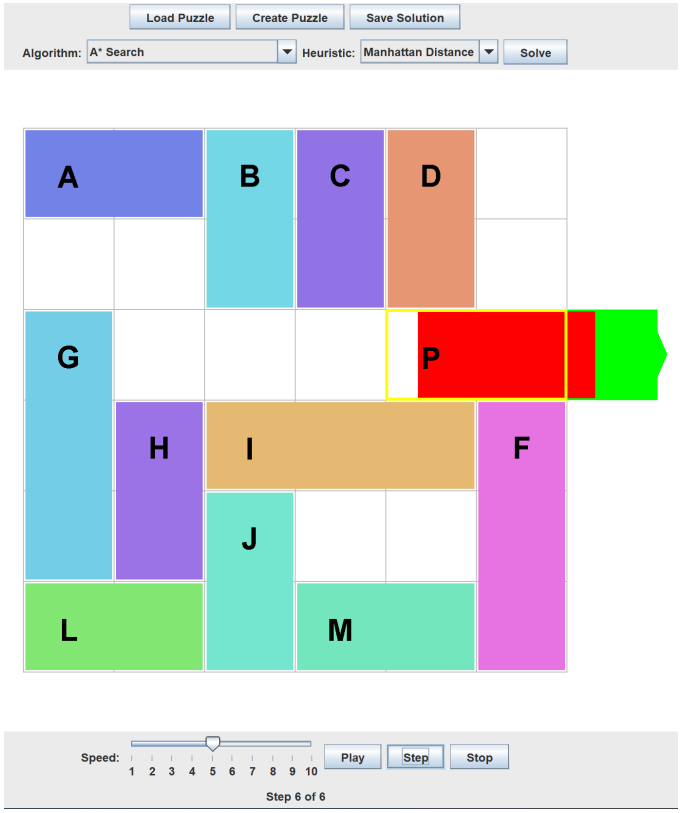
GBFS (Combined)

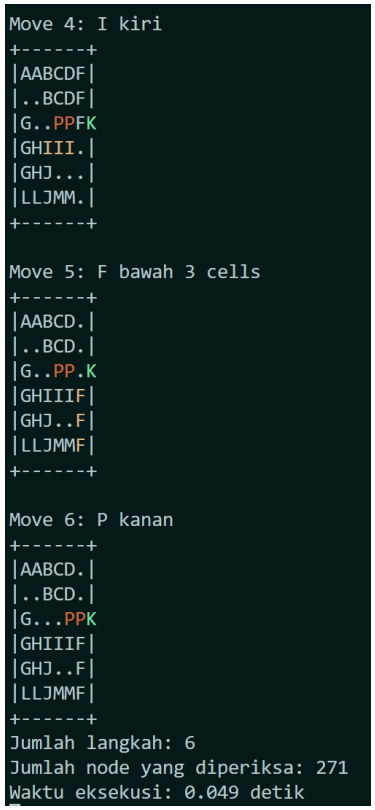
```
Using heuristic: Combined (Blocking + Manhattan)

Initial Board:
+------+
|AAB..F|
|..BCDF|
|GPPCDFK
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 1: C atas
+------+
|AABC.F|
|..BCDF|
|GPP.DFK
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 2: D atas
+------+
|AABCDF|
|..BCDF|
|GPP..FK
|GH.III|
|GHJ...|
|LLJMM.|
+------+
```

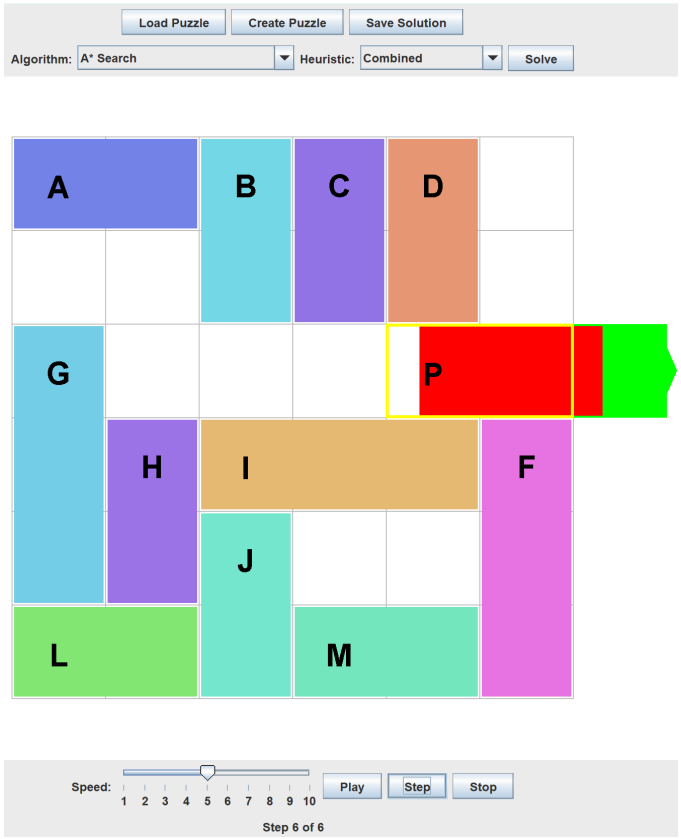
```
Move 10: I kiri
+------+
|AA.CDF|
|..BCDF|
|GHBPPFK
|GHIII.|
|G.J...|
|LLJMM.|
+------+

Move 11: F bawah 3 cells
+------+
|AA.CD.|
|..BCD.|
|GHBPP.K
|GHIIIF|
|G.J..F|
|LLJMMF|
+------+

Move 12: P kanan
+------+
|AA.CD.|
|..BCD.|
|GHB.PPK
|GHIIIF|
|G.J..F|
|LLJMMF|
+------+
Jumlah langkah: 12
Jumlah node yang diperiksa: 37
Waktu eksekusi: 0.009 detik
```

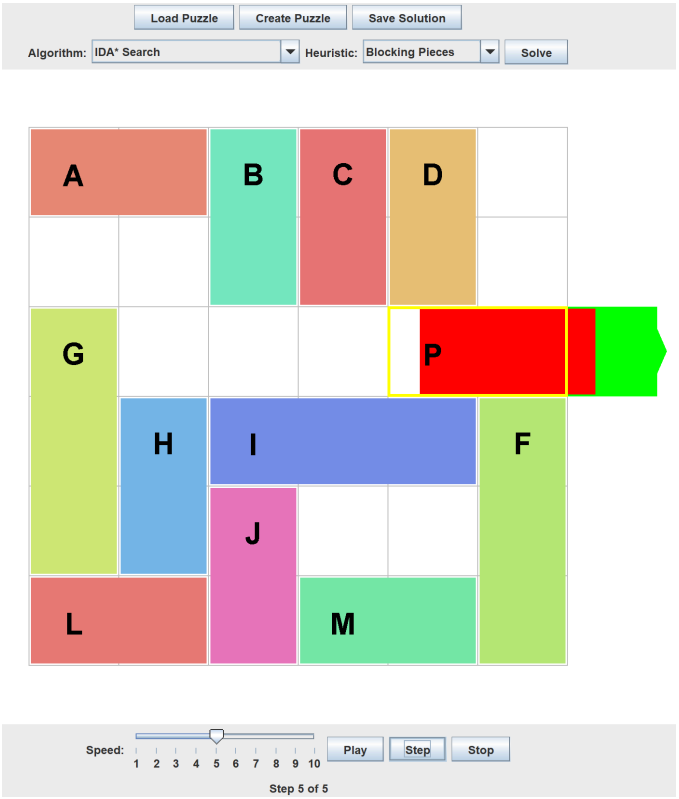


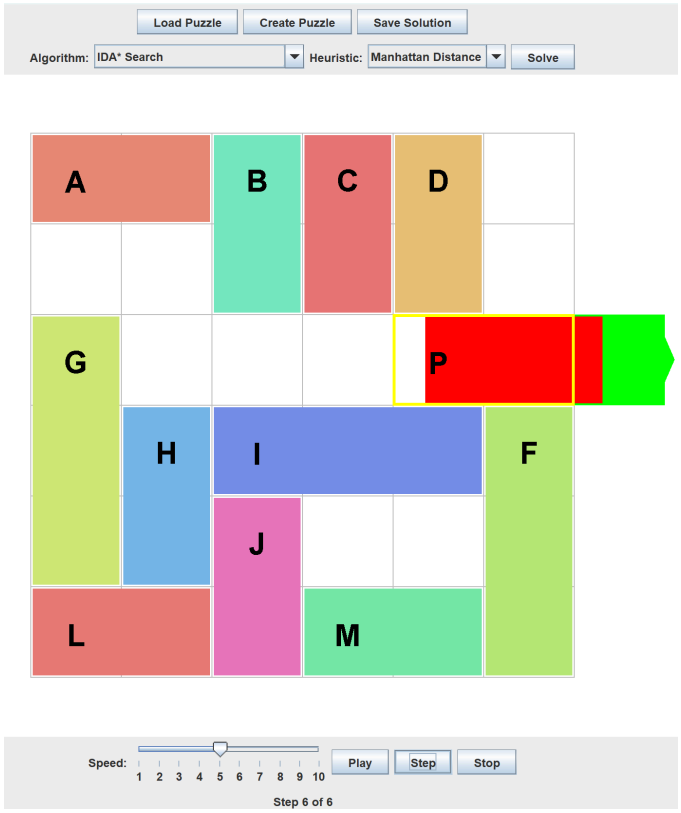
A* (Blocking)	Using heuristic: Blocking Pieces Initial Board: +------+ |AAB..F| |..BCDF| |GPPCDFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 1: C atas +------+ |AABC.F| |..BCDF| |GPP.DFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 2: D atas +------+ |AABCDF| |..BCDF| |GPP..FK |GH.III| |GHJ...| |LLJMM.| +------+ Move 3: I kiri +------+ |AABCDF| |..BCDF| |GPP..FK |GHIII.| |GHJ...| |LLJMM.| +------+ Move 4: F bawah 3 cells +------+ |AABCD.| |..BCD.| |GPP...K |GHIIIF| |GHJ..F| |LLJMMF| +------+ Move 5: P kanan 3 cells +------+ |AABCD.| |..BCD.| |G...PPK |GHIIIF| |GHJ..F| |LLJMMF| +------+ Jumlah langkah: 5 Jumlah node yang diperiksa: 44 Waktu eksekusi: 0.016 detik	

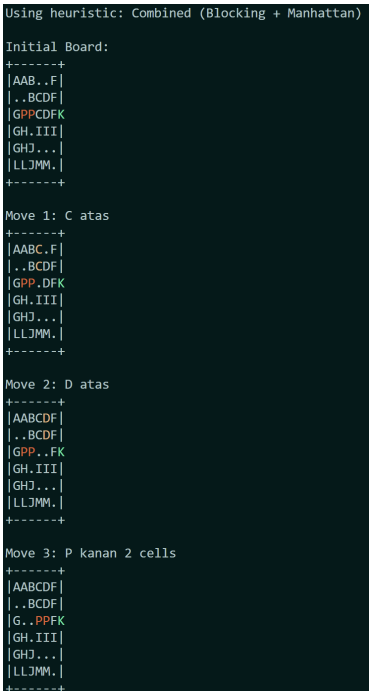
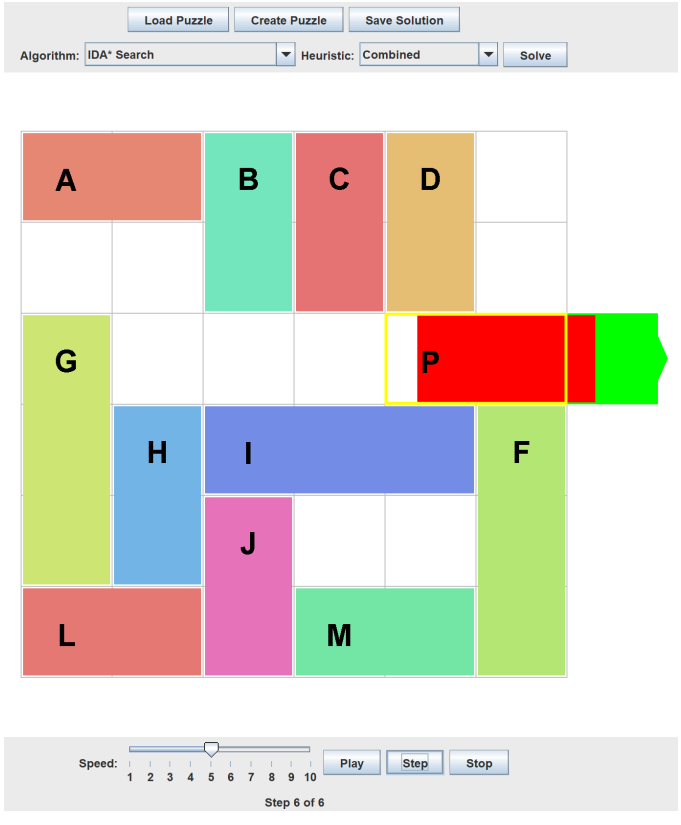
A* (Manhattan)	Using heuristic: Manhattan Distance Initial Board: +------+ |AAB..F| |..BCDF| |GPPCDFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 1: C atas +------+ |AABC.F| |..BCDF| |GPP.DFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 2: D atas +------+ |AABCDF| |..BCDF| |GPP..FK |GH.III| |GHJ...| |LLJMM.| +------+ Move 3: P kanan 2 cells +------+ |AABCDF| |..BCDF| |G..PPFK |GH.III| |GHJ...| |LLJMM.| +------+	

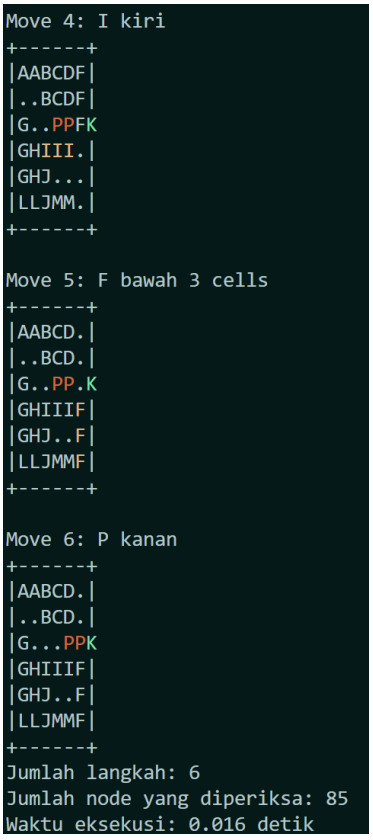
	 Move 4: I kiri +------+ |AABCDF| |..BCDF| |G..PPFK |GHIII.| |GHJ...| |LLJMM.| +------+ Move 5: F bawah 3 cells +------+ |AABCD.| |..BCD.| |G..PP.K |GHIIIF| |GHJ..F| |LLJMMF| +------+ Move 6: P kanan +------+ |AABCD.| |..BCD.| |G...PPK |GHIIIF| |GHJ..F| |LLJMMF| +------+ Jumlah langkah: 6 Jumlah node yang diperiksa: 271 Waktu eksekusi: 0.049 detik	

A* (Combined)	Using heuristic: Combined (Blocking + Manhattan) Initial Board: +------+ |AAB..F| |..BCDF| |GPPCDFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 1: C atas +------+ |AABC.F| |..BCDF| |GPP.DFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 2: D atas +------+ |AABCDF| |..BCDF| |GPP..FK |GH.III| |GHJ...| |LLJMM.| +------+ Move 3: P kanan 2 cells +------+ |AABCDF| |..BCDF| |G..PPFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 4: I kiri +-------+ |AABCDF| |..BCDF| |G..PPFK |GHIII.| |GHJ...| |LLJMM.| +-------+ Move 5: F bawah 3 cells +-------+ |AABCD.| |..BCD.| |G..PP.K |GHIIIF| |GHJ..F| |LLJMMF| +-------+ Move 6: P kanan +-------+ |AABCD.| |..BCD.| |G...PPK |GHIIIF| |GHJ..F| |LLJMMF| +-------+ Jumlah langkah: 6 Jumlah node yang diperiksa: 13 Waktu eksekusi: 0.005 detik	

IDA* (Blocking)	Using heuristic: Blocking Pieces Initial Board: +------+ |AAB..F| |..BCDF| |GPPCDFK |GH.III| |GHJ...| |LLJMM.| +------+ Increasing threshold to: 4 Increasing threshold to: 5 Move 1: C atas +------+ |AABC.F| |..BCDF| |GPP.DFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 2: D atas +------+ |AABCDF| |..BCDF| |GPP..FK |GH.III| |GHJ...| |LLJMM.| +------+ Move 3: I kiri +------+ |AABCDF| |..BCDF| |GPP..FK |GHIII.| |GHJ...| |LLJMM.| +------+ Move 4: F bawah 3 cells +------+ |AABCD.| |..BCD.| |GPP...K |GHIIIF| |GHJ..F| |LLJMMF| +------+ Move 5: P kanan 3 cells +------+ |AABCD.| |..BCD.| |G...PPK |GHIIIF| |GHJ..F| |LLJMMF| +------+ Jumlah langkah: 5 Jumlah node yang diperiksa: 876 Waktu eksekusi: 0.247 detik	

IDA* (Manhattan)	Using heuristic: Manhattan Distance Initial Board: +------+ |AAB..F| |..BCDF| |GPPCDFK |GH.III| |GHJ...| |LLJMM.| +------+ Increasing threshold to: 4 Increasing threshold to: 5 Increasing threshold to: 6 Move 1: C atas +------+ |AABC.F| |..BCDF| |GPP.DFK |GH.III| |GHJ...| |LLJMM.| +------+ Move 2: D atas +------+ |AABCDF| |..BCDF| |GPP..FK |GH.III| |GHJ...| |LLJMM.| +------+ Move 3: P kanan 2 cells +------+ |AABCDF| |..BCDF| |G..PPFK |GH.III| |GHJ...| |LLJMM.| +------+	

		
IDA* (Combined)		

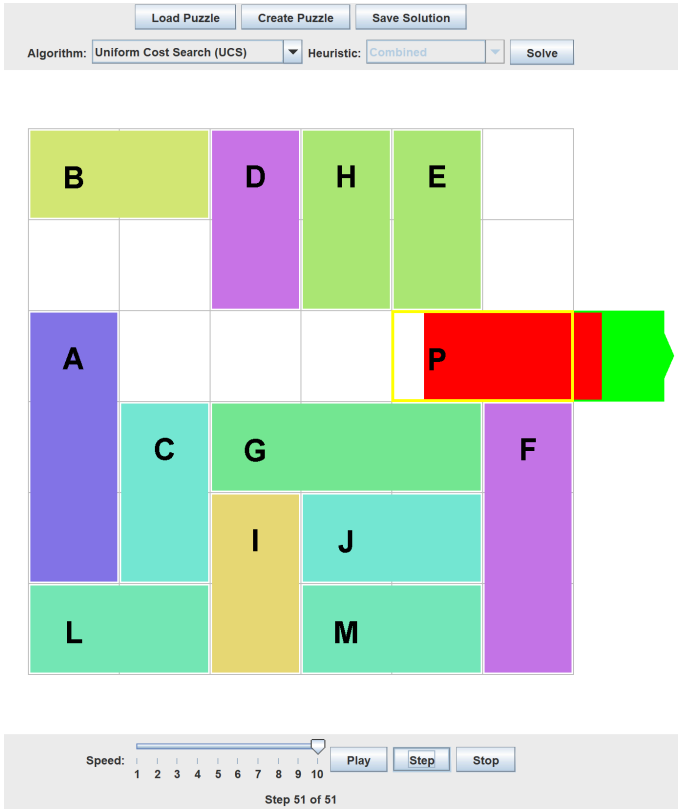
		

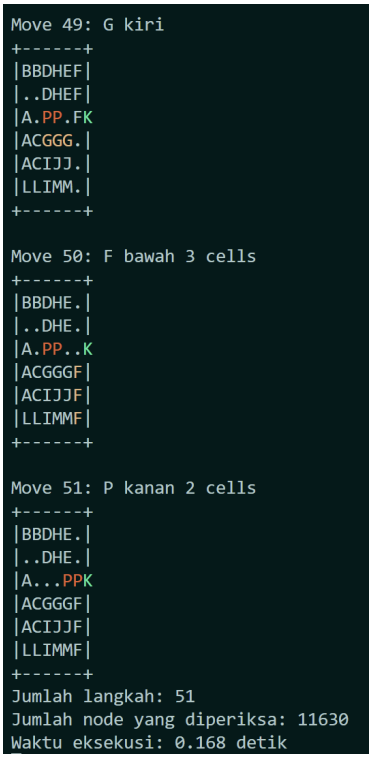
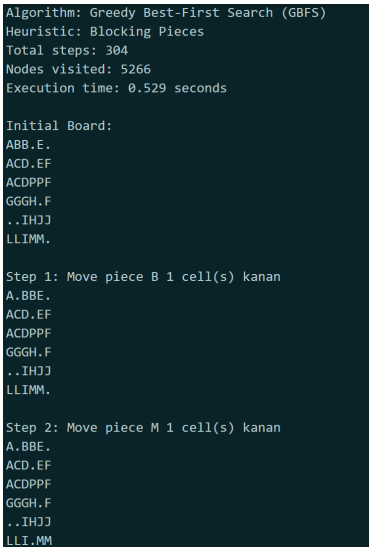
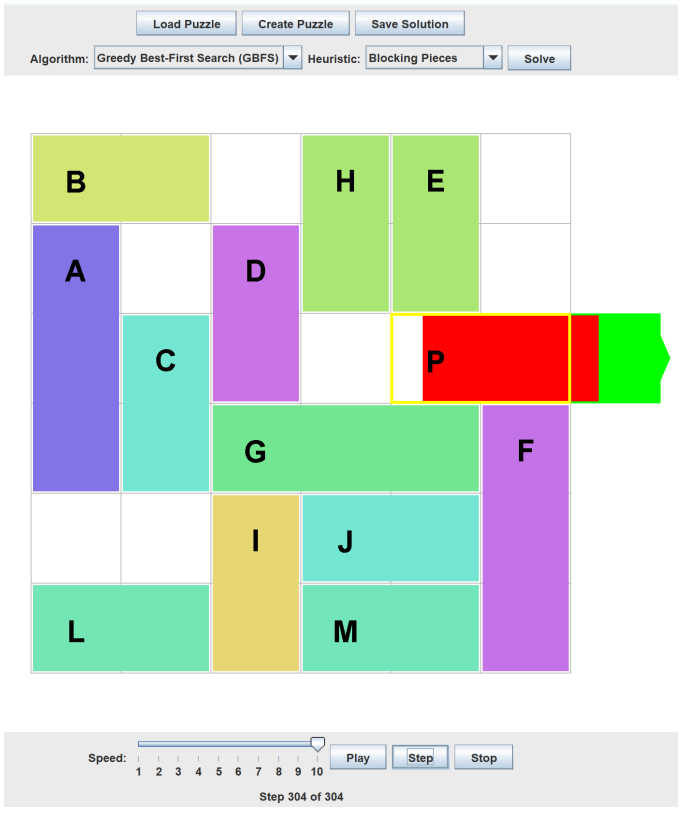
3.2 Kasus 2 (Kompleks)

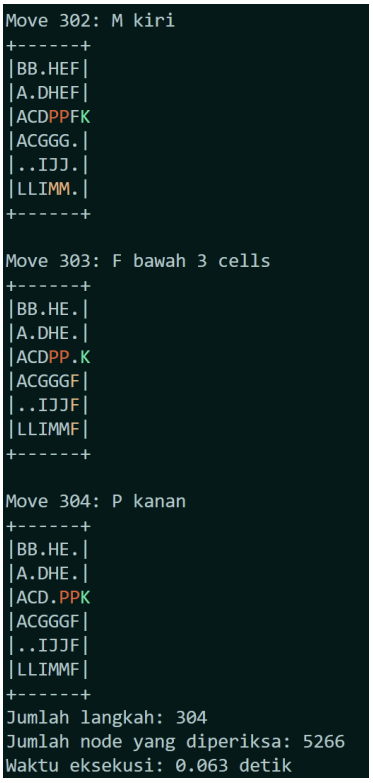
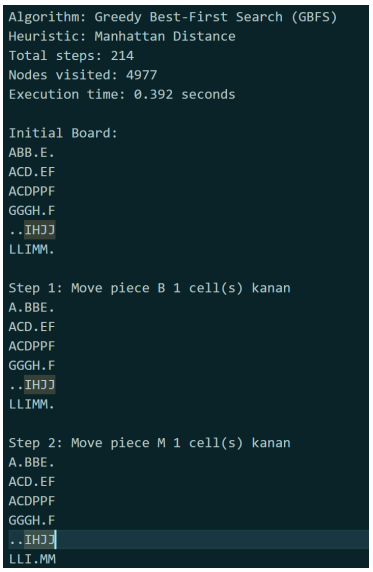
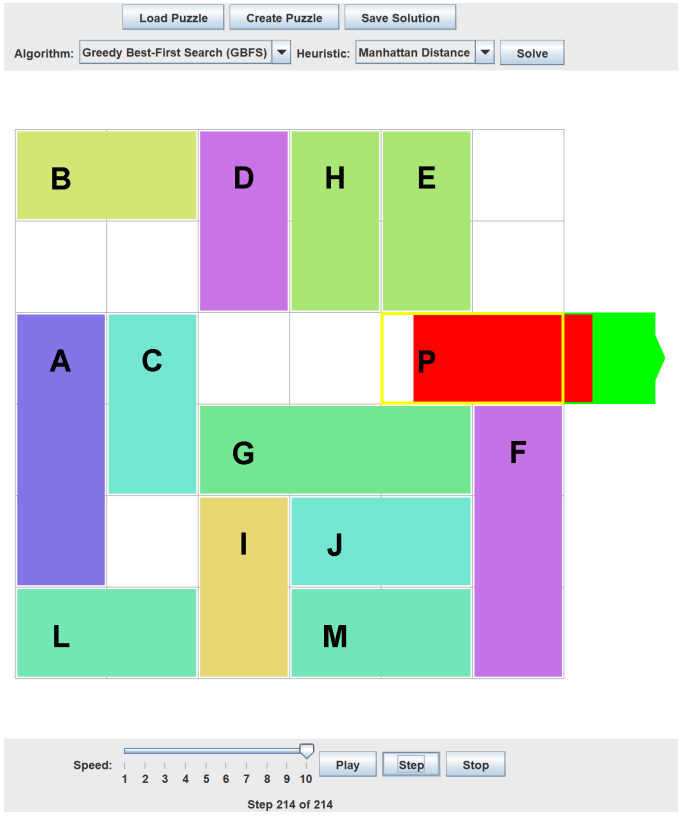
Input:

```
6 6
12
ABB.E.
ACD.EF
ACDPPFK
GGGH.F
..IHJJ
LLIMM.
```

Algoritma (heuristik)	Output CLI	Output GUI

UCS	 Initial Board: +------+ |ABB.E.| |ACD.EF| |ACDPPFK |GGGH.F| |..IHJJ| |LLIMM.| +------+ Move 1: M kanan +------+ |ABB.E.| |ACD.EF| |ACDPPFK |GGGH.F| |..IHJJ| |LLI.MM| +------+ Move 2: H bawah +------+ |ABB.E.| |ACD.EF| |ACDPPFK |GGG..F| |..IHJJ| |LLIHMM| +------+	 Load Puzzle Create Puzzle Save Solution Algorithm: Uniform Cost Search (UCS) Heuristic: Combined Solve B D H E A P C G F I J L M Speed: 1 2 3 4 5 6 7 8 9 10 Play Step Stop Step 51 of 51

		
GBFS (blocking)	 (Pakai di txt karena tidak bisa ss yang awal)	

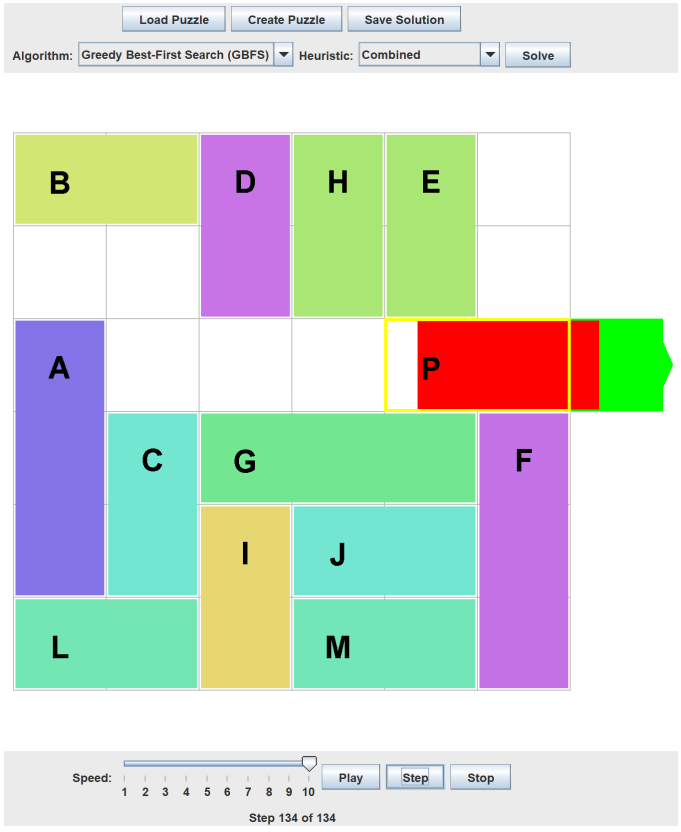
		
GBFS (Manhattan Distance)	 (Pakai dari txt karena tidak bisa terlihat)	

```
Move 212: M kiri
+------+
|BBDHEF|
|..DHEF|
|AC.PPFK
|ACGGG.|
|A.IJJ.|
|LLIMM.|
+------+

Move 213: F bawah 3 cells
+------+
|BBDHE.|
|..DHE.|
|AC.PP.K
|ACGGGF|
|A.IJJF|
|LLIMMF|
+------+

Move 214: P kanan
+------+
|BBDHE.|
|..DHE.|
|AC..PPK
|ACGGGF|
|A.IJJF|
|LLIMMF|
|LLIMMF|
+------+
Jumlah langkah: 214
Jumlah node yang diperiksa: 4977
Waktu eksekusi: 0.115 detik
```

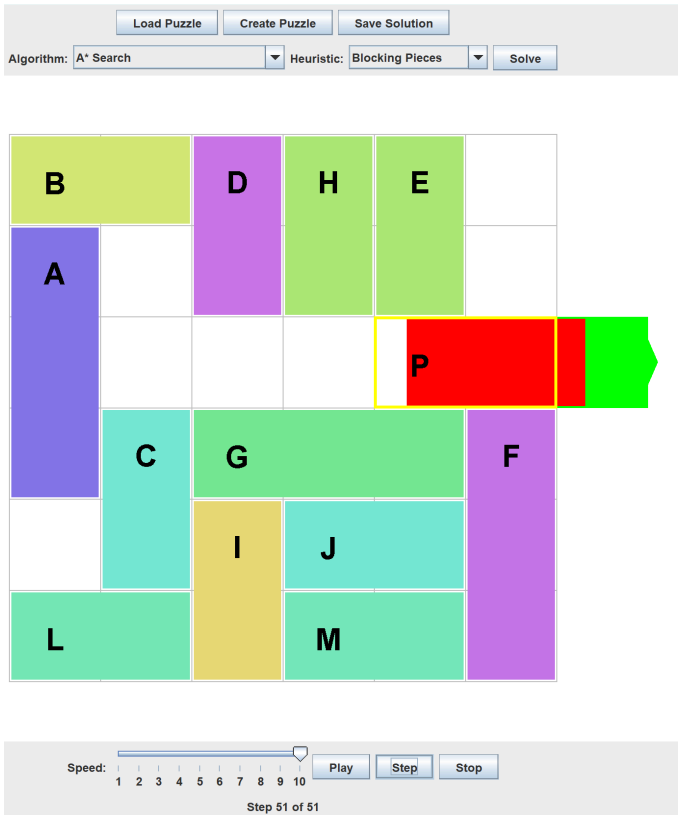
GBFS (Combined)

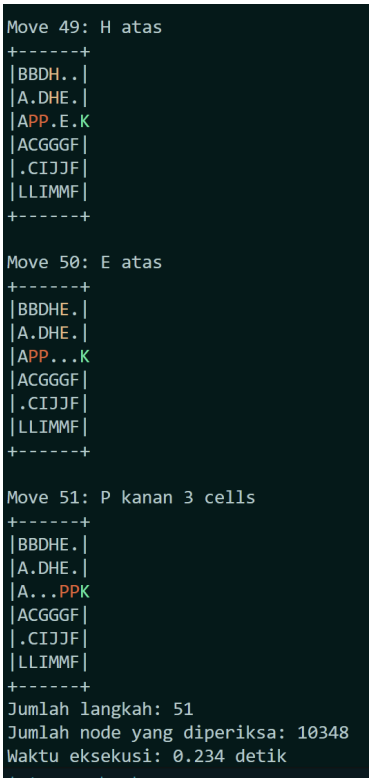
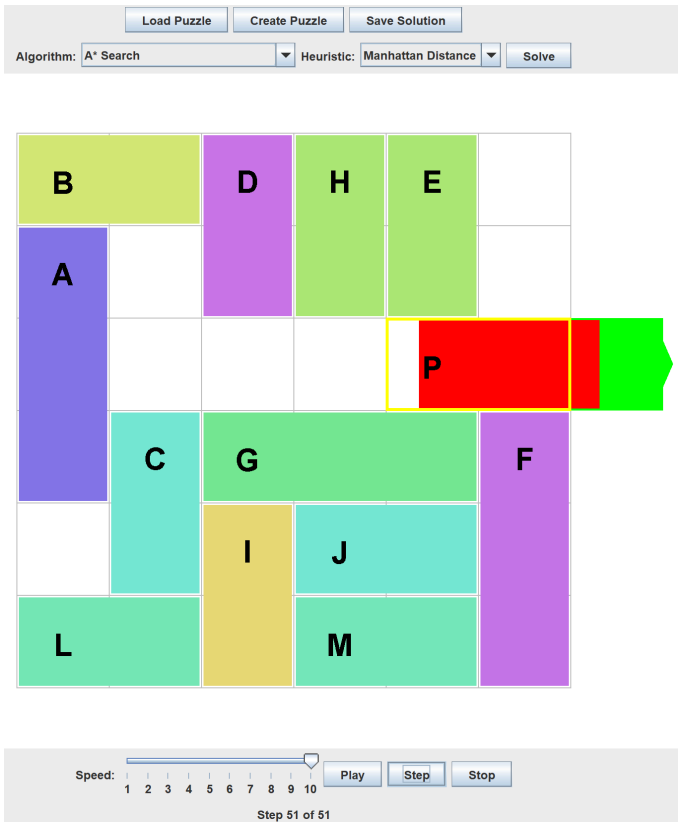
```
Algorithm: Greedy Best-First Search (GBFS)
Heuristic: Combined
Total steps: 134
Nodes visited: 5034
Execution time: 0.62 seconds

Initial Board:
ABB.E.
ACD.EF
ACDPPF
GGGH.F
..IHJJ
LLIMM.

Step 1: Move piece F 1 cell(s) atas
ABB.EF
ACD.EF
ACDPPF
GGGH..
..IHJJ
LLIMM.

Step 2: Move piece M 1 cell(s) kanan
ABB.EF
ACD.EF
ACDPPF
GGGH..
..IHJJ
LLI.MM
```

(Pakai dari txt karena tidak bisa terlihat)

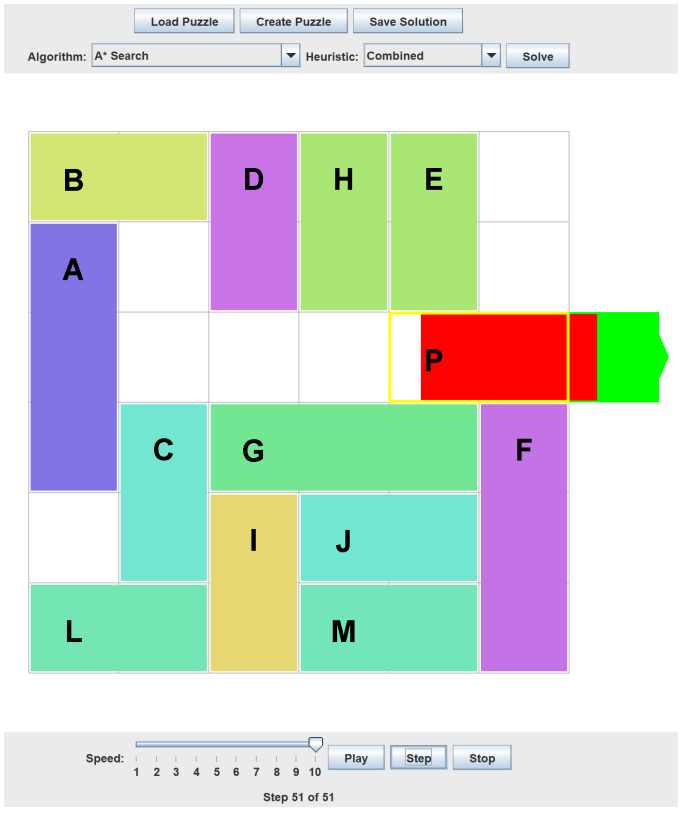


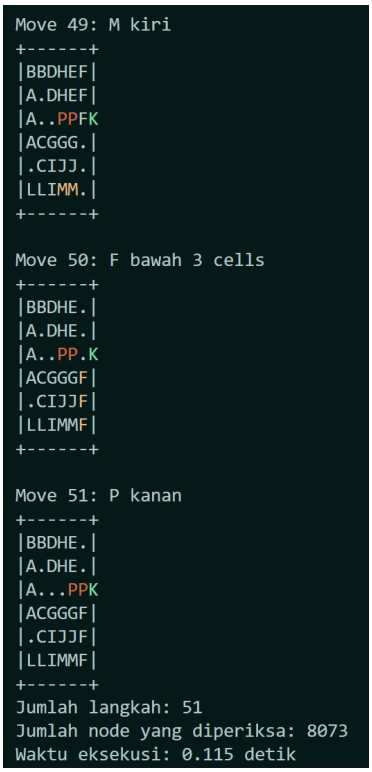
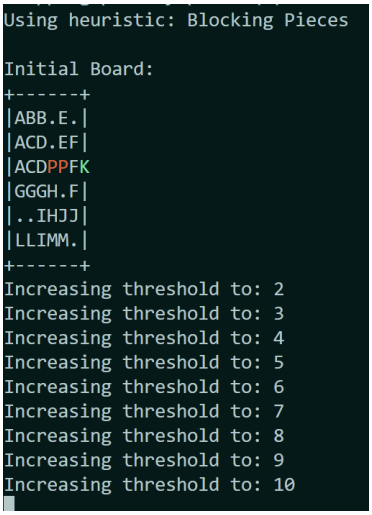
	<pre>Move 132: J kiri
	+------+
	|BBDHEF|
	|..DHEF|
	|A..PPFK
	|ACGGG.|
	|ACIJJ.|
	|LLIMM.|
	+------+
	
	Move 133: F bawah 3 cells
	+------+
	|BBDHE.|
	|..DHE.|
	|A..PP.K
	|ACGGGF|
	|ACIJJF|
	|LLIMMF|
	+------+
	
	Move 134: P kanan
	+------+
	|BBDHE.|
	|..DHE.|
	|A...PPK
	|ACGGGF|
	|ACIJJF|
	|LLIMMF|
	+------+
	Jumlah langkah: 134
	Jumlah node yang diperiksa: 5034
	Waktu eksekusi: 0.304 detik</pre>	

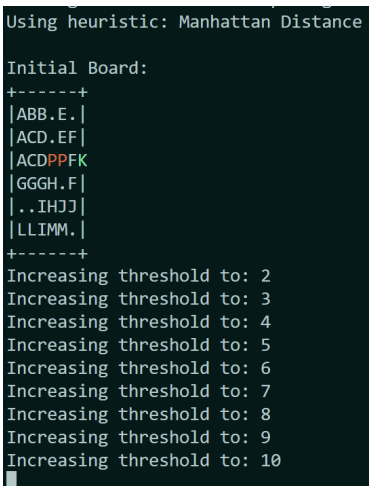
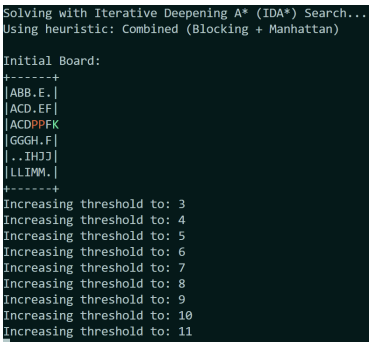
A* (Blocking)	<pre>Using heuristic: Blocking Pieces
	
	Initial Board:
	+------+
	|ABB.E.|
	|ACD.EF|
	|ACDPPFK
	|GGGH.F|
	|..IHJJ|
	|LLIMM.|
	+------+
	
	Move 1: M kanan
	+------+
	|ABB.E.|
	|ACD.EF|
	|ACDPPFK
	|GGGH.F|
	|..IHJJ|
	|LLI.MM|
	+------+
	
	Move 2: H bawah
	+------+
	|ABB.E.|
	|ACD.EF|
	|ACDPPFK
	|GGG..F|
	|..IHJJ|
	|LLIHMM|
	+------+</pre>
		

	 Move 49: H atas +------+ |BBDH..| |A.DHE.| |APP.E.K |ACGGGF| |.CIJJF| |LLIMMF| +------+ Move 50: E atas +------+ |BBDHE.| |A.DHE.| |APP...K |ACGGGF| |.CIJJF| |LLIMMF| +------+ Move 51: P kanan 3 cells +------+ |BBDHE.| |A.DHE.| |A...PPK |ACGGGF| |.CIJJF| |LLIMMF| +------+ Jumlah langkah: 51 Jumlah node yang diperiksa: 10348 Waktu eksekusi: 0.234 detik	
A* (Manhattan)	 Using heuristic: Manhattan Distance Initial Board: +------+ |ABB.E.| |ACD.EF| |ACDPPFK |GGGH.F| |..IHJJ| |LLIMM.| +------+ Move 1: M kanan +------+ |ABB.E.| |ACD.EF| |ACDPPFK |GGGH.F| |..IHJJ| |LLI.MM| +------+ Move 2: H bawah +------+ |ABB.E.| |ACD.EF| |ACDPPFK |GGG..F| |..IHJJ| |LLIHMM| +------+	 Load Puzzle Create Puzzle Save Solution Algorithm: A* Search Heuristic: Manhattan Distance Solve B D H E A P C G F I J L M Speed: 1 2 3 4 5 6 7 8 9 10 Play Step Stop Step 51 of 51

	<pre>Move 49: J kiri
	+------+
	|BBDHEF|
	|A.DHEF|
	|A..PPFK
	|ACGGG.|
	|.CIJJ.|
	|LLIMM.|
	+------+
	
	Move 50: F bawah 3 cells
	+------+
	|BBDHE.|
	|A.DHE.|
	|A..PP.K
	|ACGGGF|
	|.CIJJF|
	|LLIMMF|
	+------+
	
	Move 51: P kanan
	+------+
	|BBDHE.|
	|A.DHE.|
	|A...PPK
	|ACGGGF|
	|.CIJJF|
	|LLIMMF|
	+------+
	Jumlah langkah: 51
	Jumlah node yang diperiksa: 10033
	Waktu eksekusi: 0.264 detik</pre>	

A* (Combined)	<pre>Using heuristic: Combined (Blocking + Manhattan)
	
	Initial Board:
	+------+
	|ABB.E.|
	|ACD.EF|
	|ACDPPFK
	|GGGH.F|
	|..IHJJ|
	|LLIMM.|
	+------+
	
	Move 1: M kanan
	+------+
	|ABB.E.|
	|ACD.EF|
	|ACDPPFK
	|GGGH.F|
	|..IHJJ|
	|LLI.MM|
	+------+
	
	Move 2: H bawah
	+------+
	|ABB.E.|
	|ACD.EF|
	|ACDPPFK
	|GGG..F|
	|..IHJJ|
	|LLIHMM|
	+------+</pre>
		

		
IDA* (Blocking)		-

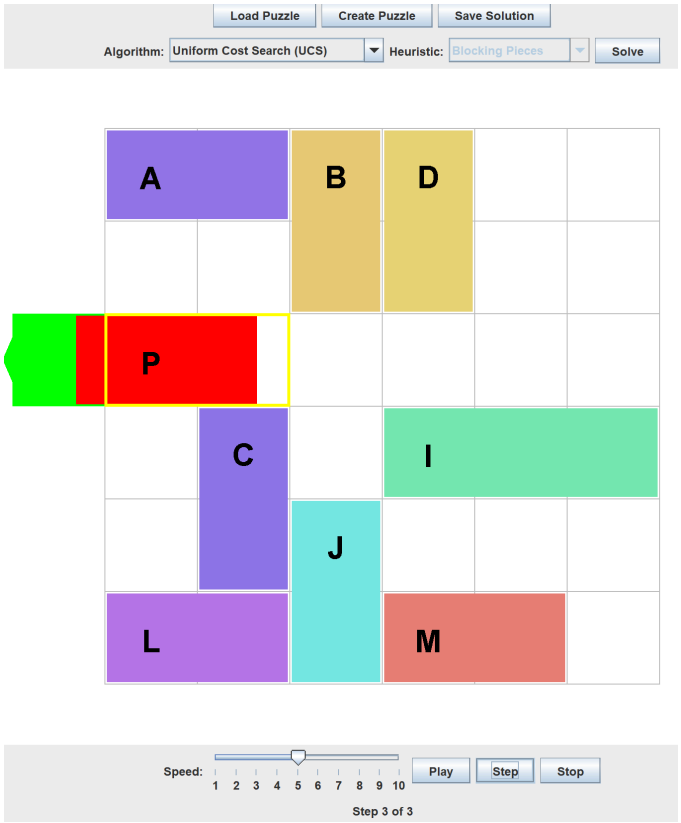
IDA* (Manhattan)		-
IDA* (Combined)		-

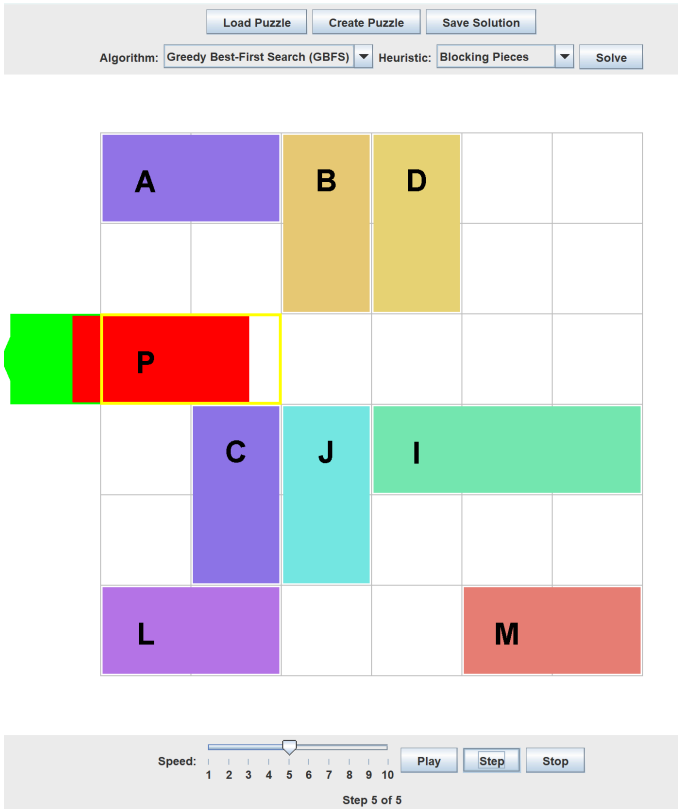
3.3 Kasus 3 (Exit di kiri)

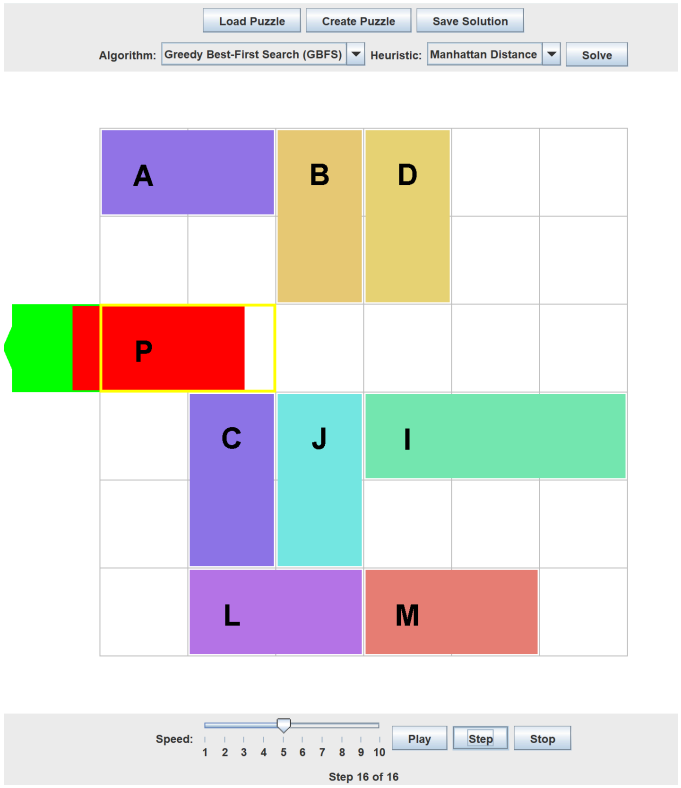
Input:

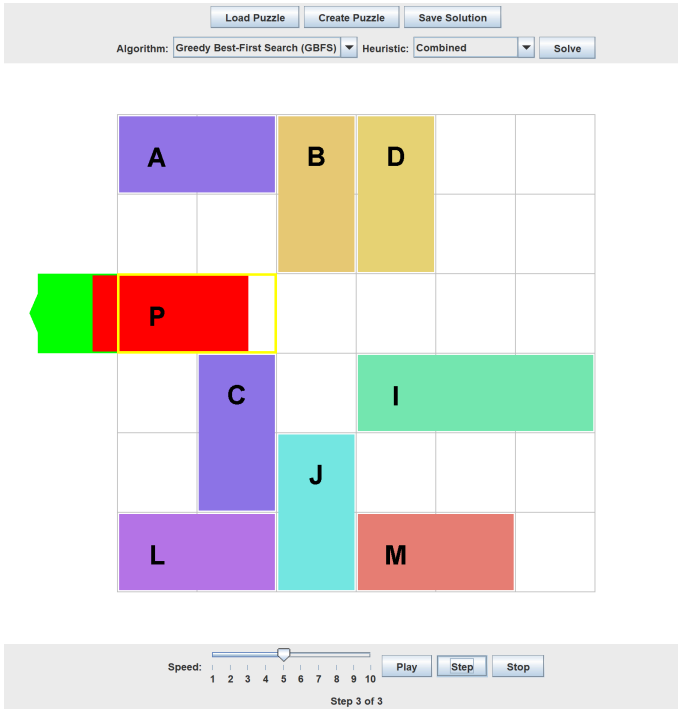
```
6 6
8
 AAB...
 ..BD..
K.C.DPP
 .C.III
 ..J...
 LLJMM.
```

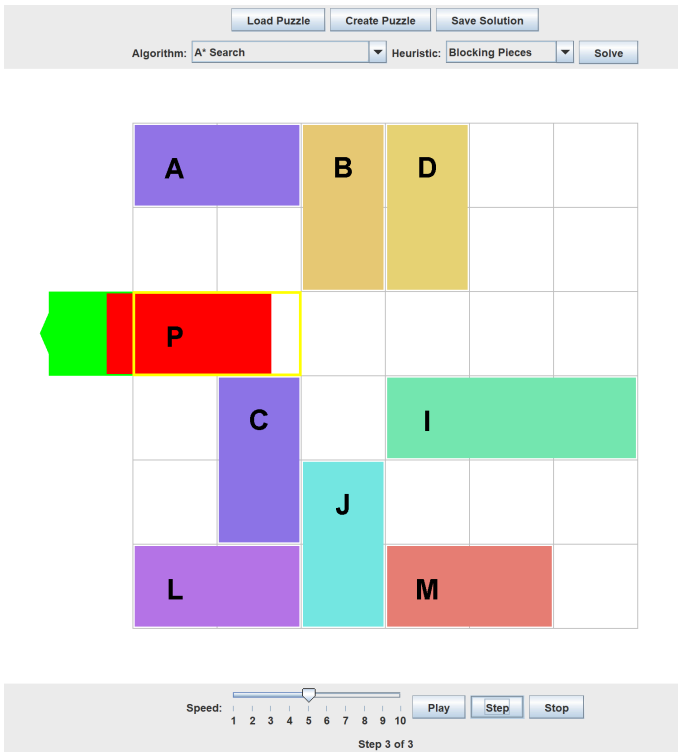
Algoritma (heuristik)	Output CLI	Output GUI

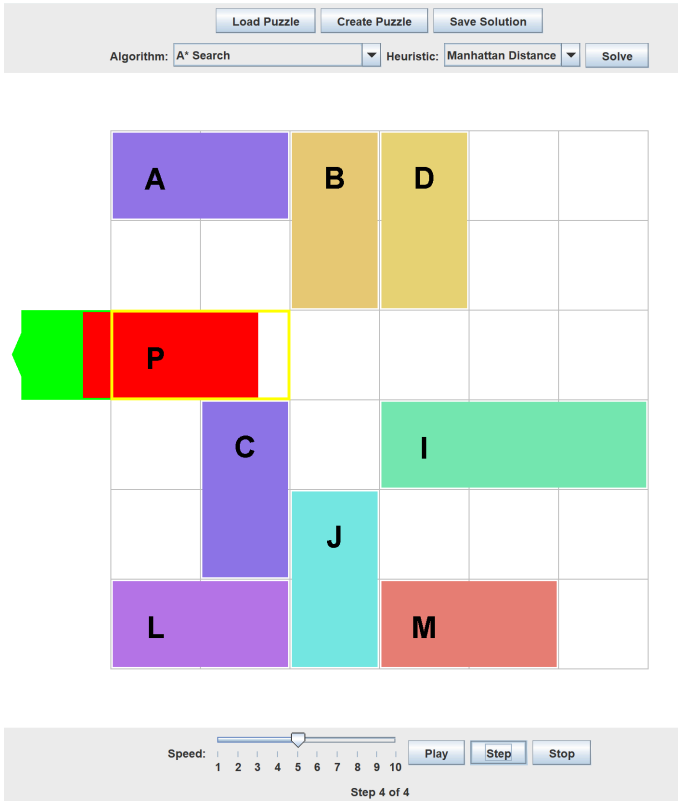
UCS	Initial Board: +------+ |AAB...| |..BD..| K.C.DPP| |.C.III| |..J...| |LLJMM.| +------+ Move 1: C bawah +------+ |AAB...| |..BD..| K...DPP| |.C.III| |.CJ...| |LLJMM.| +------+ Move 2: D atas +------+ |AABD..| |..BD..| K....PP| |.C.III| |.CJ...| |LLJMM.| +------+ Move 3: P kiri 4 cells +------+ |AABD..| |..BD..| KPP....| |.C.III| |.CJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 104 Waktu eksekusi: 0.008 detik	

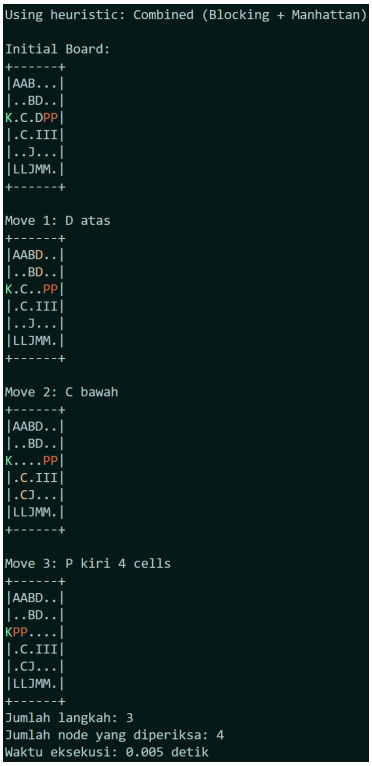
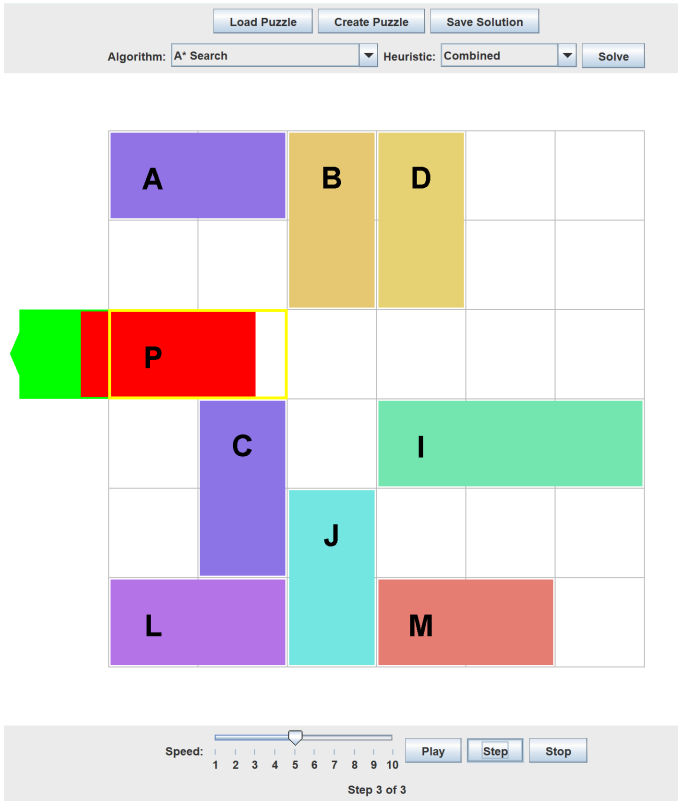
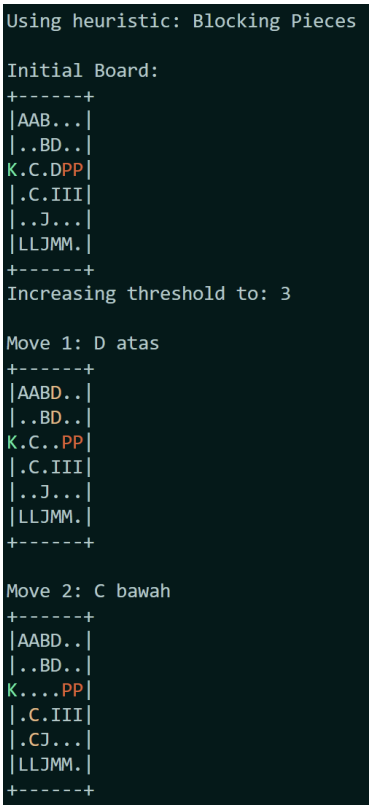
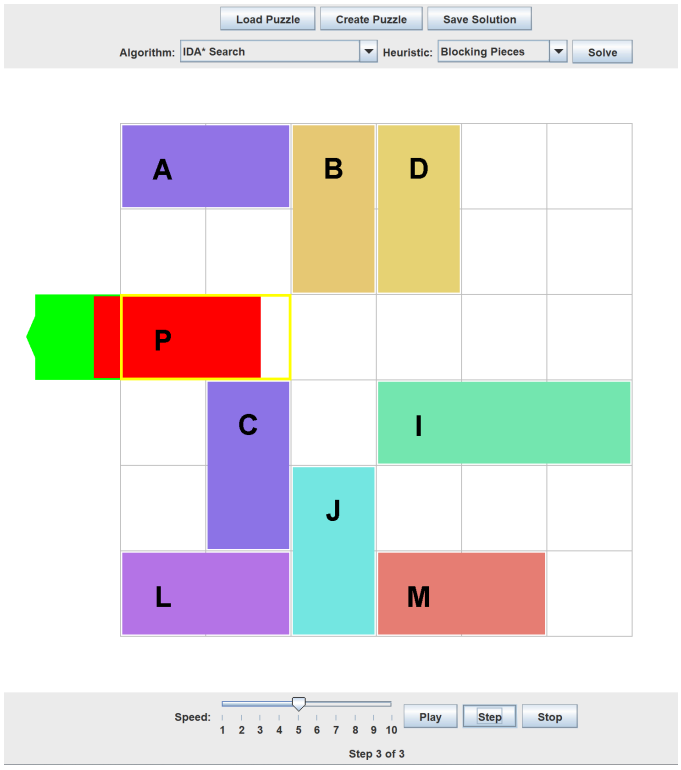
GBFS (blocking)	Using heuristic: Blocking Pieces Initial Board: +------+ |AAB...| |..BD..| K.C.DPP| |.C.III| |..J...| |LLJMM.| +------+ Move 1: D atas +------+ |AABD..| |..BD..| K.C..PP| |.C.III| |..J...| |LLJMM.| +------+ Move 2: M kanan +------+ |AABD..| |..BD..| K.C..PP| |.C.III| |..J...| |LLJ.MM| +------+ Move 3: J atas +------+ |AABD..| |..BD..| K.C..PP| |.CJIII| |..J...| |LL..MM| +------+ Move 4: C bawah +------+ |AABD..| |..BD..| K....PP| |.CJIII| |.CJ...| |LL..MM| +------+ Move 5: P kiri 4 cells +------+ |AABD..| |..BD..| KPP....| |.CJIII| |.CJ...| |LL..MM| +------+ Jumlah langkah: 5 Jumlah node yang diperiksa: 22 Waktu eksekusi: 0.017 detik	

GBFS (Manhattan Distance)	Using heuristic: Manhattan Distance Initial Board: +------+ |AAB...| |..BD..| K.C.DPP| |.C.III| |..J...| |LLJMM.| +------+ Move 1: M kanan +------+ |AAB...| |..BD..| K.C.DPP| |.C.III| |..J...| |LLJ.MM| +------+ Move 2: J atas +------+ |AAB...| |..BD..| K.C.DPP| |.CJIII| |..J...| |LL..MM| +------+ Move 15: D atas +------+ |AABD..| |..BD..| K....PP| |.CJIII| |.CJ...| |.LLMM.| +------+ Move 16: P kiri 4 cells +------+ |AABD..| |..BD..| KPP....| |.CJIII| |.CJ...| |.LLMM.| +------+ Jumlah langkah: 16 Jumlah node yang diperiksa: 31 Waktu eksekusi: 0.008 detik	

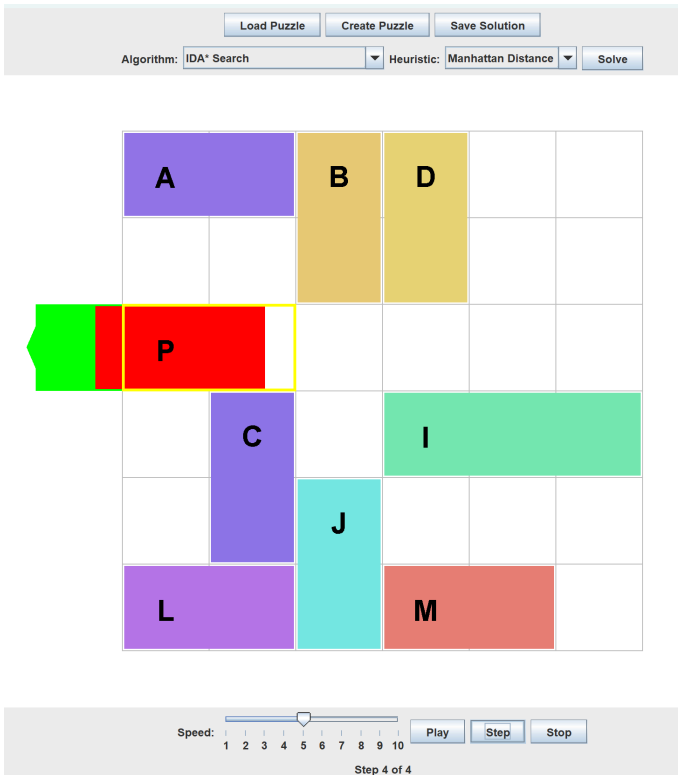
GBFS (Combined)	Using heuristic: Combined (Blocking + Manhattan) Initial Board: +------+ |AAB...| |..BD..| K.C.DPP| |.C.III| |..J...| |LLJMM.| +------+ Move 1: D atas +------+ |AABD..| |..BD..| K.C..PP| |.C.III| |..J...| |LLJMM.| +------+ Move 2: C bawah +------+ |AABD..| |..BD..| K....PP| |.C.III| |.CJ...| |LLJMM.| +------+ Move 3: P kiri 4 cells +------+ |AABD..| |..BD..| KPP...| |.C.III| |.CJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 4 Waktu eksekusi: 0.019 detik	

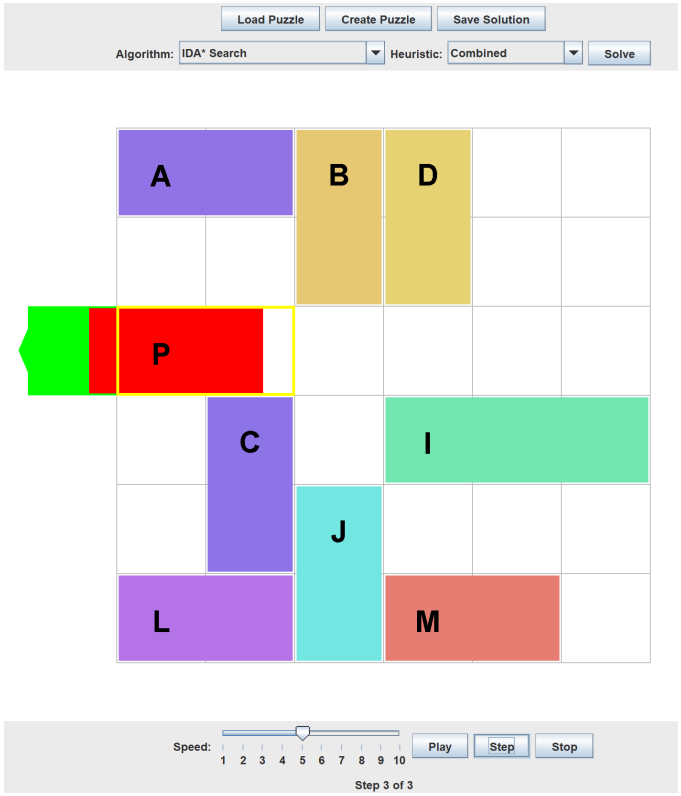
A* (Blocking)	Using heuristic: Blocking Pieces Initial Board: +------+ |AAB...| |..BD..| K.C.DPP| |.C.III| |..J...| |LLJMM.| +------+ Move 1: D atas +------+ |AABD..| |..BD..| K.C..PP| |.C.III| |..J...| |LLJMM.| +------+ Move 2: C bawah +------+ |AABD..| |..BD..| K....PP| |.C.III| |.CJ...| |LLJMM.| +------+ Move 3: P kiri 4 cells +------+ |AABD..| |..BD..| KPP....| |.C.III| |.CJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 24 Waktu eksekusi: 0.011 detik	

A* (Manhattan)	Using heuristic: Manhattan Distance Initial Board: +------+ |AAB...| |..BD..| K.C.DPP| |.C.III| |..J...| |LLJMM.| +------+ Move 1: D atas +------+ |AABD..| |..BD..| K.C..PP| |.C.III| |..J...| |LLJMM.| +------+ Move 2: P kiri 2 cells +------+ |AABD..| |..BD..| K.CPP..| |.C.III| |..J...| |LLJMM.| +------+ Move 3: C bawah +------+ |AABD..| |..BD..| K..PP..| |.C.III| |.CJ...| |LLJMM.| +------+ Move 4: P kiri 2 cells +------+ |AABD..| |..BD..| KPP....| |.C.III| |.CJ...| |LLJMM.| +------+ Jumlah langkah: 4 Jumlah node yang diperiksa: 13 Waktu eksekusi: 0.008 detik	

A* (Combined)		
IDA* (Blocking)		

	<pre>Move 3: P kiri 4 cells
	+------+
	|AABD..|
	|..BD..|
	KPP....|
	|.C.III|
	|.CJ...|
	|LLJMM.|
	+------+
	Jumlah langkah: 3
	Jumlah node yang diperiksa: 88
	Waktu eksekusi: 0.011 detik</pre>	

IDA* (Manhattan)	<pre>Using heuristic: Manhattan Distance
	
	Initial Board:
	+------+
	|AAB...|
	|..BD..|
	K.C.DPP|
	|.C.III|
	|..J...|
	|LLJMM.|
	+------+
	Increasing threshold to: 5
	
	Move 1: D atas
	+------+
	|AABD..|
	|..BD..|
	K.C..PP|
	|.C.III|
	|..J...|
	|LLJMM.|
	+------+
	
	Move 2: P kiri 2 cells
	+------+
	|AABD..|
	|..BD..|
	K.CPP..|
	|.C.III|
	|..J...|
	|LLJMM.|
	+------+
	
	Move 3: C bawah
	+------+
	|AABD..|
	|..BD..|
	K..PP..|
	|.C.III|
	|.CJ...|
	|LLJMM.|
	+------+</pre>
	<pre>Move 4: P kiri 2 cells
	+------+
	|AABD..|
	|..BD..|
	KPP....|
	|.C.III|
	|.CJ...|
	|LLJMM.|
	+------+
	Jumlah langkah: 4
	Jumlah node yang diperiksa: 51
	Waktu eksekusi: 0.006 detik</pre>
		

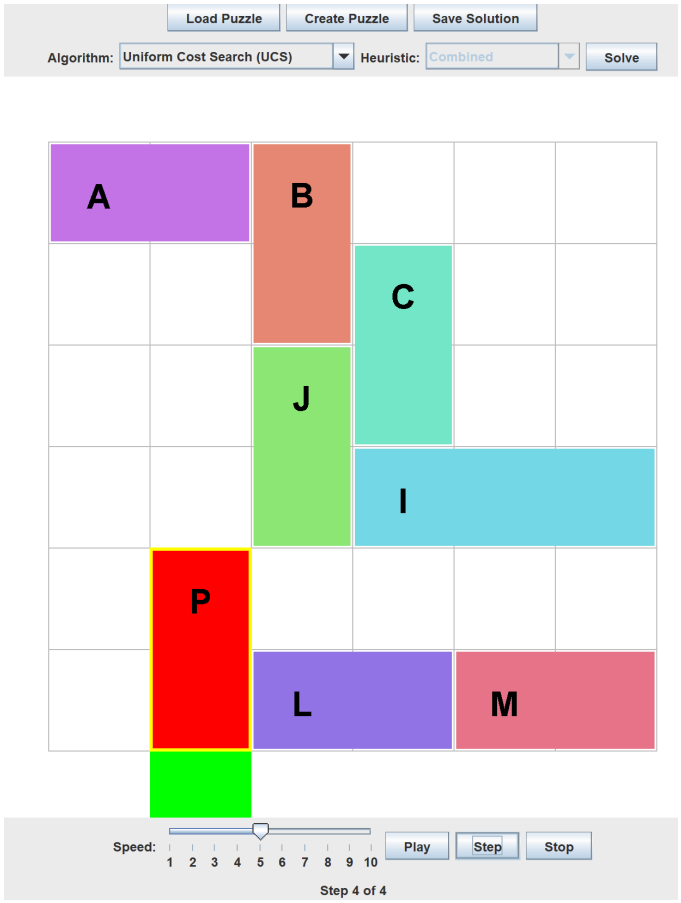
IDA* (Combined)	Using heuristic: Combined (Blocking + Manhattan) Initial Board: +------+ |AAB...| |..BD..| K.C.DPP| |.C.III| |..J...| |LLJMM.| +------+ Move 1: D atas +------+ |AABD..| |..BD..| K.C..PP| |.C.III| |..J...| |LLJMM.| +------+ Move 2: C bawah +------+ |AABD..| |..BD..| K....PP| |.C.III| |.CJ...| |LLJMM.| +------+ Move 3: P kiri 4 cells +------+ |AABD..| |..BD..| KPP...| |.C.III| |.CJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 13 Waktu eksekusi: 0.01 detik	

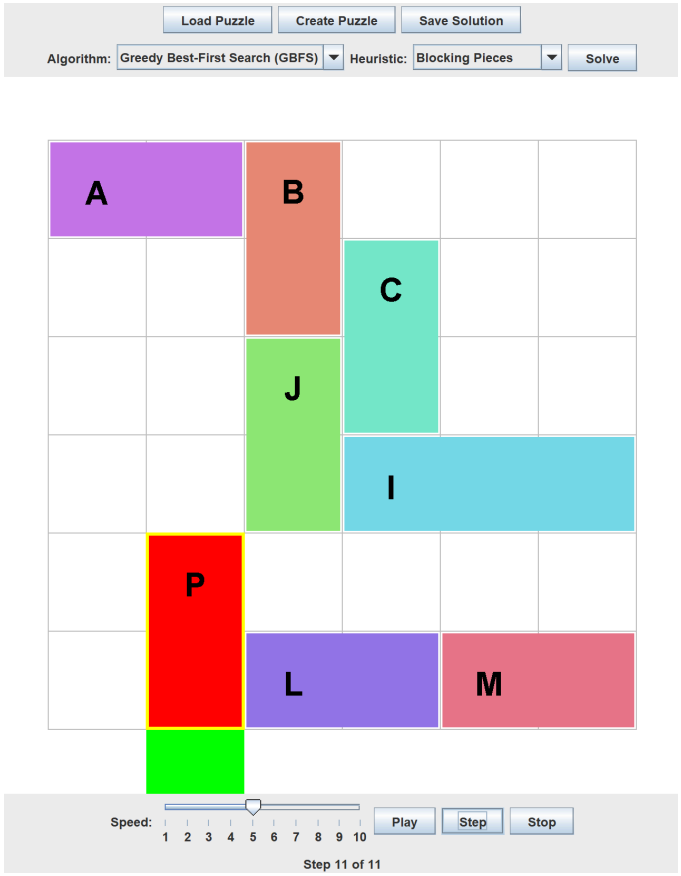
3.4 Kasus 4 (Exit di bawah)

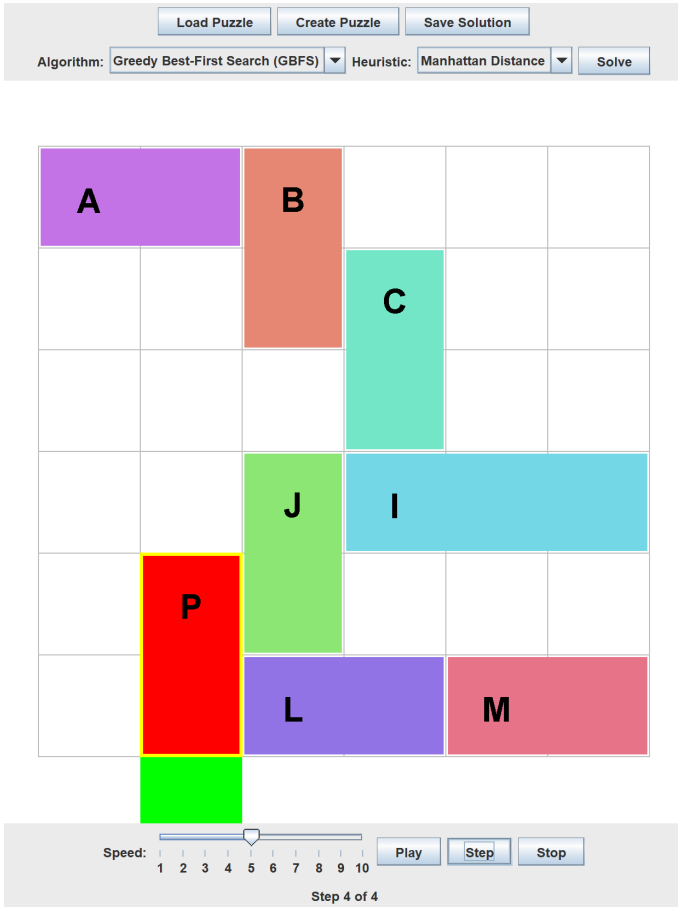
Input:

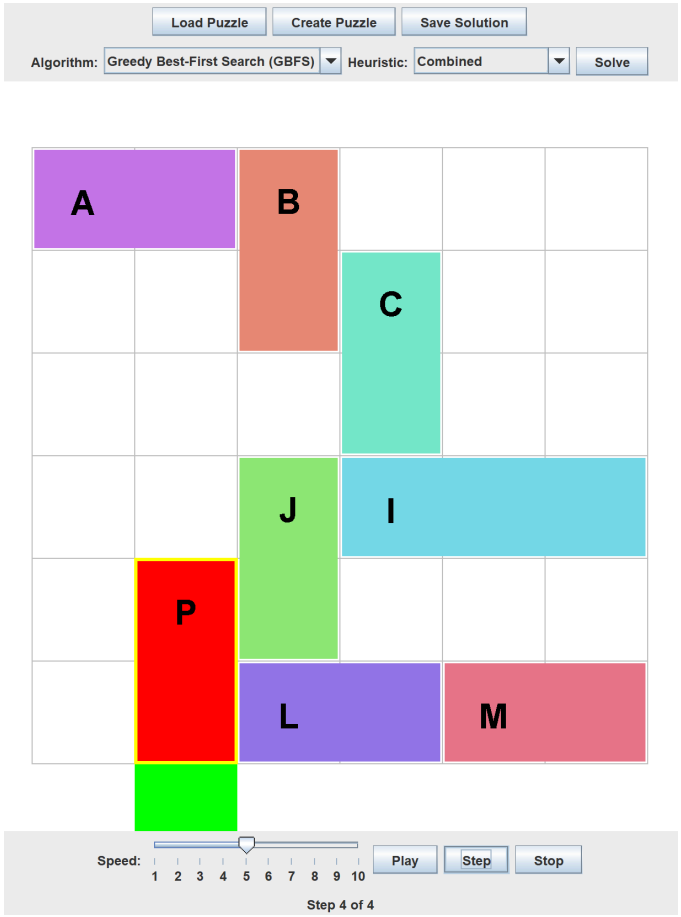
```
6 6
7
AAB...
..BC..
...C..
.P.III
.PJ...
LLJMM.
 K
```

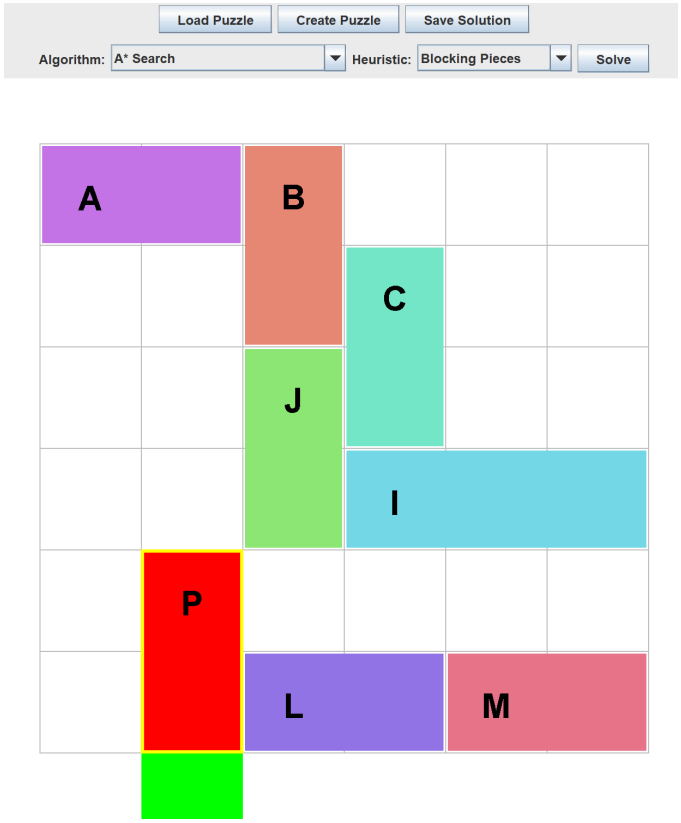
Algoritma (heuristik)	Output CLI	Output GUI

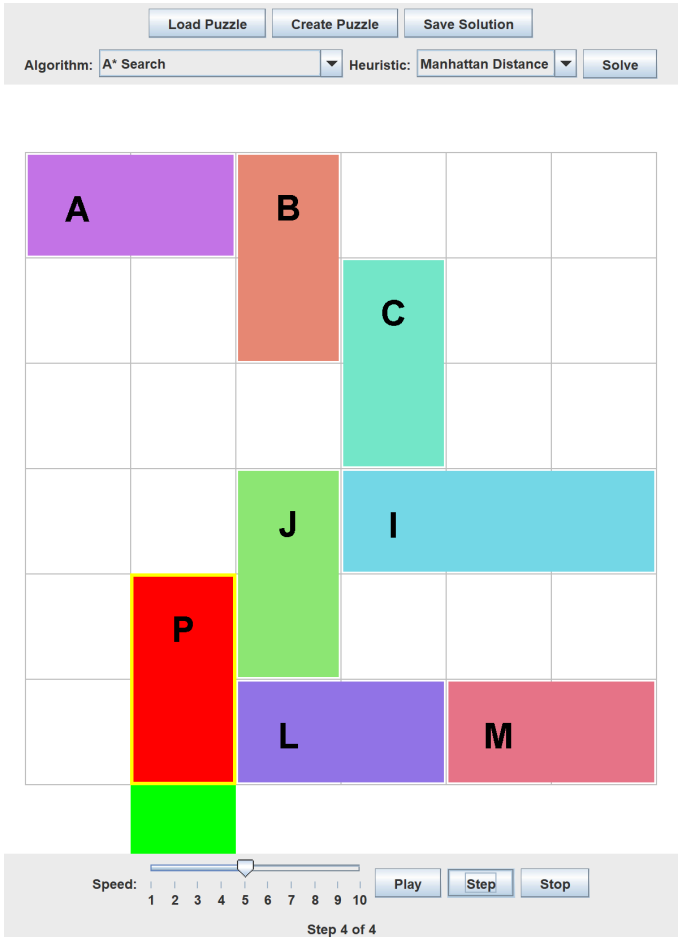
UCS	Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Move 1: J atas 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL.MM.| +-K----+ Move 2: M kanan +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |..JC..| |..JIII| |.P....| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 576 Waktu eksekusi: 0.019 detik	

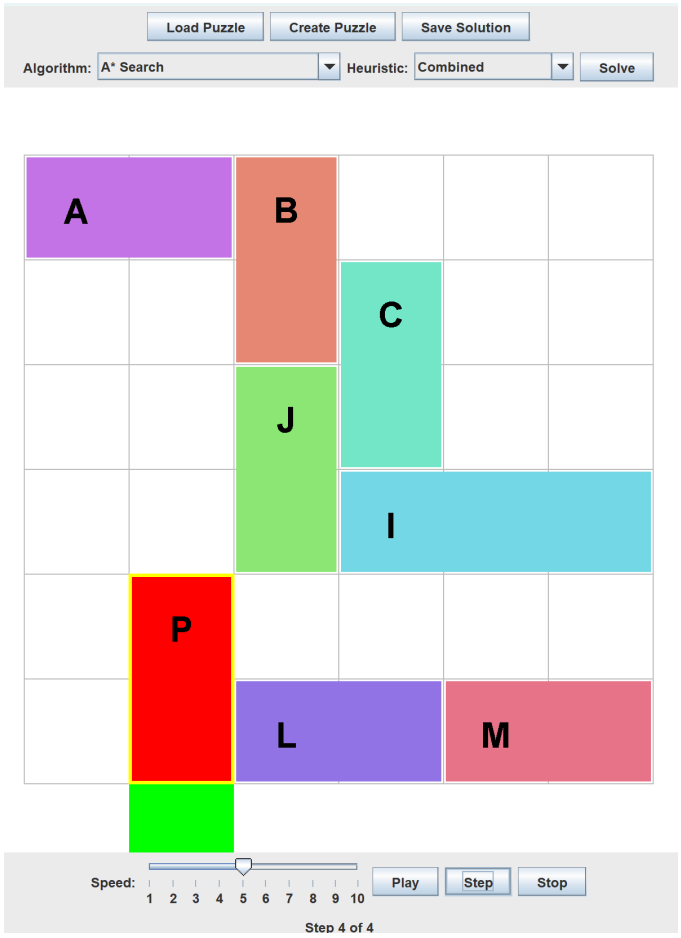
GBFS (blocking)	Using heuristic: Blocking Pieces Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Move 1: M kanan +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJ.MM| +-K----+ Move 2: J atas +------+ |AAB...| |..BC..| |...C..| |.PJIII| |.PJ...| |LL..MM| +-K----+ Move 9: M kanan +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |.LL.MM| +-K----+ Move 10: L kanan +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |..LLMM| +-K----+ Move 11: P bawah +------+ |AAB...| |..BC..| |..JC..| |..JIII| |.P....| |.PLLMM| +-K----+ Jumlah langkah: 11 Jumlah node yang diperiksa: 22 Waktu eksekusi: 0.004 detik	

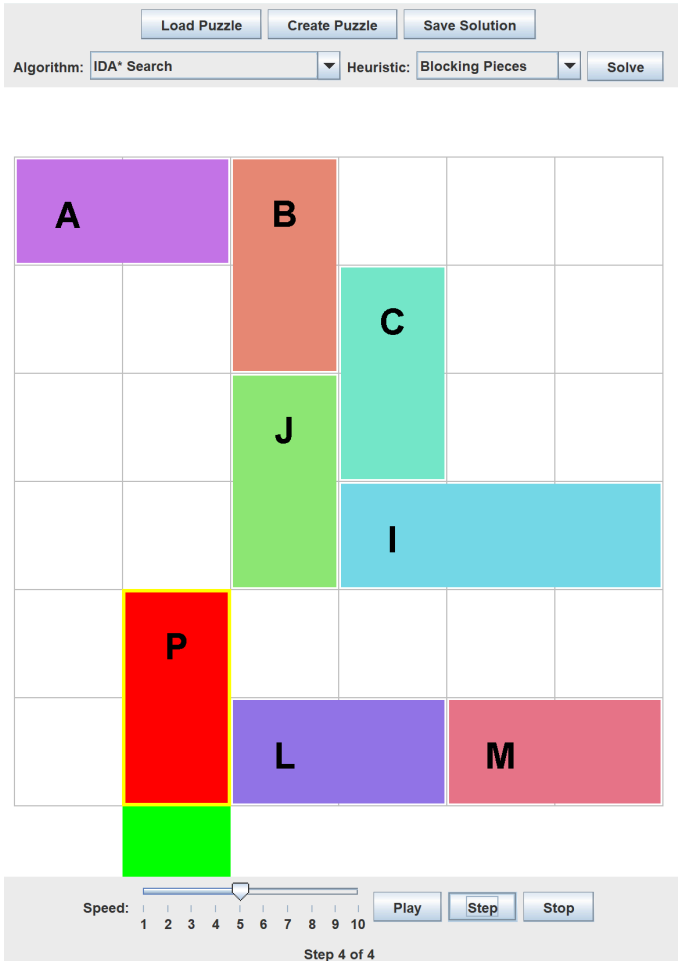
GBFS (Manhattan Distance)	Using heuristic: Manhattan Distance Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Move 1: M kanan +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJ.MM| +-K----+ Move 2: J atas +------+ |AAB...| |..BC..| |...C..| |.PJIII| |.PJ...| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |...C..| |.PJIII| |.PJ...| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |...C..| |..JIII| |.PJ...| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 7 Waktu eksekusi: 0.006 detik	

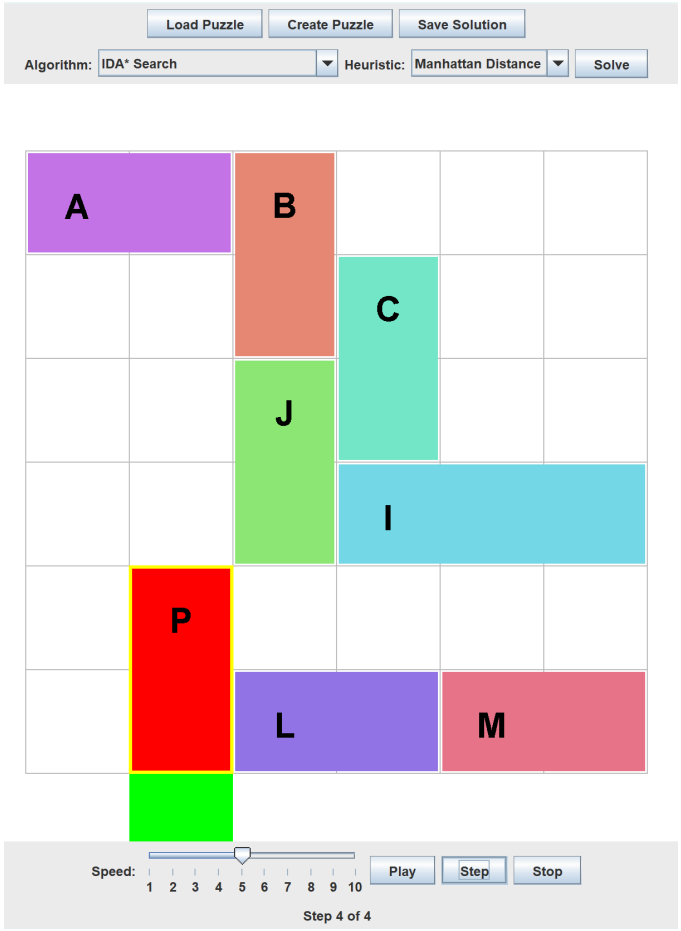
GBFS (Combined)	Using heuristic: Combined (Blocking + Manhattan) Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Move 1: M kanan +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJ.MM| +-K----+ Move 2: J atas +------+ |AAB...| |..BC..| |...C..| |.PJIII| |.PJ...| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |...C..| |.PJIII| |.PJ...| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |...C..| |..JIII| |.PJ...| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 7 Waktu eksekusi: 0.007 detik	

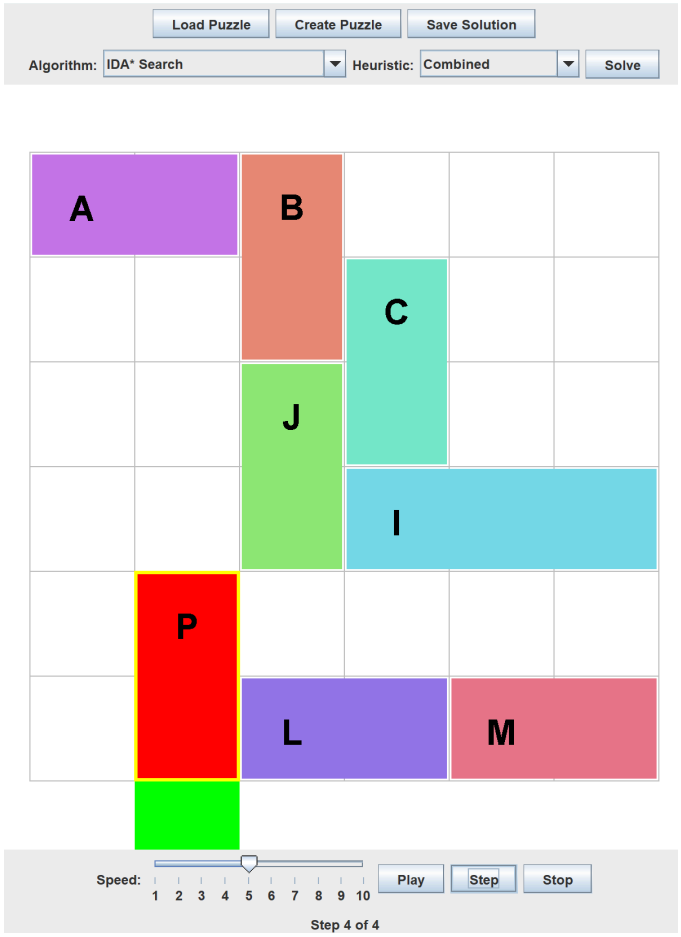
A* (Blocking)	Using heuristic: Blocking Pieces Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Move 1: J atas 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL.MM.| +-K----+ Move 2: M kanan +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |..JC..| |..JIII| |.P....| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 237 Waktu eksekusi: 0.031 detik	

A* (Manhattan)	Using heuristic: Manhattan Distance Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Move 1: M kanan +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJ.MM| +-K----+ Move 2: J atas +------+ |AAB...| |..BC..| |...C..| |.PJIII| |.PJ...| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |...C..| |.PJIII| |.PJ...| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |...C..| |..JIII| |.PJ...| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 75 Waktu eksekusi: 0.011 detik	

A* (Combined)	Using heuristic: Combined (Blocking + Manhattan) Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Move 1: J atas 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL.MM.| +-K----+ Move 2: M kanan +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |..JC..| |..JIII| |.P....| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 12 Waktu eksekusi: 0.008 detik	

IDA* (Blocking)	Using heuristic: Blocking Pieces Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Increasing threshold to: 2 Increasing threshold to: 3 Increasing threshold to: 4 Move 1: J atas 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL.MM.| +-K----+ Move 2: M kanan +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |..JC..| |..JIII| |.P....| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 5275 Waktu eksekusi: 0.116 detik	

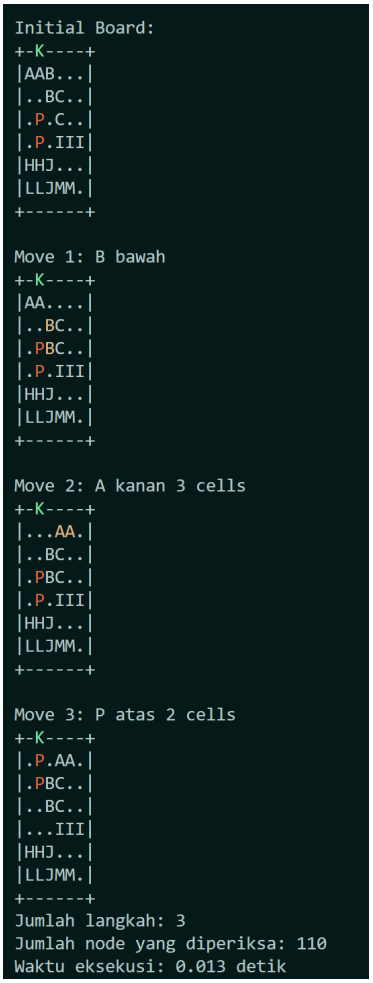
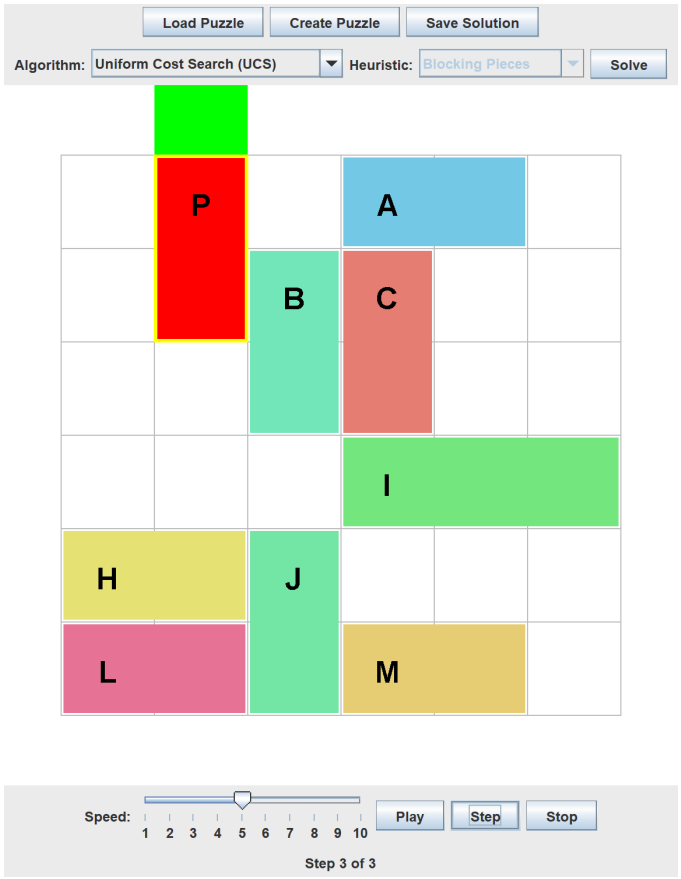
IDA* (Manhattan)	Using heuristic: Manhattan Distance Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Increasing threshold to: 2 Increasing threshold to: 3 Increasing threshold to: 4 Move 1: J atas 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL.MM.| +-K----+ Move 2: M kanan +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |..JC..| |..JIII| |.P....| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 2388 Waktu eksekusi: 0.052 detik	

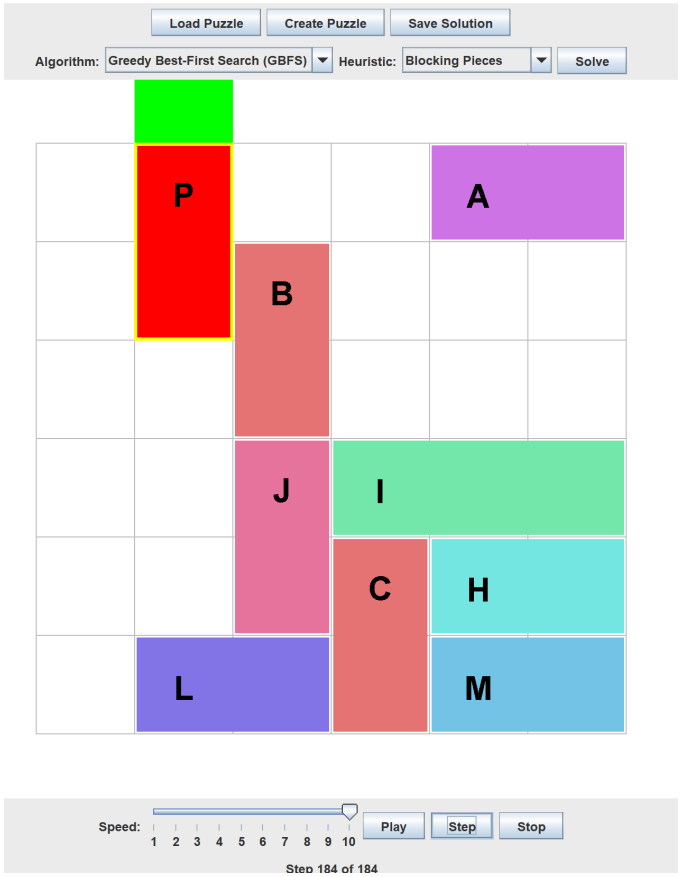
IDA* (Combined)	Using heuristic: Combined (Blocking + Manhattan) Initial Board: +------+ |AAB...| |..BC..| |...C..| |.P.III| |.PJ...| |LLJMM.| +-K----+ Increasing threshold to: 3 Increasing threshold to: 4 Move 1: J atas 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL.MM.| +-K----+ Move 2: M kanan +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |LL..MM| +-K----+ Move 3: L kanan 2 cells +------+ |AAB...| |..BC..| |..JC..| |.PJIII| |.P....| |..LLMM| +-K----+ Move 4: P bawah +------+ |AAB...| |..BC..| |..JC..| |..JIII| |.P....| |.PLLMM| +-K----+ Jumlah langkah: 4 Jumlah node yang diperiksa: 378 Waktu eksekusi: 0.006 detik	

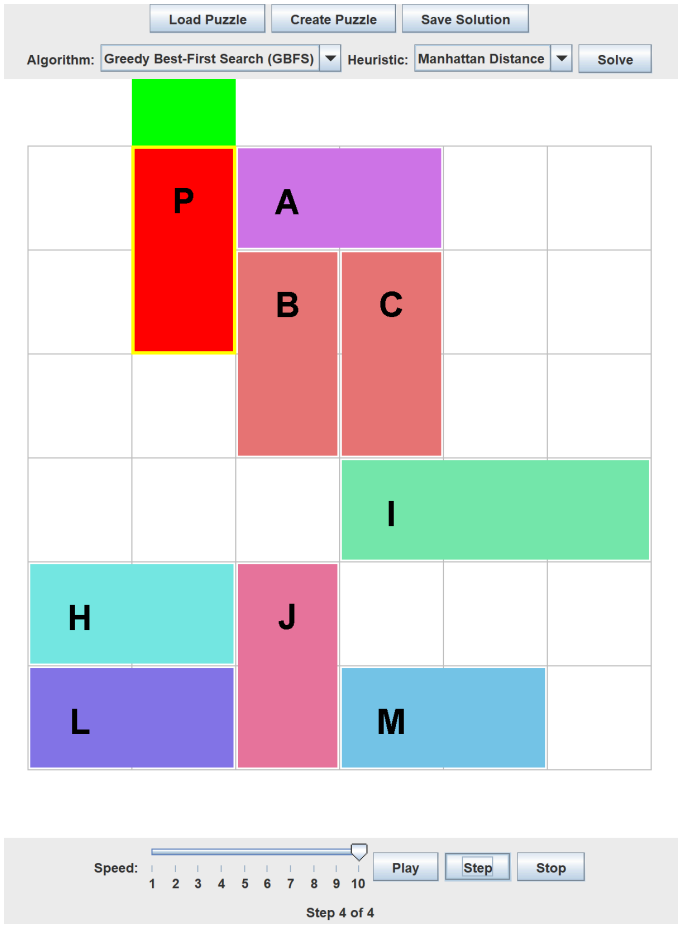
3.5 Kasus 5 (Exit di atas)

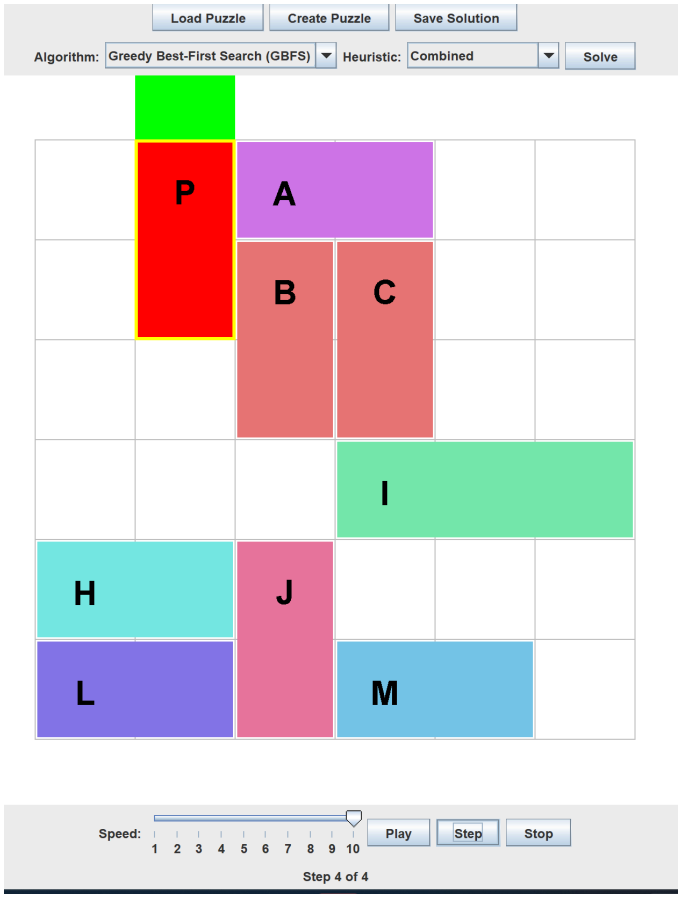
Input:

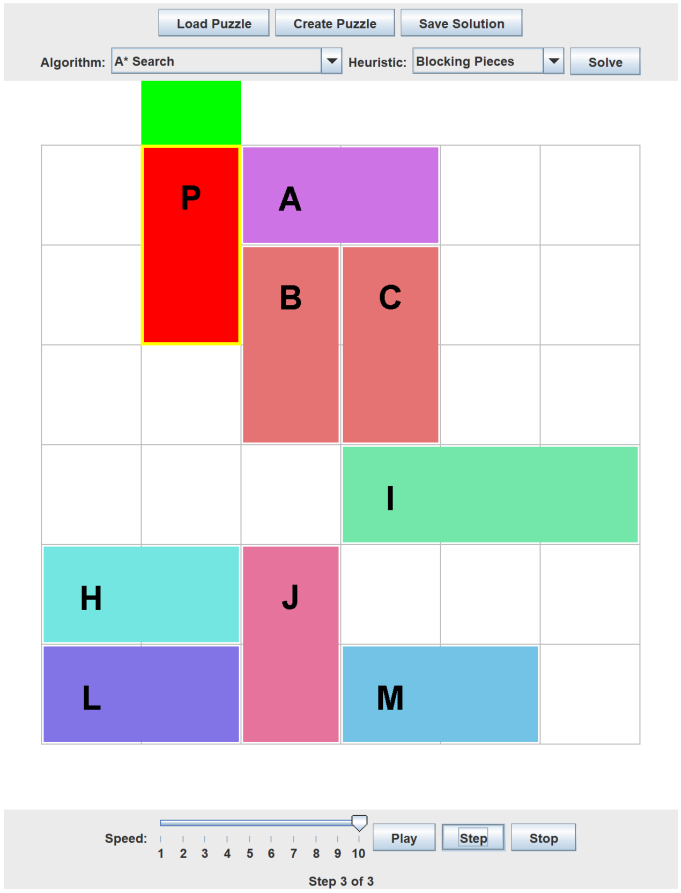
```
6 6
8
 K
AAB...
..BC..
.P.C..
.P.III
HHJ...
LLJMM.
```

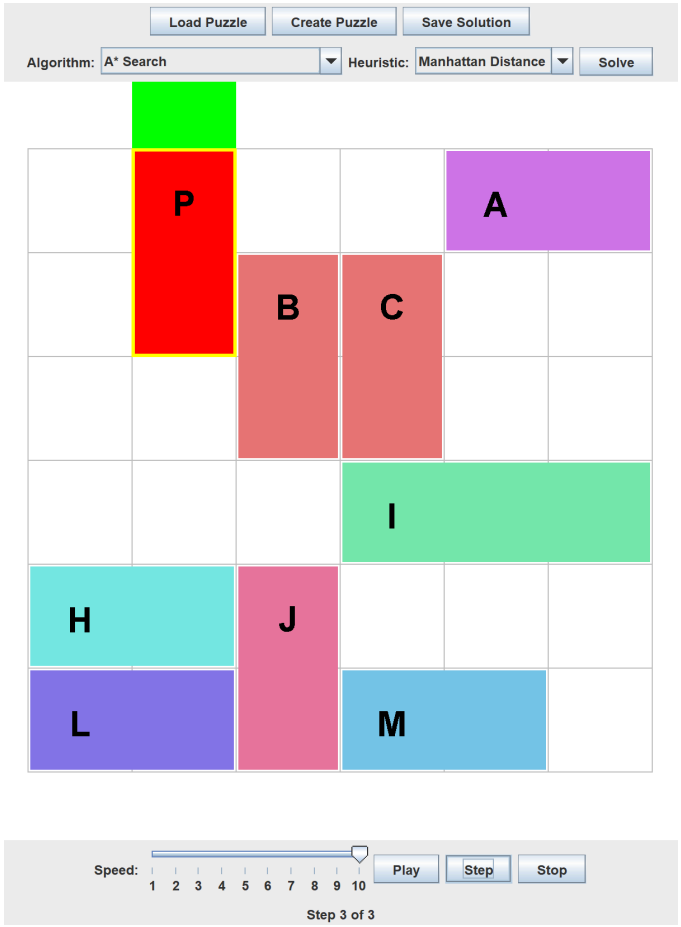
Algoritma (heuristik)	Output CLI	Output GUI
UCS		

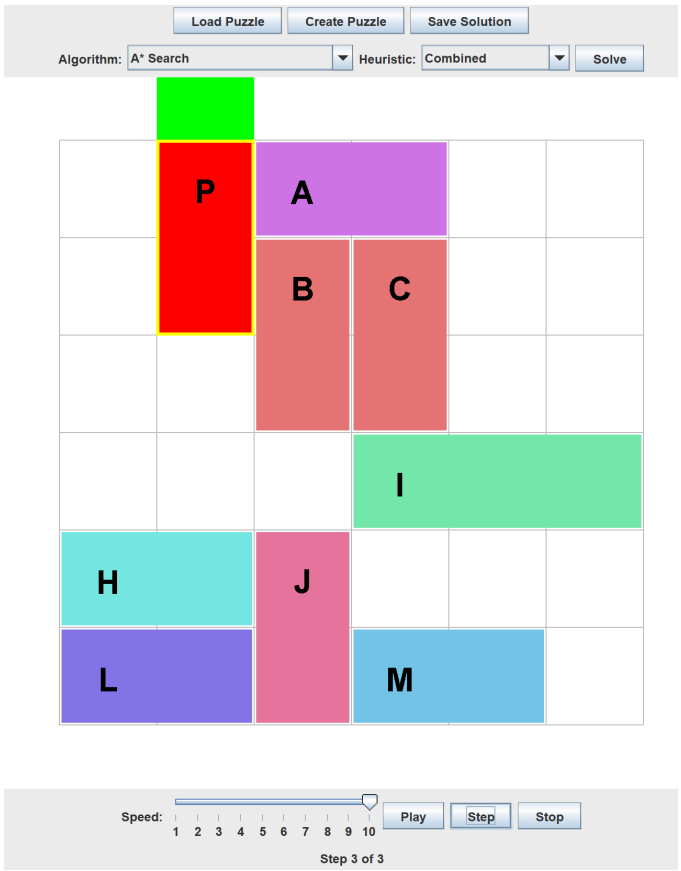
GBFS (blocking)	Algorithm: Greedy Best-First Search (GBFS) Heuristic: Blocking Pieces Total steps: 184 Nodes visited: 371 Execution time: 0.533 seconds Initial Board: AAB... ..BC.. .P.C.. .P.III HHJ... LLJMM. Step 1: Move piece M 1 cell(s) kanan AAB... ..BC.. .P.C.. .P.III HHJ... LLJ.MM Step 2: Move piece J 1 cell(s) atas AAB... ..BC.. .P.C.. .PJIII HHJ... LL..MM (Pakai di txt karena tidak bisa ss yang awal) Move 182: L kanan +-K----+ |AA....| |..B...| |.PB...| |.PJIII| |..JCHH| |.LLCMM| +------+ Move 183: A kanan 4 cells +-K----+ |....AA| |..B...| |.PB...| |.PJIII| |..JCHH| |.LLCMM| +------+ Move 184: P atas 2 cells +-K----+ |.P..AA| |.PB...| |..B...| |..JIII| |..JCHH| |.LLCMM| +------+ Jumlah langkah: 184 Jumlah node yang diperiksa: 371 Waktu eksekusi: 0.013 detik	

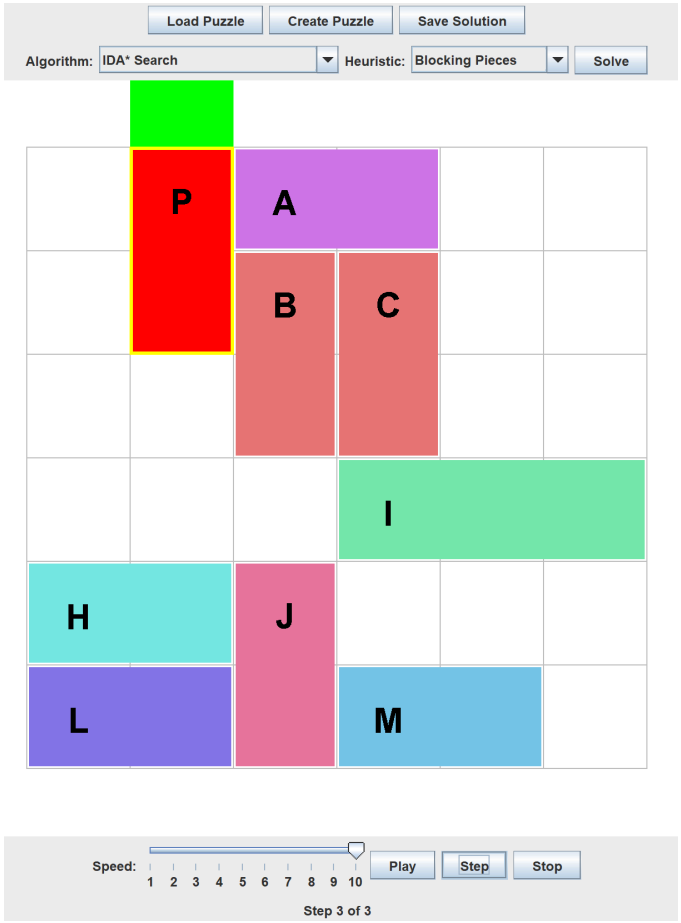
GBFS (Manhattan Distance)	Using heuristic: Manhattan Distance Initial Board: +-K----+ |AAB...| |..BC..| |.P.C..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 1: P atas +-K----+ |AAB...| |.PBC..| |.P.C..| |...III| |HHJ...| |LLJMM.| +------+ Move 2: B bawah +-K----+ |AA....| |.PBC..| |.PBC..| |...III| |HHJ...| |LLJMM.| +------+ Move 3: A kanan 2 cells +-K----+ |..AA..| |.PBC..| |.PBC..| |...III| |HHJ...| |LLJMM.| +------+ Move 4: P atas +-K----+ |.PAA..| |.PBC..| |..BC..| |...III| |HHJ...| |LLJMM.| +------+ Jumlah langkah: 4 Jumlah node yang diperiksa: 5 Waktu eksekusi: 0.004 detik	

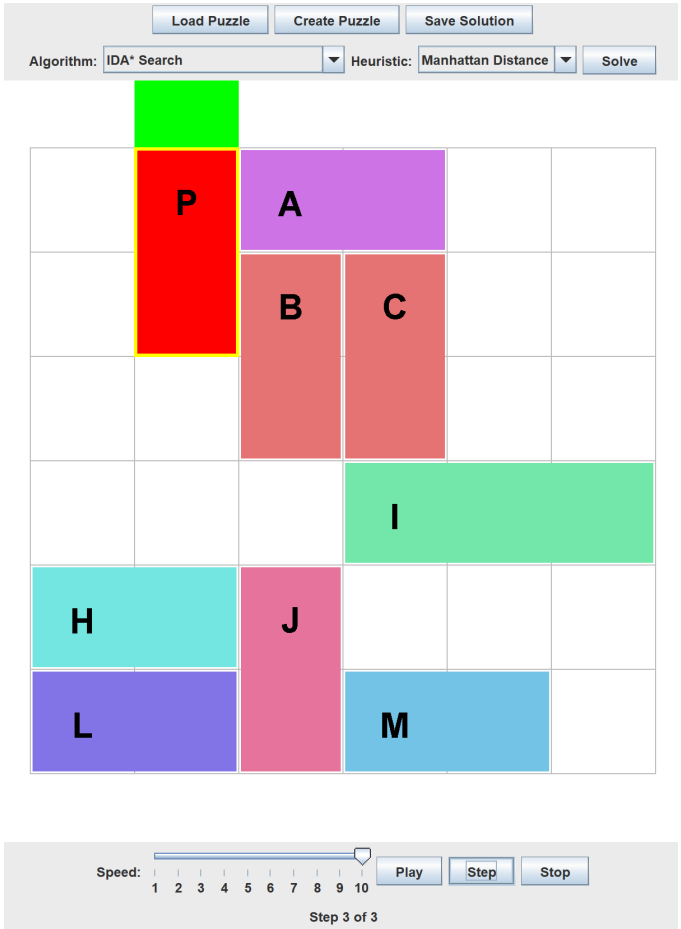
GBFS (Combined)	Using heuristic: Combined (Blocking + Manhattan) Initial Board: +-K----+ |AAB...| |..BC..| |.P.C..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 1: P atas +-K----+ |AAB...| |.PBC..| |.P.C..| |...III| |HHJ...| |LLJMM.| +------+ Move 2: B bawah +-K----+ |AA....| |.PBC..| |.PBC..| |...III| |HHJ...| |LLJMM.| +------+ Move 3: A kanan 2 cells +-K----+ |..AA..| |.PBC..| |.PBC..| |...III| |HHJ...| |LLJMM.| +------+ Move 4: P atas +-K----+ |.PAA..| |.PBC..| |..BC..| |...III| |HHJ...| |LLJMM.| +------+ Jumlah langkah: 4 Jumlah node yang diperiksa: 5 Waktu eksekusi: 0.003 detik	

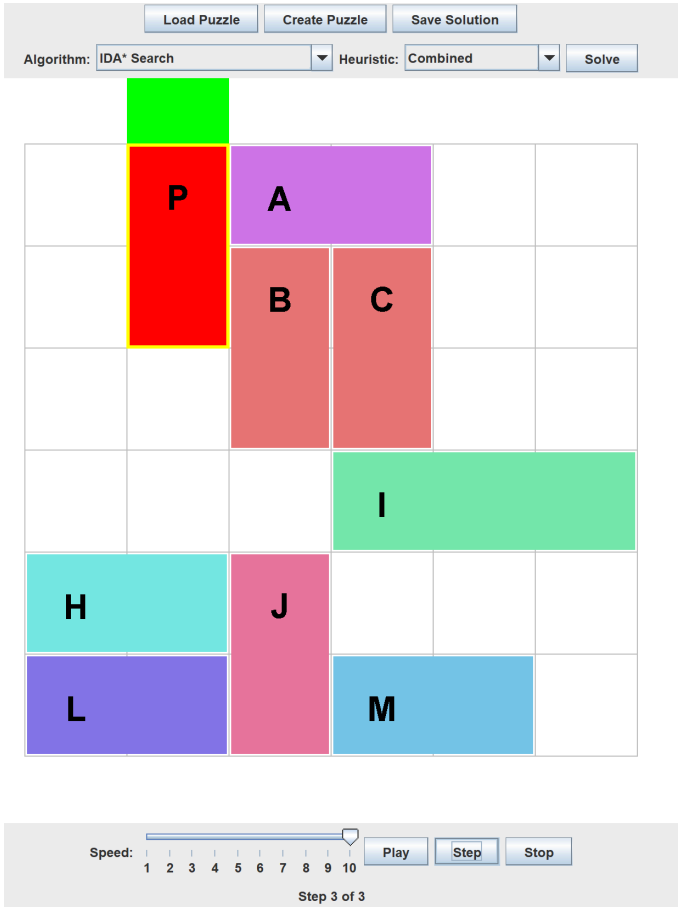
A* (Blocking)	Initial Board: +-K----+ |AAB...| |..BC..| |.P.C..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 1: B bawah +-K----+ |AA....| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 2: A kanan 2 cells +-K----+ |..AA..| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 3: P atas 2 cells +-K----+ |.PAA..| |.PBC..| |..BC..| |...III| |HHJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 57 Waktu eksekusi: 0.04 detik	

A* (Manhattan)	Using heuristic: Manhattan Distance Initial Board: +-K----+ |AAB...| |..BC..| |.P.C..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 1: B bawah +-K----+ |AA....| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 2: A kanan 4 cells +-K----+ |....AA| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 3: P atas 2 cells +-K----+ |.P..AA| |.PBC..| |..BC..| |...III| |HHJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 32 Waktu eksekusi: 0.007 detik	

A* (Combined)	Using heuristic: Combined (Blocking + Manhattan) Initial Board: +-K----+ |AAB...| |..BC..| |.P.C..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 1: B bawah +-K----+ |AA....| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 2: A kanan 2 cells +-K----+ |..AA..| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 3: P atas 2 cells +-K----+ |.PAA..| |.PBC..| |..BC..| |...III| |HHJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 7 Waktu eksekusi: 0.003 detik	

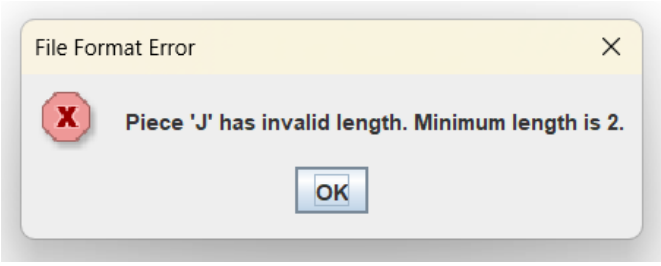
IDA* (Blocking)	Initial Board: +-K----+ |AAB...| |..BC..| |.P.C..| |.P.III| |HHJ...| |LLJMM.| +------+ Increasing threshold to: 2 Increasing threshold to: 3 Move 1: B bawah +-K----+ |AA....| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 2: A kanan 2 cells +-K----+ |..AA..| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 3: P atas 2 cells +-K----+ |.PAA..| |.PBC..| |..BC..| |...III| |HHJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 205 Waktu eksekusi: 0.011 detik	

IDA* (Manhattan)	Initial Board: +-K----+ |AAB...| |..BC..| |.P.C..| |.P.III| |HHJ...| |LLJMM.| +------+ Increasing threshold to: 3 Increasing threshold to: 4 Move 1: B bawah +-K----+ |AA....| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 2: A kanan 2 cells +-K----+ |..AA..| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 3: P atas 2 cells +-K----+ |.PAA..| |.PBC..| |..BC..| |...III| |HHJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 252 Waktu eksekusi: 0.008 detik	

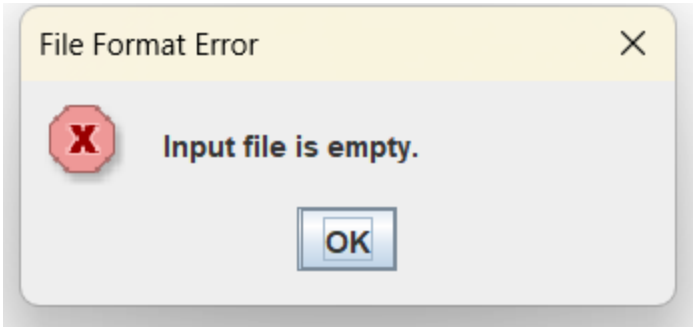
IDA* (Combined)	Initial Board: +-K----+ |AAB...| |..BC..| |.P.C..| |.P.III| |HHJ...| |LLJMM.| +------+ Increasing threshold to: 4 Move 1: B bawah +-K----+ |AA....| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 2: A kanan 2 cells +-K----+ |..AA..| |..BC..| |.PBC..| |.P.III| |HHJ...| |LLJMM.| +------+ Move 3: P atas 2 cells +-K----+ |.PAA..| |.PBC..| |..BC..| |...III| |HHJ...| |LLJMM.| +------+ Jumlah langkah: 3 Jumlah node yang diperiksa: 27 Waktu eksekusi: 0.003 detik	

3.6 Kasus 6 (Error Handling)

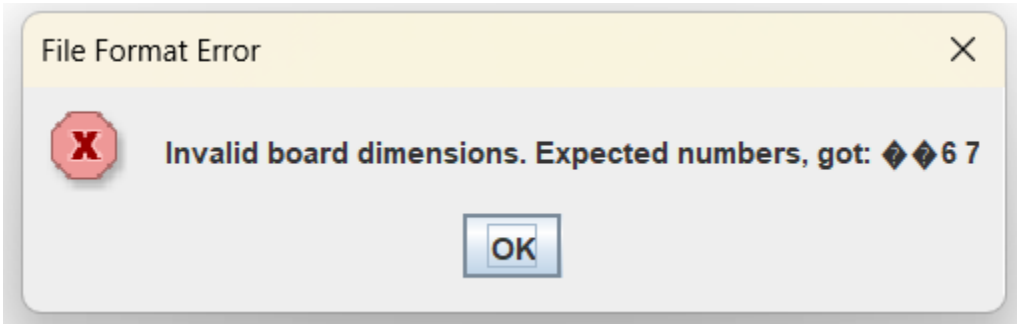
Error terdapat mobil yang panjangnya 1:



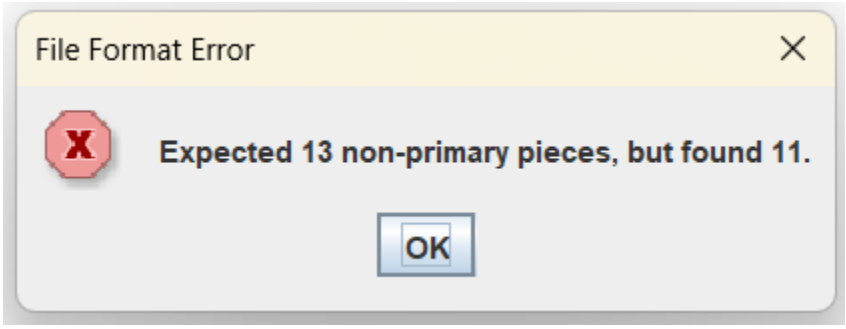
File kosong:



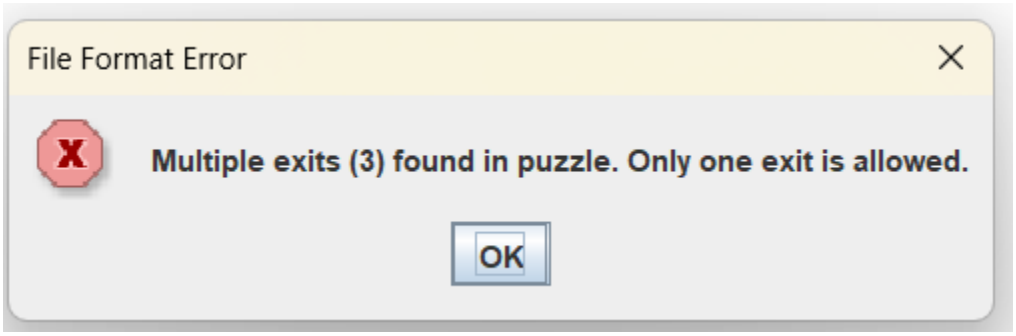
Dimensi tidak sesuai:



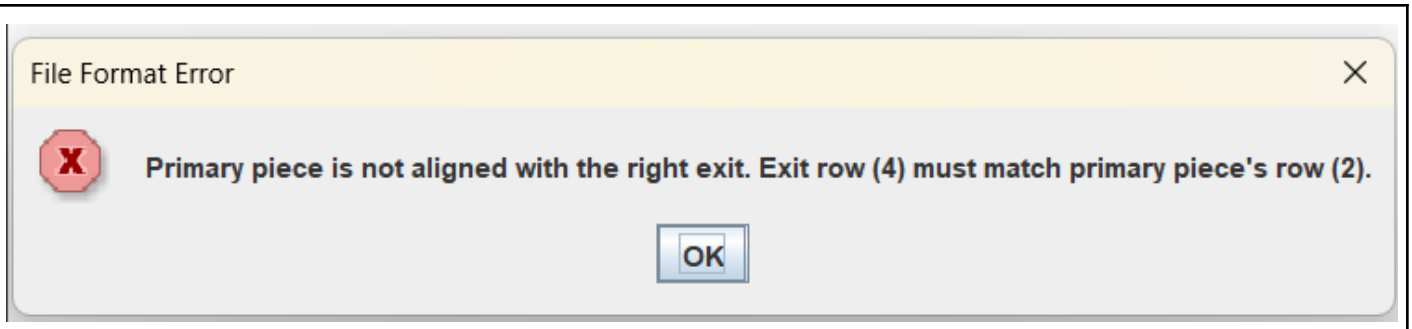
Jumlah piece tidak sesuai:



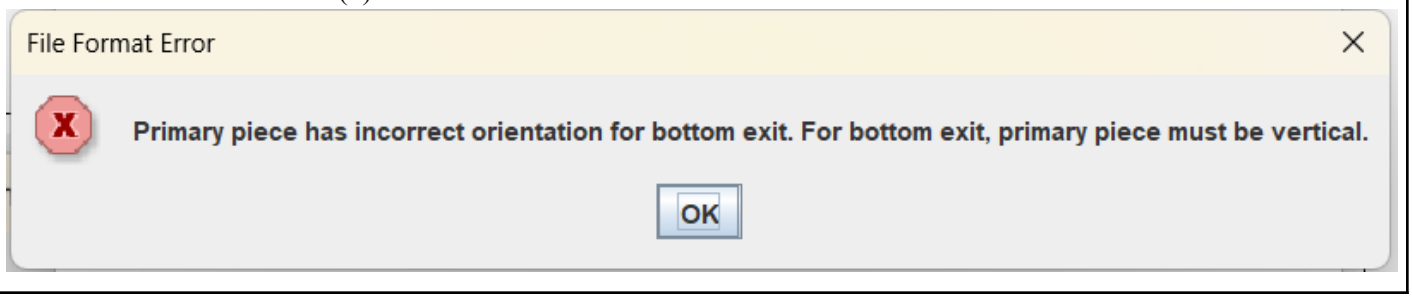
Multiple exit:



Letak exit tidak sesuai:



Letak exit tidak tidak sesuai (2):

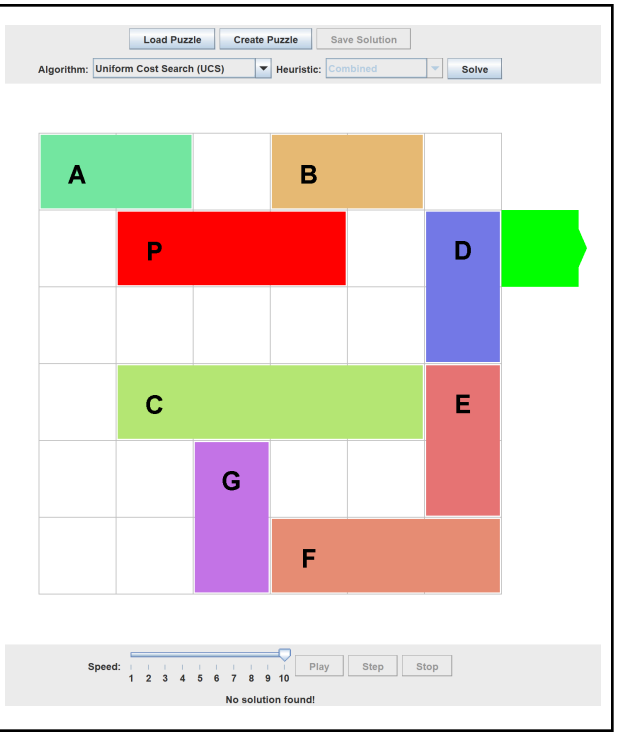


3.7 Kasus 7 (No Solution)

Input:

```
6 6
7
AA.BB.
.PPP.DK
.....D
.CCCCE
..G..E
..GFFF
```

Hasil:



BAB 4
HASIL DAN ANALISIS

Hasil eksperimen pada Bab 3 menunjukkan bahwa setiap algoritma bekerja dengan seharusnya. Program dapat berjalan dengan baik untuk beberapa test case, seperti pintu keluar yang berada di sebelah kanan, kiri, bawah, maupun atas. Setiap jenis algoritma yang diimplementasikan juga berjalan dengan baik, dimulai dari UCS, GBFS, A* hingga IDA*, sesuai dengan harapan sebelum implementasi.

Pada setiap pengujian (kasus 1-5), dapat dilihat bahwa solusi yang ditemukan UCS selalu optimal, yaitu banyak gerakan yang paling sedikit dibandingkan algoritma lainnya. Ini sesuai dengan apa yang diharapkan. Akan tetapi, kekurangannya adalah nodes visited cenderung paling banyak dibandingkan algoritma yang lain. Hal ini karena UCS tidak menggunakan heuristik apapun, mengeksplorrasi berdasarkan path cost (jumlah moves yang dibuat). Pada program yang dibuat ini, kompleksitas algoritma dari UCS adalah $O(b^d)$, dengan *b* adalah *branching factor* dan *d* adalah *depth of shortest solution*.

Jika melihat pada hasil pengujian GBFS di kelima kasus, hasil yang didapatkan dari GBFS dapat berbeda-beda jumlah langkah dan nodes visitednya. Akan tetapi, secara keseluruhan, dapat dilihat bahwa GBFS lebih cepat dalam menemukan solusi dibandingkan UCS dengan nodes visited yang lebih rendah juga. Akan tetapi, kekurangannya adalah solusi yang didapatkan tidak menjamin optimal, bahkan terkadang dapat jauh lebih besar dibandingkan solusi optimal seperti pada contoh kasus 1 dan kasus 2 dengan GBFS heuristik bloking, dimana seharusnya solusi optimal dapat dicapai dengan 5 dan 51 langkah, tetapi GBFS menemukan solusi dengan 64 dan 304 langkah. Walaupun memang node visited dan waktu pencariannya sangat cepat, tetapi mengorbankan keoptimalan solusi. Dalam program ini. GBFS memiliki kompleksitas algoritma $O(b^d)$.

Meninjau dari algoritma A* (A-Star), dari pengujian yang telah dilakukan dapat dilihat bahwa algoritma yang paling "baik" adalah algoritma ini. Hal ini karena A* memungkinkan untuk mendapatkan solusi optimal dalam waktu yang singkat juga. Pada contoh kasus 1, algoritma A* dapat menghasilkan solusi optimal (5 langkah), dalam waktu yang jauh lebih singkat dibandingkan UCS, dengan nodes visited yang lebih rendah dibandingkan GBFS juga. Dalam kasus kompleks juga A* dapat menghasilkan solusi optimal dengan nodes visited lebih sedikit serta waktu yang lebih singkat, walaupun memang belum sesingkat GBFS. Kompleksitas algoritma untuk A* juga $O(b^d)$.

Akan tetapi, perlu diingat bahwa algoritma A* dalam program ini tidak selalu menemukan solusi optimal dalam kasus-kasus simpel, tergantung heuristik yang digunakan apakah *admissible* atau tidak, bisa juga karena hal teknis lainnya. Seperti contoh, pada kasus simpel seperti kasus 1 dan kasus 3, dengan penggunaan algoritma A* menggunakan heuristik Manhattan mendapatkan 6 langkah, dibandingkan solusi optimal yang seharusnya 5 langkah. Setelah dilihat prosesnya, perbedaan satu langkah ini disebabkan karena fitur "beberapa langkah *straight* menjadi satu langkah" dalam program ini. Jika fitur ini dihilangkan, dimana gerakan 2 blok lurus dihitung sebagai 2 langkah (bukan 1 langkah), maka total langkah yang didapatkan tetap sama, yang menunjukkan sebetulnya sudah optimal.

Sebagai bonus, dalam program ini terdapat algoritma IDA*. Setelah pengujian, dapat dilihat bahwa IDA* mampu untuk mendapatkan solusi optimal seperti A*, tetapi nodes visitednya jauh lebih banyak, menunjukkan eksplorasi dilakukan lebih banyak. Meskipun demikian, dapat dilihat dari kasus 1

dan kasus 3 bahwa waktu eksekusinya tidak lebih lama dibandingkan A*, menunjukkan node processing yang efisien, tetapi banyak revisitnya. Akan tetapi untuk kasus yang sangat kompleks seperti kasus 2, IDA* sangat lama prosesnya, terlalu banyak revisit sehingga dapat dikatakan tidak cocok untuk menemukan solusi. Algoritma tambahan IDA* ini memiliki kompleksitas algoritma $O(b^d)$.

Selain itu, metode heuristik yang dipilih juga mempengaruhi bagaimana algoritma GBFS , A* dan IDA* ini bekerja. Dalam program, terdapat 3 metode heuristik tersedia yang dapat digunakan, yaitu mobil yang menghalangi, Manhattan distance, serta gabungan dari kedua heuristik tersebut. Setelah pengujian, dalam GBFS, kombinasi > Manhattan > *blocking*. Untuk A* heuristik kombinasi dinyatakan paling baik untuk kualitas solusi dan efisiensi node yang dikunjungi, begitu pula dengan IDA*.

BAB 5
PENJELASAN ALGORITMA BONUS

5.1 Implementasi Algoritma Alternatif

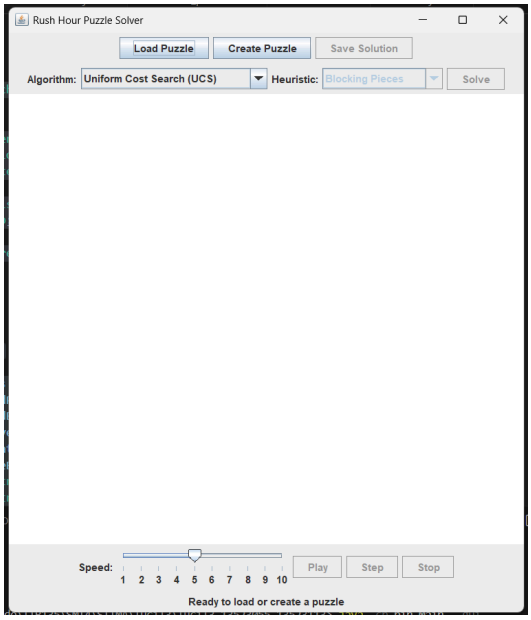
Algoritma alternatif yang kita buat adalah IDA* (Iterative Deepening A*), yaitu perpaduan antara Depth-First-Search dengan A*. Untuk penjelasan selengkapnya terkait algoritma alternatif ini, dapat dilihat pada Bab 1, serta analisis terkaitnya pada Bab 4.

5.2 Analisis Heuristik Alternatif

Heuristik yang dibuat dalam program ini ada tiga, yaitu *blocked cars* (menghitung jumlah mobil yang menghalangi mobil utama ke pintu keluar), Manhattan distance (menghitung jarak garis lurus dari mobil utama ke pintu keluar), serta kombinasi dari kedua heuristik, yaitu menghitung jumlah mobil yang menghalangi dan jarak mobil utama ke pintu keluar. Penjelasan lebih lengkapnya dapat dilihat pada Bab 1.5.2, dan analisisnya pada Bab 4.

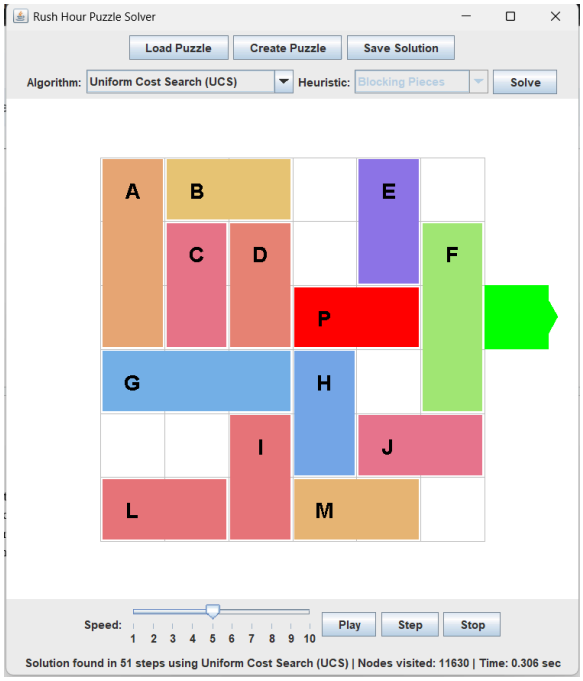
5.3 Implementasi GUI

Dalam program ini terdapat bonus GUI, yaitu *interface* untuk memudahkan penggunaan program Rush Hour Solver ini. GUI dibuat menggunakan Java Swing, memungkinkan pengguna untuk mendapatkan pengalaman interaktif untuk *solve* permainan Rush Hour ini. Pada GUI ini, pengguna dapat memuat file .txt yang menyediakan konfigurasi permainan, kemudian memilih jenis algoritma untuk menyelesaikan puzzlenya. Jika pengguna memilih GBFS, A* atau IDA*, maka terdapat pilihan tambahan untuk heuristik yang akan digunakan.

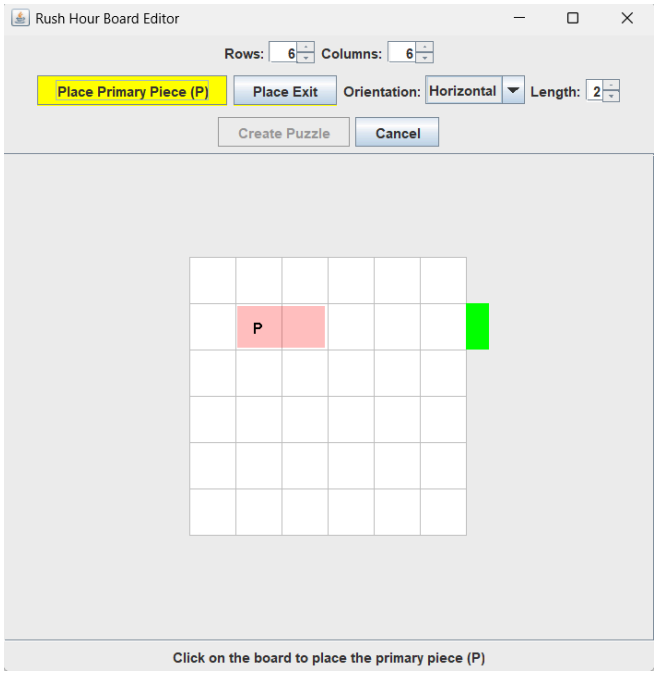


Gambar 5.1. Tampilan Utama GUI

Setelah pengguna menekan tombol solve, maka program akan berjalan dan menemukan solusi sesuai algoritma dan heuristik yang dipilih. Pengguna dapat menekan tombol play untuk visualisasi langkah-langkah mobil mencari jalan keluar yang kecepatannya dapat diatur menggunakan slider, atau memilih melihat step by step dengan tombol “step”. Selain itu, pengguna juga dapat membuat puzzle sendiri, memilih berapa dimensi papan, meletakkan mobil-mobil penghalang, mobil utama serta pintu keluar dan kemudian disave ke dalam sebuah file .txt.



Gambar 5.2. Tampilan GUI setelah solve



Gambar 5.3. Tampilan Pembuatan Puzzle

BAB 6
KESIMPULAN

Permainan puzzle Rush Hour dapat diselesaikan menggunakan berbagai implementasi algoritma *pathfinding*, yaitu UCS (Uniform Cost Search), GBFS (Greedy Best First Search), A* Search, dan bonus algoritma IDA* Search (Iterative Deepening A*). Keempat algoritma diterapkan untuk memindahkan mobil lain yang menghalangi jalan keluar mobil utama, agar mobil utama dapat keluar dari papan permainan pada pintu keluar yang telah disediakan dengan cara yang paling efisien.

Berdasarkan hasil implementasi dan pengujian, algoritma UCS selalu menghasilkan solusi optimal dalam jumlah langkah, namun mengeksplorasi lebih banyak node karena tidak menggunakan heuristik apapun. Di sisi lain, GBFS mendahulukan kecepatan karena hanya mempertimbangkan seberapa dekat node dengan goal, tetapi algoritma ini tidak menjamin solusi optimal. Algoritma A* menggabungkan kelebihan UCS dan GBFS, dengan heuristik yang *admissible* dapat memberikan solusi optimal dengan jumlah node yang diperluas lebih sedikit dari UCS. Untuk algoritma bonus, IDA* memberikan alternatif yang hemat memori karena menggunakan pendekatan *depth-first iterative deepening* dengan evaluasi seperti A*. Seperti A*, IDA* juga menjamin solusi optimal meskipun terkadang memerlukan lebih banya kiterasi dibandingkan A*.

Pada implementasi ini, digunakan tiga heuristik, yaitu jumlah mobil yang menghalangi jalur utama, jarak Manhattan dari mobil utama ke pintu keluar, serta penggabungan keduanya sebagai satu heuristik. Analisis berdasarkan eksperimen membuktikan bahwa A* adalah algoritma paling efisien secara keseluruhan.

LAMPIRAN

Pranala ke *repository*: https://github.com/tkanigara/Tucil3_13523055_13523113

Tabel keterselesaian:

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