LAPORAN TUGAS KECIL III IF2211 STRATEGI ALGORITMA

SEMESTER II TAHUN 2024/2025

Oleh:

13523053 - Sakti Bimasena

13523118 - Farrel Athalla Putra

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

JL. GANESA 10, BANDUNG 40132

2025

DAFTAR ISI

DAFTAR ISI **2**

BAB I DESKRIPSI MASALAH DAN ALGORITMA **4**

1.1 Algoritma Uniform Cost Search (UCS) 4

1.2 Algoritma Greedy Best First Search (GBFS) 5

1.3 Algoritma A-Star (A*) 6

1.4 Algoritma Iterative Deepening A-Star (IDA*) 7

1.5 Penyelesaian Rush Hour dengan Algoritma Pathfinding 8

BAB II ALGORITMA PROGRAM **9**

BAB III ANALISIS ALGORITMA **36**

BAB IV SOURCE CODE PROGRAM **38**

4.1 AStar.java 38

4.2 GameAnimation.java 44

4.3 GamePanel.java 50

4.4 GBFS.java 59

4.5 IDAStar.java 64

4.6 RushHourGame.java 71

4.7 RushHourGUI.java 83

4.8 RushHourIO.java 94

4.9 RushHourSolver.java 103

4.10 SearchAlgoritm.java 111

4.11 UCS.java 115

BAB V MASUKAN DAN LUARAN PROGRAM **119**

5.1 Isi test1.txt 119

5.2 Isi test2.txt 121

5.3 Isi test3.txt 123

5.4 Isi test4.txt 126

5.5 Isi test5.txt 128

5.6 Isi test6.txt 129

5.7 Isi test7.txt 129

5.8 Isi test8.txt 129

5.9 Isi test9.txt 129

BAB VI ANALISIS IMPLEMENTASI **130**

BAB VII IMPLEMENTASI BONUS **132**

8.1 Implementasi Algoritma Pathfinding Alternatif (IDA*) 132

8.2 Implementasi Heuristik Alternatif 132

8.3 Implementasi Graphical User Interface (GUI) 132

BAB VIII KESIMPULAN **134**

BAB IX LAMPIRAN **135**

BAB I
DESKRIPSI MASALAH DAN ALGORITMA

1.1 Algoritma Uniform Cost Search (UCS)

Uniform Cost Search adalah sebuah algoritma pencarian yang menggunakan prinsip eksplorasi berdasarkan biaya terendah untuk menemukan jalur terpendek dari keadaan awal ke keadaan tujuan. Proses utama UCS melibatkan ekspansi simpul dengan biaya kumulatif terendah pada setiap iterasi. Algoritma ini menggunakan struktur data priority queue untuk menyimpan simpul-simpul yang akan dieksplorasi, dengan prioritas berdasarkan biaya kumulatif dari simpul awal hingga simpul tersebut. UCS menjamin menemukan jalur dengan biaya terendah jika setiap langkah memiliki biaya non-negatif.

Dalam konteks permasalahan Rush Hour Puzzle, UCS akan mengeksplorasi konfigurasi-konfigurasi papan permainan dengan mengutamakan konfigurasi yang dicapai dengan jumlah pergerakan (biaya) paling sedikit. Algoritma ini mempertimbangkan semua kemungkinan pergerakan mobil dan memprioritaskan jalur yang membutuhkan langkah minimal. Keunggulan UCS terletak pada kemampuannya menghasilkan solusi optimal, yaitu jalur dengan jumlah langkah paling sedikit untuk memindahkan mobil merah ke pintu keluar.

Keunggulan UCS dibandingkan algoritma pencarian lainnya seperti Breadth-First Search (BFS) adalah kemampuannya menangani kasus dimana setiap langkah memiliki biaya yang berbeda-beda. Dalam konteks Rush Hour, jika kita memberikan bobot berbeda untuk pergerakan mobil yang berbeda, UCS akan tetap menemukan jalur optimal. Namun, ketika semua langkah memiliki biaya yang sama (seperti dalam Rush Hour klasik dimana setiap pergerakan mobil dihitung sebagai satu langkah), UCS akan berperilaku seperti BFS. Kompleksitas waktu UCS bergantung pada jumlah simpul yang dikunjungi, yang dalam kasus terburuk adalah O(b^d), dimana b adalah faktor percabangan dan d adalah kedalaman solusi. Sementara kompleksitas ruang juga O(b^d) karena algoritma harus menyimpan semua simpul yang dieksplorasi.

Meski UCS menjamin solusi optimal, kelemahannya pada Rush Hour Puzzle adalah potensi eksplorasi ruang pencarian yang besar jika solusi berada jauh dari posisi awal. Karena algoritma tidak menggunakan informasi heuristik mengenai seberapa dekat suatu konfigurasi dengan tujuan, UCS dapat mengeksplorasi banyak jalur yang sebenarnya menjauhi solusi sebelum akhirnya menemukan jalur optimal.

1.2 Algoritma Greedy Best First Search (GBFS)

Greedy Best-First Search adalah algoritma pencarian informasi yang menggunakan fungsi heuristik untuk mengestimasi jarak dari setiap simpul ke simpul tujuan. Berbeda dengan UCS yang mengutamakan biaya kumulatif perjalanan dari simpul awal, GBFS sepenuhnya bergantung pada nilai heuristik untuk memilih simpul berikutnya yang akan dieksplorasi. Algoritma ini menggunakan priority queue yang memprioritaskan simpul dengan nilai heuristik terendah, yang mengindikasikan perkiraan jarak terdekat ke tujuan.

Dalam konteks Rush Hour Puzzle, GBFS akan memilih konfigurasi papan permainan yang dianggap paling menjanjikan berdasarkan fungsi heuristik. Contoh heuristik yang dapat digunakan adalah jumlah mobil yang menghalangi jalur mobil merah ke pintu keluar, atau jarak mobil merah dari pintu keluar. Algoritma ini akan selalu memprioritaskan ekspansi konfigurasi yang memiliki nilai heuristik terbaik, dengan harapan dapat menemukan solusi lebih cepat.

Keunggulan utama GBFS adalah efisiensinya dalam mengeksplorasi ruang pencarian. Dengan fokus pada simpul yang dianggap paling dekat dengan tujuan, GBFS dapat menemukan solusi dengan eksplorasi yang jauh lebih sedikit dibandingkan UCS atau BFS, terutama pada permasalahan dengan ruang pencarian yang luas seperti Rush Hour Puzzle. Kompleksitas waktu dan ruang GBFS dalam kasus terbaik dapat mencapai O(bd), dimana b adalah faktor percabangan dan d adalah kedalaman solusi, namun dalam kasus terburuk tetap O(b^d) jika algoritma harus mengeksplorasi hampir seluruh ruang pencarian.

Meskipun GBFS dapat menemukan solusi dengan cepat, kelemahan utamanya adalah tidak adanya jaminan optimalitas. Karena algoritma ini hanya mempertimbangkan perkiraan jarak ke tujuan tanpa memperhitungkan biaya perjalanan dari simpul awal, GBFS dapat terjebak dalam jalur yang tampak menjanjikan secara heuristik tetapi sebenarnya tidak optimal. Misalnya, dalam Rush Hour Puzzle, GBFS mungkin memilih konfigurasi dimana mobil merah lebih dekat ke pintu keluar, meskipun mencapai konfigurasi tersebut membutuhkan langkah yang jauh lebih banyak dibandingkan jalur optimal.

Selain itu, kualitas solusi GBFS sangat bergantung pada fungsi heuristik yang digunakan. Heuristik yang tidak akurat dapat menyebabkan algoritma terperangkap dalam jalur suboptimal atau bahkan gagal menemukan solusi. Dalam konteks Rush Hour Puzzle, merancang heuristik yang tepat merupakan tantangan tersendiri karena kompleksitas interaksi antar mobil pada papan permainan. Heuristik sederhana seperti jarak mobil merah ke pintu keluar mungkin tidak cukup mencerminkan situasi sebenarnya dimana mobil lain dapat menghalangi jalan.

1.3 Algoritma A-Star (A*)

A-Star adalah algoritma pencarian terinformasi yang menggabungkan kelebihan Uniform Cost Search (UCS) dan Greedy Best-First Search (GBFS). Algoritma ini menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, dimana $g(n)$ adalah biaya sebenarnya dari simpul awal ke simpul n (seperti pada UCS), dan $h(n)$ adalah nilai heuristik yang mengestimasi biaya dari simpul n ke simpul tujuan (seperti pada GBFS). A* menggunakan priority queue untuk menyimpan simpul-simpul yang akan dieksplorasi, dengan prioritas berdasarkan nilai $f(n)$ terendah.

Dalam konteks Rush Hour Puzzle, A* akan memperhitungkan baik jumlah langkah yang telah dilakukan ($g(n)$) maupun estimasi langkah yang diperlukan untuk mencapai solusi ($h(n)$). Fungsi $g(n)$ merepresentasikan jumlah pergerakan mobil yang telah dilakukan untuk mencapai konfigurasi papan saat ini, sementara $h(n)$ dapat berupa estimasi jumlah langkah tambahan yang diperlukan, misalnya berdasarkan jumlah mobil yang menghalangi jalan keluar dan kompleksitas posisinya. Dengan mempertimbangkan kedua faktor ini, A* dapat menemukan jalur yang efisien sekaligus optimal.

Keunggulan utama A* adalah jaminan optimalitasnya dengan syarat fungsi heuristik yang digunakan bersifat admissible (tidak pernah melebih-lebihkan estimasi biaya ke tujuan) dan consistent (memenuhi ketidaksamaan segitiga). Jika kedua syarat tersebut terpenuhi, A* dijamin menemukan jalur dengan biaya terendah ke tujuan. Selain itu, A* juga lebih efisien dibandingkan UCS karena menggunakan informasi heuristik untuk mengarahkan pencarian, sehingga dapat mengurangi jumlah simpul yang perlu dieksplorasi.

Dalam penerapannya pada Rush Hour Puzzle, A* dengan heuristik yang baik akan mengeksplorasi jauh lebih sedikit konfigurasi papan dibandingkan UCS, sekaligus tetap menjamin solusi dengan jumlah langkah minimal. Heuristik yang umum digunakan untuk permasalahan ini mencakup jumlah mobil yang menghalangi jalur mobil merah, jarak Manhattan mobil merah ke pintu keluar, atau kombinasi keduanya. Kompleksitas waktu dan ruang A* dalam kasus terburuk tetap $O(b^d)$, namun dalam praktiknya sering kali jauh lebih baik tergantung pada kualitas heuristik yang digunakan.

Meskipun A* memiliki banyak kelebihan, algoritma ini juga memiliki beberapa kelemahan. Kebutuhan memori A* dapat menjadi masalah serius pada permasalahan dengan ruang pencarian yang sangat besar, karena algoritma harus menyimpan semua simpul yang telah dieksplorasi. Dalam Rush Hour Puzzle dengan papan berukuran besar atau banyak mobil, jumlah konfigurasi yang mungkin dapat sangat besar, menyebabkan kebutuhan memori yang tinggi. Selain itu, kinerja A* sangat bergantung pada kualitas fungsi heuristik; heuristik yang buruk dapat mengakibatkan kinerja yang mendekati UCS atau bahkan lebih buruk.

1.4 Algoritma Iterative Deepening A-Star (IDA*)

Iterative Deepening A-Star (IDA*) adalah algoritma pencarian terinformasi yang menggabungkan prinsip Iterative Deepening Depth-First Search (IDDFS) dengan fungsi evaluasi A*. Algoritma ini dirancang untuk mengatasi kelemahan utama A* berupa kebutuhan memori yang besar. IDA* melakukan serangkaian pencarian depth-first dengan batas yang semakin meningkat, dimana batas ini didasarkan pada nilai fungsi f(n) = g(n) + h(n) seperti pada A*. Pada setiap iterasi, IDA* melakukan pencarian depth-first dengan memangkas cabang-cabang yang memiliki nilai f(n) melebihi batas yang ditentukan untuk iterasi tersebut.

Dalam konteks Rush Hour Puzzle, IDA* akan melakukan eksplorasi mendalam pada konfigurasi-konfigurasi papan permainan, namun membatasi eksplorasi berdasarkan estimasi total biaya (jumlah langkah yang telah dilakukan ditambah estimasi langkah yang tersisa). Batas awal ditetapkan sebesar nilai heuristik dari konfigurasi awal, dan pada setiap iterasi, batas dinaikkan ke nilai f terkecil yang melebihi batas sebelumnya. Pendekatan ini memungkinkan IDA* secara bertahap memperluas ruang pencariannya hingga menemukan solusi optimal.

Keunggulan utama IDA* adalah efisiensi memori yang sangat baik, dengan kompleksitas ruang hanya O(d) dimana d adalah kedalaman pencarian, karena algoritma ini hanya menyimpan simpul-simpul yang berada pada jalur eksplorasi saat ini. Hal ini membuat IDA* sangat cocok untuk permasalahan dengan ruang pencarian yang sangat besar seperti varian Rush Hour Puzzle yang kompleks. Seperti halnya A*, IDA* juga menjamin solusi optimal jika fungsi heuristik yang digunakan bersifat admissible dan consistent.

Selain itu, IDA* menggabungkan kelebihan pencarian depth-first yang efisien secara ruang dengan kemampuan A* dalam mengarahkan pencarian menggunakan informasi heuristik. Pada permasalahan Rush Hour Puzzle yang memiliki faktor percabangan bervariasi (jumlah pergerakan yang mungkin berbeda-beda pada setiap konfigurasi), IDA* dapat beradaptasi dengan baik, mengeksplorasi lebih dalam pada area yang menjanjikan dan memangkas cabang yang kurang potensial.

Namun, kelemahan utama IDA* adalah kemungkinan melakukan kalkulasi berulang. Karena tidak menyimpan simpul-simpul yang telah dieksplorasi pada iterasi sebelumnya, IDA* sering mengeksplorasi ulang simpul-simpul yang sama pada iterasi berbeda. Dalam Rush Hour Puzzle, dimana banyak konfigurasi papan berbeda dapat dicapai melalui urutan langkah yang berbeda, pengulangan ini dapat menjadi signifikan. Kompleksitas waktu IDA* dalam kasus terburuk adalah O(b^d), namun dengan overhead tambahan dari iterasi berulang.

1.5 Penyelesaian Rush Hour dengan Algoritma Pathfinding

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

Komponen penting dari permainan Rush Hour terdiri dari papan (board), piece (kendaraan), primary piece (kendaraan utama berwarna merah), pintu keluar, dan gerakan. Papan merupakan tempat permainan dimainkan, terdiri atas cell yang menjadi satuan dari papan. Piece adalah kendaraan di dalam papan yang memiliki posisi, ukuran (biasanya 2 atau 3 cell), dan orientasi (horizontal atau vertikal). Primary piece adalah mobil utama yang harus dikeluarkan dari papan untuk menyelesaikan permainan. Pintu keluar adalah tempat primary piece dapat digerakkan keluar. Gerakan yang dimaksud adalah pergeseran piece di dalam permainan, dimana piece hanya dapat bergerak sesuai orientasinya dan tidak dapat menembus piece lain.

Permainan ini dapat dipandang sebagai masalah pencarian jalur (pathfinding) dimana setiap konfigurasi papan merepresentasikan satu keadaan (state) dan setiap pergerakan piece merupakan transisi dari satu keadaan ke keadaan lain. Tujuan dari pathfinding adalah menemukan serangkaian gerakan yang akan membawa keadaan awal (konfigurasi awal papan) ke keadaan tujuan (primary piece mencapai pintu keluar).

Pendekatan penyelesaian Rush Hour dengan algoritma pathfinding melibatkan beberapa konsep kunci. Pertama, ruang keadaan (state space) yang terdiri dari semua kemungkinan konfigurasi papan. Kedua, fungsi penerus (successor function) yang menghasilkan keadaan-keadaan baru berdasarkan pergerakan yang mungkin dilakukan pada keadaan saat ini. Ketiga, kondisi tujuan (goal condition) yaitu kondisi dimana primary piece telah mencapai pintu keluar. Keempat, representasi gerakan (action representation) yang mendefinisikan bagaimana piece dapat digerakkan sesuai aturan permainan.

BAB II
ALGORITMA PROGRAM

Salah satu cara untuk menyelesaikan permainan ini adalah dengan menggunakan algoritma pathfinding. Berikut adalah langkah-langkah algoritma pathfinding yang digunakan dalam program penyelesaian Puzzle Rush Hour ini:

1. Membaca dan Memvalidasi Input

```
public static RushHourGame loadGameFromFile(String filename) throws
IOException {
    BufferedReader reader = new BufferedReader(new FileReader(filename));

    try {
        // Read dimensions
        String[] dimensions = reader.readLine().trim().split("\\s+");
        if (dimensions.length != 2) {
            throw new IOException("Invalid format: Baris dan kolom tidak
valid");
        }

        int rows = Integer.parseInt(dimensions[0]);
        int cols = Integer.parseInt(dimensions[1]);

        if (rows <= 0 || cols <= 0 || rows > 20 || cols > 20) {
            throw new IOException("Dimensi tidak valid");
        }

        // Read number of pieces (excluding primary piece)
        String numPiecesStr = reader.readLine().trim();
        int numPieces = Integer.parseInt(numPiecesStr);

        if (numPieces <= 0 || numPieces > 24) {
            throw new IOException("Invalid number of pieces:Jumlah piece
tidak valid");
        }

        // Create game instance
        RushHourGame game = new RushHourGame(rows, cols);
        // ... membaca konfigurasi papan ...

        // Validasi dan inisialisasi game state
        validateAndInitializeGame(game);

        return game;
    } finally {
        reader.close();
    }
}
```

Program melakukan validasi terhadap input untuk memastikan bahwa dimensi papan valid (di antara 1-20), jumlah piece valid (di antara 0-24), ada primary piece 'P' dan pintu keluar 'K', semua piece memiliki bentuk yang valid (horizontal atau vertikal), dan pintu keluar berada di dinding papan dan sejajar dengan orientasi primary piece.

2. **Representasi Keadaan (State Representation)**

```
public class RushHourGame {
    private int rows, cols;
    private char[][] board;
    private char primaryPiece = 'P';
    private char exit = 'K';
    protected int exitRow, exitCol;
    protected int primaryRow, primaryCol;
    private Map<Character, List<int[]>> pieces;

    // Constructor dan metode-metode lainnya

    // Copy constructor untuk menciptakan state baru
    public RushHourGame(RushHourGame other) {
        this.rows = other.rows;
        this.cols = other.cols;
        this.board = new char[rows][cols];
        this.pieces = new HashMap<>();

        // Deep copy board
        for (int i = 0; i < rows; i++) {
            System.arraycopy(other.board[i], 0, this.board[i], 0, cols);
        }

        // Deep copy pieces
        for (Map.Entry<Character, List<int[]>> entry :
other.pieces.entrySet()) {
            List<int[]> newPieces = new ArrayList<>();
            for (int[] piece : entry.getValue()) {
                newPieces.add(piece.clone());
            }
            this.pieces.put(entry.getKey(), newPieces);
        }

        this.exitRow = other.exitRow;
        this.exitCol = other.exitCol;
        this.primaryRow = other.primaryRow;
        this.primaryCol = other.primaryCol;
    }
}
```

Setiap konfigurasi papan permainan direpresentasikan sebagai satu state dalam ruang pencarian. Kelas RushHourGame menyimpan informasi tentang papan, posisi setiap piece, dan orientasi piece. Setiap state menyimpan: dimensi papan (rows dan cols),

representasi papan 2D (board), posisi primary piece dan pintu keluar, dan daftar semua piece dan posisinya (pieces).

3. **Membangkitkan State Berikutnya (Successor Function)**

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

    for (Map.Entry<Character, List<int[]>> entry : pieces.entrySet()) {
        char piece = entry.getKey();
        List<int[]> positions = entry.getValue();

        if (positions.size() < 2) continue;

        // Determine if piece is horizontal or vertical
        boolean isHorizontal = positions.get(0)[0] == positions.get(1)[0];

        if (isHorizontal) {
            // Try moving left and right
            tryMoveHorizontal(nextStates, piece, -1); // left
            tryMoveHorizontal(nextStates, piece, 1);  // right
        } else {
            // Try moving up and down
            tryMoveVertical(nextStates, piece, -1); // up
            tryMoveVertical(nextStates, piece, 1);  // down
        }
    }

    return nextStates;
}
```

Successor function membangkitkan semua state yang mungkin dicapai dari state saat ini dengan menggeser satu piece. Metode getNextStates() mengimplementasikan fungsi ini. Metode ini melakukn iterasi melalui semua piece pada papan, menentukan orientasi piece (horizontal atau vertikal), mencoba menggeser piece ke segala arah yang mungkin sesuai orientasinya, membuat state baru untuk setiap pergerakan yang valid, dan memastikan bahwa pergerakan tidak melanggar aturan (tidak melewati batas papan atau menembus piece lain)

4. **Mengecek Keadaan Tujuan**

```
public boolean isGoalState() {
    // Check if primary piece can exit
    List<int[]> primaryPositions = pieces.get('P');
    if (primaryPositions == null) return false;

    // Determine orientation of primary piece
    boolean isHorizontal = primaryPositions.size() > 1 &&
            primaryPositions.get(0)[0] == primaryPositions.get(1)[0];
```

```
    if (isHorizontal) {
        // Primary piece is horizontal, exit should be on the same row
        if (exitRow != primaryRow) return false;

        // Check if we can slide the primary piece to the exit
        int minCol = Integer.MAX_VALUE, maxCol = Integer.MIN_VALUE;
        for (int[] pos : primaryPositions) {
            minCol = Math.min(minCol, pos[1]);
            maxCol = Math.max(maxCol, pos[1]);
        }

        // Check if exit is reachable
        if (exitCol == cols) {
            // Exit is at right border
            if (maxCol == cols - 1) {
                // Primary piece is already at the edge, goal reached
                return true;
            } else {
                // Check if path to right edge is clear
                for (int j = maxCol + 1; j < cols; j++) {
                    if (board[exitRow][j] != '.') {
                        return false;
                    }
                }
                return true;
            }
        }
        // ... cases for other possible exit locations ...
    } else {
        // Similar check for vertical piece
        // ...
    }

    return false;
}
```

Fungsi isGoalState() mengecek apakah state saat ini adalah keadaan tujuan, yaitu primary piece dapat keluar melalui pintu keluar. Fungsi ini memeriksa apakah primary piece memiliki orientasi yang sejalan dengan pintu keluar, jalur dari primary piece ke pintu keluar tidak terhalang oleh piece lain, dan primary piece sudah mencapai pintu keluar atau dapat digeser langsung ke pintu keluar.

5. Fungsi Heuristik

Program mengimplementasikan dua fungsi heuristik yang dapat digunakan dalam algoritma pencarian terinformasi (GBFS, A*, dan IDA*). Heuristik 1 menghitung jarak Manhattan dari primary piece ke pintu keluar, sementara Heuristik 2 mempertimbangkan jumlah piece yang menghalangi jalur primary piece ke pintu keluar serta jarak.

Heuristik 1 : Manhattan Distance

```
// Calculate heuristic 1: Manhattan distance from primary piece to exit
public int getHeuristic1() {
    if (isGoalState()) return 0;

    List<int[]> primaryPositions = pieces.get('P');
    if (primaryPositions == null) return Integer.MAX_VALUE;

    // Determine if primary piece is horizontal or vertical
    boolean isHorizontal = primaryPositions.size() > 1 &&
            primaryPositions.get(0)[0] == primaryPositions.get(1)[0];

    if (isHorizontal) {
        // Primary piece is horizontal
        int minCol = Integer.MAX_VALUE, maxCol = Integer.MIN_VALUE;
        for (int[] pos : primaryPositions) {
            minCol = Math.min(minCol, pos[1]);
            maxCol = Math.max(maxCol, pos[1]);
        }

        // Calculate distance to exit
        if (exitRow == primaryRow) {
            if (exitCol == cols) {
                // Exit is at right border
                return cols - maxCol - 1;
            } else if (exitCol > maxCol) {
                return exitCol - maxCol;
            } else if (exitCol < minCol) {
                return minCol - exitCol;
            }
        }
        return Math.abs(exitRow - primaryRow) + Math.min(Math.abs(exitCol -
minCol), Math.abs(exitCol - maxCol));
    } else {
        // Primary piece is vertical
        int minRow = Integer.MAX_VALUE, maxRow = Integer.MIN_VALUE;
        for (int[] pos : primaryPositions) {
            minRow = Math.min(minRow, pos[0]);
            maxRow = Math.max(maxRow, pos[0]);
        }

        // Calculate distance to exit
        if (exitCol == primaryCol) {
            if (exitRow == rows) {
```

```
                // Exit is at bottom border
                return rows - maxRow - 1;
            } else if (exitRow > maxRow) {
                return exitRow - maxRow;
            } else if (exitRow < minRow) {
                return minRow - exitRow;
            }
        }
        return Math.abs(exitCol - primaryCol) + Math.min(Math.abs(exitRow - 
minRow), Math.abs(exitRow - maxRow));
    }
}
```

Heuristik pertama menghitung jarak Manhattan dari primary piece ke pintu keluar. Jarak Manhattan adalah jumlah langkah minimum yang diperlukan untuk mencapai tujuan dalam gerak horizontal dan vertikal, tanpa mempertimbangkan penghalang.
Langkah-langkahnya sebagai berikut :

a. Pengecekan Awal
   - Jika state sudah merupakan goal state, heuristik bernilai 0
   - Jika primary piece tidak ditemukan, heuristik bernilai maksimum (infinity)
b. Menentukan Orientasi Primary Piece
   - Jika baris (row) semua bagian primary piece sama, maka orientasinya horizontal
   - Jika tidak, orientasinya vertikal
c. Untuk Primary Piece Horizontal
   - Mencari posisi paling kiri (minCol) dan paling kanan (maxCol) primary piece
   - Jika exit berada pada baris yang sama dengan primary piece:
     - Jika exit di border kanan: jarak = cols - maxCol - 1
     - Jika exit di sebelah kanan primary piece: jarak = exitCol - maxCol
     - Jika exit di sebelah kiri primary piece: jarak = minCol - exitCol
   - Jika exit tidak pada baris yang sama: menghitung jarak Manhattan dengan rumus: |exitRow - primaryRow| + min(|exitCol - minCol|, |exitCol - maxCol|)
d. Untuk Primary Piece Vertikal:
   - Mencari posisi paling atas (minRow) dan paling bawah (maxRow) primary piece
   - Jika exit berada pada kolom yang sama dengan primary piece:
     - Jika exit di border bawah: jarak = rows - maxRow - 1
     - Jika exit di bawah primary piece: jarak = exitRow - maxRow
     - Jika exit di atas primary piece: jarak = minRow - exitRow
   - Jika exit tidak pada kolom yang sama: menghitung jarak Manhattan dengan rumus: |exitCol - primaryCol| + min(|exitRow - minRow|, |exitRow - maxRow|)

Heuristik 2: Blocking Pieces + Distance

```
// Calculate heuristic 2: Number of blocking pieces + distance
public int getHeuristic2() {
    if (isGoalState()) return 0;

    List<int[]> primaryPositions = pieces.get('P');
    if (primaryPositions == null) return Integer.MAX_VALUE;

    // Determine if primary piece is horizontal or vertical
    boolean isHorizontal = primaryPositions.size() > 1 &&
            primaryPositions.get(0)[0] == primaryPositions.get(1)[0];

    int blockingPieces = 0;
    int distance = 0;

    if (isHorizontal) {
        // Primary piece is horizontal
        int minCol = Integer.MAX_VALUE, maxCol = Integer.MIN_VALUE;
        for (int[] pos : primaryPositions) {
            minCol = Math.min(minCol, pos[1]);
            maxCol = Math.max(maxCol, pos[1]);
        }

        // Count blocking pieces and calculate distance
        if (exitRow == primaryRow) {
            if (exitCol == cols) {
                // Exit is at right border
                distance = cols - maxCol - 1;
                for (int j = maxCol + 1; j < cols; j++) {
                    if (board[exitRow][j] != '.') {
                        blockingPieces++;
                    }
                }
            } else if (exitCol > maxCol) {
                distance = exitCol - maxCol;
                for (int j = maxCol + 1; j < exitCol; j++) {
                    if (board[exitRow][j] != '.') {
                        blockingPieces++;
                    }
                }
            } else if (exitCol < minCol) {
                distance = minCol - exitCol;
                for (int j = exitCol + 1; j < minCol; j++) {
                    if (board[exitRow][j] != '.') {
                        blockingPieces++;
                    }
                }
            }
        } else {
            distance = Math.abs(exitRow - primaryRow);
            blockingPieces = 1; // Primary piece needs to be moved
vertically first
        }
    } else {
        // Primary piece is vertical
```

```
        int minRow = Integer.MAX_VALUE, maxRow = Integer.MIN_VALUE;
        for (int[] pos : primaryPositions) {
            minRow = Math.min(minRow, pos[0]);
            maxRow = Math.max(maxRow, pos[0]);
        }

        // Count blocking pieces and calculate distance
        if (exitCol == primaryCol) {
            if (exitRow == rows) {
                // Exit is at bottom border
                distance = rows - maxRow - 1;
                for (int i = maxRow + 1; i < rows; i++) {
                    if (board[i][exitCol] != '.') {
                        blockingPieces++;
                    }
                }
            } else if (exitRow > maxRow) {
                distance = exitRow - maxRow;
                for (int i = maxRow + 1; i < exitRow; i++) {
                    if (board[i][exitCol] != '.') {
                        blockingPieces++;
                    }
                }
            } else if (exitRow < minRow) {
                distance = minRow - exitRow;
                for (int i = exitRow + 1; i < minRow; i++) {
                    if (board[i][exitCol] != '.') {
                        blockingPieces++;
                    }
                }
            }
        } else {
            distance = Math.abs(exitCol - primaryCol);
            blockingPieces = 1; // Primary piece needs to be moved
horizontally first
        }
    }

    return blockingPieces * 2 + distance;
}
```

Heuristik kedua menggabungkan jarak dengan jumlah piece yang menghalangi jalur langsung dari primary piece ke pintu keluar. Heuristik ini lebih informatif karena mempertimbangkan penghalang dalam papan. Langkah-langkahnya sebagai berikut :

a. Pengecekan Awal
   - Jika state sudah merupakan goal state, heuristik bernilai 0
   - Jika primary piece tidak ditemukan, heuristik bernilai maksimum (infinity)
b. Inisialisasi Variabel:
   - blockingPieces: menghitung jumlah piece yang menghalangi
   - distance: menghitung jarak ke pintu keluar

c. Untuk Primary Piece Horizontal
   - Jika exit pada baris yang sama dengan primary piece:
     - Menghitung jarak seperti pada heuristik 1
     - Menghitung jumlah piece yang menghalangi dengan memeriksa setiap sel antara primary piece dan exit
   -Jika exit tidak pada baris yang sama:
     - Jarak = jarak vertikal antara primary piece dan exit
     - BlockingPieces = minimal 1 (karena primary piece perlu digeser vertikal terlebih dahulu, yang umumnya tidak mungkin)
d. Untuk Primary Piece Vertikal:
   - Jika exit pada kolom yang sama dengan primary piece:
     - Menghitung jarak seperti pada heuristik 1
     - Menghitung jumlah piece yang menghalangi dengan memeriksa setiap sel antara primary piece dan exit
   - Jika exit tidak pada kolom yang sama:
     - Jarak = jarak horizontal antara primary piece dan exit
     - BlockingPieces = minimal 1 (karena primary piece perlu digeser horizontal terlebih dahulu, yang umumnya tidak mungkin)
e. Perhitungan Nilai Akhir Heuristik:
   - Heuristik = blockingPieces * 2 + distance
   - Mengalikan blockingPieces dengan 2 memberikan bobot lebih pada piece yang menghalangi, karena mengeluarkan piece penghalang umumnya membutuhkan minimal 2 langkah (geser keluar, geser kembali)

6. Menyimpan State yang Dikunjungi

```
// Contoh implementasi dalam algoritma UCS
Set<RushHourGame> visited = new HashSet<>();
```

Semua algoritma pencarian menggunakan struktur data untuk melacak state yang telah dikunjungi guna menghindari eksplorasi berulang dan infinite loop. State disimpan dalam Set untuk memungkinkan pengecekan cepat apakah state sudah pernah dikunjungi sebelumnya.

7. Menyimpan State dalam Priority Queue

```
// Contoh implementasi dalam algoritma UCS
PriorityQueue<SearchNode> frontier = new PriorityQueue<>(
    Comparator.comparingInt(node -> node.g)
);
```

Algoritma pencarian menggunakan priority queue untuk menyimpan state yang akan dikunjungi, dengan prioritas berbeda tergantung algoritma yang digunakan.

8. Implementasi Node dan Membangun Jalur Solusi

```
class SearchNode implements Comparable<SearchNode> {
    public RushHourGame state;
    public SearchNode parent;
    public int g; // Cost from start
    public int h; // Heuristic value
    public int f; // Total cost (g + h)
    public String action; // Action taken to reach this state

    public SearchNode(RushHourGame state, SearchNode parent, int g, int h,
String action) {
        this.state = state;
        this.parent = parent;
        this.g = g;
        this.h = h;
        this.f = g + h;
        this.action = action;
    }

    @Override
    public int compareTo(SearchNode other) {
        if (this.f != other.f) {
            return Integer.compare(this.f, other.f);
        }
        // Tie-breaking: prefer node with lower h value
        return Integer.compare(this.h, other.h);
    }
}
```

Semua algoritma menggunakan kelas SearchNode untuk merepresentasikan node dalam ruang pencarian. Setiap node menyimpan:

- State permainan (state)
- Node induk (parent) untuk rekonstruksi jalur
- Cost dari state awal (g)
- Nilai heuristik (h)
- Total cost (f = g + h)
- Aksi yang diambil untuk mencapai state ini (action)

```
protected List<RushHourGame> buildSolution(SearchNode goalNode) {
    List<RushHourGame> path = new ArrayList<>();
    List<String> actions = new ArrayList<>();
    SearchNode current = goalNode;

    while (current != null) {
        path.add(0, current.state);
        if (current.action != null && !current.action.equals("Initial")) {
```

```
            actions.add(0, current.action);
        }
        current = current.parent;
    }

    solutionActions = actions;
    return path;
}
```

Ketika keadaan tujuan ditemukan, program merekonstruksi jalur solusi dengan menelusuri kembali dari node tujuan ke node awal melalui pointer parent.

9. Menentukan Aksi antara State

```
protected String getAction(RushHourGame from, RushHourGame to) {
    // Find which piece moved
    for (char piece : from.getPieces().keySet()) {
        List<int[]> fromPos = from.getPieces().get(piece);
        List<int[]> toPos = to.getPieces().get(piece);

        // Check if piece moved
        boolean moved = false;
        for (int i = 0; i < fromPos.size(); i++) {
            if (fromPos.get(i)[0] != toPos.get(i)[0] || fromPos.get(i)[1] !=
toPos.get(i)[1]) {
                moved = true;
                break;
            }
        }

        if (moved) {
            // Determine direction of movement
            int[] oldPos = fromPos.get(0);
            int[] newPos = toPos.get(0);

            if (oldPos[0] != newPos[0]) {
                // Vertical movement
                if (newPos[0] > oldPos[0]) {
                    return piece + "-bawah";
                } else {
                    return piece + "-atas";
                }
            } else {
                // Horizontal movement
                if (newPos[1] > oldPos[1]) {
                    return piece + "-kanan";
                } else {
                    return piece + "-kiri";
                }
            }
        }
    }
```

```
    return "Unknown";
}
```

Program menentukan aksi yang diambil untuk beralih dari satu state ke state lain dengan membandingkan posisi piece di kedua state. Fungsi ini:

- Iterasi melalui semua piece untuk menemukan yang bergerak
- Membandingkan posisi piece di state awal dan state akhir
- Menentukan arah pergerakan (atas, bawah, kiri, kanan)
- Mengembalikan string yang menggambarkan aksi (misalnya "P-kanan" untuk menggeser primary piece ke kanan)

Langkah-langkah umum ini digunakan oleh semua algoritma pathfinding yang diimplementasikan dalam program (UCS, GBFS, A*, dan IDA*). Setiap algoritma kemudian memiliki implementasi spesifik untuk strategi pencarian mereka masing-masing.

10. Pencarian melalui UCS

a. Inisialisasi

```
@Override
public List<RushHourGame> solve(RushHourGame initial) {
    resetCounters();
    long startTime = System.currentTimeMillis();

    // Priority queue ordered by g value (path cost)
    PriorityQueue<SearchNode> frontier = new PriorityQueue<>(new
Comparator<SearchNode>() {
        @Override
        public int compare(SearchNode n1, SearchNode n2) {
            if (n1.g != n2.g) {
                return Integer.compare(n1.g, n2.g);
            }
            // Tie-breaking: prefer older nodes (FIFO)
            return 0;
        }
    });

    Set<RushHourGame> explored = new HashSet<>();

    // Initialize with starting state
    frontier.add(new SearchNode(initial, null, 0, 0, "Initial"));

    // ...
}
```

Proses inisialisasi melibatkan beberapa tahapan. Pertama, dilakukan pengaturan struktur data, yaitu dengan menyiapkan *priority queue* (frontier) untuk

menyimpan node-node yang akan dieksplorasi, di mana prioritas diberikan berdasarkan nilai biaya jalur (g). Selain itu, disiapkan pula sebuah *set* (explored) yang berfungsi untuk mencatat semua state yang telah dikunjungi, serta penghitung waktu yang digunakan untuk mengukur kinerja algoritma secara keseluruhan. Selanjutnya, definisi prioritas ditetapkan berdasarkan nilai g, di mana node dengan nilai biaya jalur yang lebih rendah akan diprioritaskan. Apabila terdapat node dengan nilai biaya yang sama, maka algoritma memberikan preferensi pada node yang lebih lama berada di frontier (prinsip FIFO), guna menjaga konsistensi proses eksplorasi. Terakhir, proses inisialisasi dilakukan dengan menambahkan state awal ke dalam frontier dengan biaya awal sebesar nol. Karena algoritma UCS tidak menggunakan fungsi heuristik, parameter h dalam tahap ini diberikan nilai nol.

b. Iterasi Pencarian

```
while (!frontier.isEmpty()) {
    SearchNode node = frontier.poll();
    nodesVisited++;

    // Check if goal state reached
    if (node.state.isGoalState()) {
        executionTime = System.currentTimeMillis() - startTime;
        solution = buildSolution(node);
        return solution;
    }

    explored.add(node.state);

    // Expand current node
    for (RushHourGame nextState : node.state.getNextStates()) {
        if (!explored.contains(nextState)) {
            // Calculate new cost (each move has cost 1)
            int newCost = node.g + 1;

            // Check if state is already in frontier
            SearchNode existingNode = findInFrontier(frontier, nextState);

            if (existingNode == null) {
                // Add new node to frontier
                String action = getAction(node.state, nextState);
                frontier.add(new SearchNode(nextState, node, newCost, 0,
action));
            } else if (newCost < existingNode.g) {
                // Found better path to existing node
                frontier.remove(existingNode);
                String action = getAction(node.state, nextState);
                frontier.add(new SearchNode(nextState, node, newCost, 0,
action));
            }
```

```
        }
    }
}
```

Dalam proses iterasi pencarian menggunakan algoritma Uniform Cost Search (UCS), langkah pertama yang dilakukan adalah memilih node dengan biaya jalur paling rendah dari *priority queue* (frontier). Node ini kemudian diproses lebih lanjut, dan jumlah node yang telah dikunjungi dicatat untuk keperluan analisis kinerja. Setelah itu, sistem memeriksa apakah node yang sedang diproses merupakan *goal state*, yakni ketika kendaraan utama (primary piece) dapat keluar dari papan permainan. Jika kondisi ini terpenuhi, maka proses pencarian dihentikan dan jalur solusi dibentuk dengan menelusuri kembali langkah-langkah dari node tujuan hingga ke node awal.

Apabila node saat ini bukan *goal state*, algoritma akan menandainya sebagai telah dieksplorasi, lalu menghasilkan semua kemungkinan pergerakan berikutnya dengan menggeser satu kendaraan (piece) dalam berbagai arah yang valid. Setiap *state* baru yang dihasilkan akan dievaluasi lebih lanjut. Jika *state* tersebut belum pernah dieksplorasi sebelumnya, algoritma menghitung biaya totalnya (yakni biaya sebelumnya ditambah satu langkah), lalu memeriksa apakah *state* tersebut sudah ada di frontier.

Dalam tahap pembaruan frontier, jika *state* baru belum ada di frontier, maka akan langsung ditambahkan sebagai node baru. Namun, jika *state* sudah ada tetapi memiliki biaya yang lebih tinggi dibanding jalur baru yang ditemukan, maka node tersebut akan diperbarui dengan jalur yang lebih efisien. Sebaliknya, jika jalur yang lama sudah lebih baik, maka *state* baru tersebut diabaikan. Proses ini terus berulang hingga ditemukan solusi atau seluruh kemungkinan telah dieksplorasi.

c. Pencarian Dalam Frontier

```
@Override
protected SearchNode findInFrontier(PriorityQueue<SearchNode> frontier,
RushHourGame state) {
    for (SearchNode n : frontier) {
        if (n.state.equals(state)) {
            return n;
        }
    }
    return null;
}
```

Metode ini mencari apakah suatu state sudah ada dalam frontier iterasi melalui semua node dalam frontier, membandingkan state untuk menemukan kecocokan, dan mengembalikan node yang cocok jika ditemukan, atau null jika tidak.

d. Pembangunan Jalur Solusi

```
protected List<RushHourGame> buildSolution(SearchNode goalNode) {
    List<RushHourGame> path = new ArrayList<>();
    List<String> actions = new ArrayList<>();
    SearchNode current = goalNode;

    while (current != null) {
        path.add(0, current.state);
        if (current.action != null && !current.action.equals("Initial")) {
            actions.add(0, current.action);
        }
        current = current.parent;
    }

    solutionActions = actions;
    return path;
}
```

Setelah goal state ditemukan, algoritma membangun jalur solusi:

1. Membuat daftar untuk menyimpan state dan aksi dalam jalur solusi
2. Menelusuri dari node tujuan kembali ke node awal menggunakan pointer parent
3. Menambahkan setiap state dan aksi di awal daftar (untuk mempertahankan urutan)
4. Mengembalikan jalur solusi lengkap

11. Pencarian melalui GBFS

a. Inisialisasi

```
public class GBFS extends SearchAlgorithm {
    private int heuristicType;

    // Heuristic types
    public static final int HEURISTIC_MANHATTAN = 1;
    public static final int HEURISTIC_BLOCKING_PIECES = 2;

    /**
     * Constructor for GBFS
     * @param heuristicType Type of heuristic to use (1 or 2)
     */
    public GBFS(int heuristicType) {
        this.heuristicType = heuristicType;
    }
```

```
    /**
     * Calculate heuristic value based on selected heuristic type
     * @param state Game state to evaluate
     * @return Heuristic value
     */
    private int calculateHeuristic(RushHourGame state) {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return state.getHeuristic1();
            case HEURISTIC_BLOCKING_PIECES:
                return state.getHeuristic2();
            default:
                return state.getHeuristic1();
        }
    }

    // ...
}
```

Proses inisialisasi pada algoritma Greedy Best-First Search (GBFS) dimulai dengan pemilihan fungsi heuristik yang akan digunakan selama pencarian solusi. Algoritma ini mendukung dua jenis pendekatan heuristik yang berbeda. Yang pertama adalah HEURISTIC_MANHATTAN (H1), yaitu jarak Manhattan dari posisi kendaraan utama (primary piece) menuju pintu keluar. Heuristik ini mengukur seberapa jauh kendaraan utama secara horizontal dari tujuan akhir. Yang kedua adalah HEURISTIC_BLOCKING_PIECES (H2), yang mengombinasikan jarak tersebut dengan jumlah kendaraan lain yang menghalangi jalur keluarnya.

Setelah tipe heuristik dipilih, fungsi calculateHeuristic() digunakan untuk mengevaluasi setiap state berdasarkan kriteria heuristik tersebut. Dalam hal ini, semakin rendah nilai heuristik suatu state, maka semakin menjanjikan posisi tersebut dalam mendekati solusi. Artinya, algoritma akan lebih memprioritaskan eksplorasi terhadap state dengan nilai heuristik terendah, karena dianggap lebih dekat dengan kondisi tujuan.

b. Implementasi Algoritma Pencarian

```
@Override
public List<RushHourGame> solve(RushHourGame initial) {
    resetCounters();
    long startTime = System.currentTimeMillis();

    // Priority queue ordered by heuristic value (h only)
    PriorityQueue<SearchNode> frontier = new PriorityQueue<>(new
Comparator<SearchNode>() {
        @Override
        public int compare(SearchNode n1, SearchNode n2) {
```

```
            if (n1.h != n2.h) {
                return Integer.compare(n1.h, n2.h);
            }
            // Tie-breaking: prefer newer nodes (LIFO for better
performance)
            return 0;
        }
    });

    Set<RushHourGame> explored = new HashSet<>();
    Set<RushHourGame> frontierStates = new HashSet<>(); // Track states in
frontier

    // Calculate initial heuristic
    int h = calculateHeuristic(initial);
    frontier.add(new SearchNode(initial, null, 0, h, "Initial"));
    frontierStates.add(initial);

    // ...
}
```

Implementasi utama algoritma Greedy Best-First Search (GBFS) menggunakan beberapa struktur data penting untuk menunjang proses pencarian. Salah satunya adalah *frontier*, yaitu *priority queue* yang menyimpan node-node berdasarkan nilai heuristik terendah. Selain itu, digunakan juga *explored*, sebuah *set* yang mencatat semua state yang telah dieksplorasi untuk menghindari eksplorasi ulang. Untuk meningkatkan efisiensi saat memeriksa apakah suatu state sudah ada di frontier, ditambahkan pula *frontierStates*, yaitu *set* tambahan yang menyimpan representasi state yang sedang ada dalam antrian.

Dalam hal perbandingan antar-node, GBFS hanya mempertimbangkan nilai heuristik (h) sebagai dasar prioritas. Berbeda dari algoritma lain yang mempertimbangkan kombinasi antara biaya jalur dan heuristik, GBFS fokus sepenuhnya pada estimasi jarak ke tujuan. Jika ada dua node dengan nilai heuristik yang sama, algoritma ini akan lebih memilih node yang lebih baru dimasukkan (mengikuti prinsip LIFO), karena pendekatan ini terbukti dapat meningkatkan kinerja dalam banyak kasus.

Saat pencarian dimulai, algoritma terlebih dahulu menghitung nilai heuristik untuk state awal berdasarkan fungsi heuristik yang telah dipilih. Meskipun GBFS tidak menggunakan nilai biaya jalur (g) dalam proses evaluasi, nilai g untuk node awal tetap diset ke nol dan kemudian state tersebut dimasukkan ke dalam frontier sebagai titik awal pencarian solusi.

c. Iterasi Pencarian

```
while (!frontier.isEmpty()) {
    SearchNode node = frontier.poll();
    frontierStates.remove(node.state);
    nodesVisited++;

    // Check if goal state reached
    if (node.state.isGoalState()) {
        executionTime = System.currentTimeMillis() - startTime;
        solution = buildSolution(node);
        return solution;
    }

    explored.add(node.state);

    // Expand current node
    for (RushHourGame nextState : node.state.getNextStates()) {
        if (!explored.contains(nextState) &&
!frontierStates.contains(nextState)) {
            // Calculate heuristic for new state
            h = calculateHeuristic(nextState);
            String action = getAction(node.state, nextState);

            // GBFS doesn't track g value, but we set it for consistency
            frontier.add(new SearchNode(nextState, node, node.g + 1, h,
action));
            frontierStates.add(nextState);
        }
    }
}
```

Dalam proses iterasi pencarian dengan algoritma Greedy Best-First Search (GBFS), langkah pertama yang dilakukan adalah mengeksplorasi node dengan nilai heuristik paling rendah dari frontier. Setelah node tersebut diambil, state-nya juga dihapus dari *frontierStates* agar tidak terjadi duplikasi. Pada saat yang sama, jumlah node yang telah dikunjungi dicatat untuk keperluan analisis performa algoritma. Setelah itu, dilakukan pengecekan apakah state yang sedang dievaluasi merupakan *goal state*—yakni kondisi di mana kendaraan utama sudah dapat keluar dari papan permainan. Jika ya, algoritma langsung membangun dan mengembalikan jalur solusi, dengan cara yang sama seperti pada UCS, yaitu dengan menelusuri kembali jejak langkah dari node tujuan ke node awal.

Jika state tersebut bukanlah tujuan, maka ia akan ditandai sebagai sudah dieksplorasi dengan memasukkannya ke dalam *explored set*. Selanjutnya, semua kemungkinan pergerakan dari state tersebut akan dibangkitkan untuk menghasilkan *successor states* yang baru. Untuk setiap *successor* yang belum pernah ada di *explored* maupun *frontier*, algoritma akan menghitung nilai heuristiknya, mencatat langkah yang diambil untuk mencapainya, lalu menambahkan state tersebut ke frontier dan *frontierStates*. Berbeda dengan UCS,

GBFS tidak melakukan pemeriksaan apakah ada jalur yang lebih baik menuju *state* yang sama jika sebelumnya sudah ada di frontier. Algoritma ini hanya berfokus pada nilai heuristik dan mengabaikan total biaya perjalanan sejauh ini. Karena itu, GBFS tidak memperbarui jalur meskipun ditemukan jalur yang lebih efisien. Semua keputusan eksplorasi hanya bergantung pada seberapa dekat sebuah state terlihat dari tujuan menurut heuristik, sehingga prioritas selalu diberikan pada node dengan nilai h terendah.

d. Analisis Heuristik

```
public String analyzeHeuristic() {
    if (solution == null) {
        return "No solution found - cannot analyze heuristic.";
    }

    StringBuilder analysis = new StringBuilder();
    analysis.append("Heuristic Analysis for
").append(getHeuristicName()).append(":\n");

    // Calculate average heuristic value throughout solution
    int totalH = 0;
    for (RushHourGame state : solution) {
        totalH += calculateHeuristic(state);
    }
    double avgH = (double) totalH / solution.size();

    analysis.append("- Average heuristic value:
").append(String.format("%.2f", avgH)).append("\n");
    analysis.append("- Initial heuristic:
").append(calculateHeuristic(solution.get(0))).append("\n");
    analysis.append("- Final heuristic:
").append(calculateHeuristic(solution.get(solution.size() -
1))).append("\n");

    // Check if heuristic generally decreases
    int decreases = 0;
    int increases = 0;
    for (int i = 1; i < solution.size(); i++) {
        int prevH = calculateHeuristic(solution.get(i - 1));
        int currH = calculateHeuristic(solution.get(i));
        if (currH < prevH) decreases++;
        else if (currH > prevH) increases++;
    }

    analysis.append("- Heuristic decreases: ").append(decreases).append("
times\n");
    analysis.append("- Heuristic increases: ").append(increases).append("
times\n");

    if (decreases > increases) {
        analysis.append("- The heuristic generally guides toward the goal
```

```
effectively.\n");
    } else {
        analysis.append("- The heuristic may not be very informative for
this problem.\n");
    }

    return analysis.toString();
}
```

Analisis statistik terhadap nilai heuristik, yaitu dengan menghitung rata-rata nilai heuristik yang muncul sepanjang jalur solusi. Selain itu, algoritma juga mencatat nilai heuristik pada state awal dan akhir sebagai bagian dari data evaluasi. Dari data tersebut, kemudian dianalisis tren nilai heuristik selama proses pencarian. Tujuannya adalah untuk melihat apakah nilai heuristik secara umum mengalami penurunan, yang menandakan bahwa algoritma memang bergerak menuju solusi. Dalam proses ini juga dihitung seberapa sering nilai heuristik menurun dibandingkan dengan meningkat dari satu langkah ke langkah berikutnya.

Hasil analisis ini digunakan untuk menilai sejauh mana heuristik yang digunakan benar-benar membantu dalam memandu pencarian. Jika jumlah penurunan nilai heuristik lebih banyak dibanding peningkatan, maka heuristik dianggap efektif dan relevan terhadap permasalahan yang dihadapi. Sebaliknya, jika nilai heuristik sering meningkat atau tidak menunjukkan pola penurunan yang konsisten, maka heuristik tersebut kemungkinan kurang informatif atau kurang sesuai untuk digunakan dalam konteks puzzle Rush Hour.

12. Pencarian melalui A*

a. Inisialisasi dan Pemilihan Heuristik

```
public class AStar extends SearchAlgorithm {
    private int heuristicType;

    // Heuristic types
    public static final int HEURISTIC_MANHATTAN = 1;
    public static final int HEURISTIC_BLOCKING_PIECES = 2;

    /**
     * Constructor for A*
     * @param heuristicType Type of heuristic to use (1 or 2)
     */
    public AStar(int heuristicType) {
        this.heuristicType = heuristicType;
    }

    /**
     * Calculate heuristic value based on selected heuristic type
```

```
     * @param state Game state to evaluate
     * @return Heuristic value
     */
    private int calculateHeuristic(RushHourGame state) {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return state.getHeuristic1();
            case HEURISTIC_BLOCKING_PIECES:
                return state.getHeuristic2();
            default:
                return state.getHeuristic1();
        }
    }

    // ...
}
```

Seperti GBFS, A* mendukung dua tipe heuristik berbeda dan memiliki metode untuk menghitung nilai heuristik. Namun, cara A* menggunakan heuristik berbeda secara fundamental.

b. Implementasi Algoritma Pencarian

```
@Override
public List<RushHourGame> solve(RushHourGame initial) {
    resetCounters();
    long startTime = System.currentTimeMillis();

    // Priority queue ordered by f = g + h
    PriorityQueue<SearchNode> frontier = new PriorityQueue<>(new
Comparator<SearchNode>() {
        @Override
        public int compare(SearchNode n1, SearchNode n2) {
            if (n1.f != n2.f) {
                return Integer.compare(n1.f, n2.f);
            }
            // Tie-breaking: prefer node with lower h value
            if (n1.h != n2.h) {
                return Integer.compare(n1.h, n2.h);
            }
            // Second tie-breaking: prefer newer nodes
            return 0;
        }
    });

    Set<RushHourGame> explored = new HashSet<>();
    Map<RushHourGame, Integer> frontierCosts = new HashMap<>(); // Track
best g values in frontier

    // Calculate initial heuristic
    int h = calculateHeuristic(initial);
    SearchNode initialNode = new SearchNode(initial, null, 0, h,
```

```
"Initial");
    frontier.add(initialNode);
    frontierCosts.put(initial, 0);

    // ...
}
```

Implementasi algoritma A* memiliki beberapa perbedaan mendasar jika dibandingkan dengan Uniform Cost Search (UCS) maupun Greedy Best-First Search (GBFS). Salah satu perbedaan utamanya terletak pada cara mengevaluasi setiap node. A* menggunakan fungsi evaluasi total berupa $f(n) = g(n) + h(n)$, di mana $g(n)$ mewakili biaya aktual dari titik awal hingga node saat ini, sedangkan $h(n)$ merupakan perkiraan biaya dari node tersebut menuju tujuan. Kombinasi kedua nilai ini memungkinkan A* untuk menyeimbangkan antara eksplorasi berdasarkan jalur terpendek sejauh ini dan estimasi jarak ke tujuan.

Untuk mendukung proses ini, struktur data yang digunakan dalam A* lebih kompleks dibanding algoritma lain. *Priority queue* atau frontier tetap menjadi komponen utama, namun kali ini prioritas ditentukan berdasarkan nilai *f* terendah. Selain itu, digunakan juga *set* untuk mencatat semua state yang telah dieksplorasi agar tidak dievaluasi ulang. Tambahan penting lainnya adalah penggunaan *HashMap* yang menyimpan biaya *g* terbaik (terkecil) untuk setiap state yang masih berada di frontier, guna memastikan hanya jalur optimal yang diproses.

Dalam situasi di mana beberapa node memiliki nilai *f* yang sama, A* menerapkan strategi *tie-breaking* yang cermat. Node dengan nilai *h* lebih rendah akan diprioritaskan karena dianggap lebih dekat ke tujuan. Jika nilai *h* juga sama, maka node yang lebih baru (dimasukkan belakangan) akan dipilih lebih dahulu, dengan pendekatan LIFO yang terbukti dapat membantu meningkatkan performa dalam beberapa kasus.

c. Iterasi Pencarian

```
while (!frontier.isEmpty()) {
    SearchNode node = frontier.poll();

    // Remove from frontier costs when popped
    frontierCosts.remove(node.state);
    nodesVisited++;

    // Check if goal state reached
    if (node.state.isGoalState()) {
        executionTime = System.currentTimeMillis() - startTime;
        solution = buildSolution(node);
        return solution;
```

```
    }

    explored.add(node.state);

    // Expand current node
    for (RushHourGame nextState : node.state.getNextStates()) {
        int newG = node.g + 1; // Each move has cost 1

        if (!explored.contains(nextState)) {
            Integer frontierG = frontierCosts.get(nextState);

            if (frontierG == null) {
                // New state not in frontier
                h = calculateHeuristic(nextState);
                String action = getAction(node.state, nextState);
                SearchNode newNode = new SearchNode(nextState, node, newG,
h, action);
                frontier.add(newNode);
                frontierCosts.put(nextState, newG);
            } else if (newG < frontierG) {
                // Found better path to state in frontier
                // Remove old node (we need to do this manually in Java)
                removeFromFrontier(frontier, nextState);
                frontierCosts.remove(nextState);

                // Add new better node
                h = calculateHeuristic(nextState);
                String action = getAction(node.state, nextState);
                SearchNode newNode = new SearchNode(nextState, node, newG,
h, action);
                frontier.add(newNode);
                frontierCosts.put(nextState, newG);
            }
        }
    }
}
```

Setiap langkah dimulai dengan mengambil node dari frontier yang memiliki nilai *f* paling rendah, di mana *f* merupakan hasil penjumlahan antara *g* (biaya aktual dari awal) dan *h* (estimasi biaya menuju tujuan). Setelah node tersebut diambil, informasi state-nya juga dihapus dari *frontierCosts*, dan kunjungannya dicatat sebagai bagian dari analisis performa. Selanjutnya, algoritma memproses semua *successor* atau state baru yang dapat dihasilkan dari node tersebut. Untuk setiap state baru yang belum pernah dieksplorasi, A* menghitung nilai *g* baru dengan menambahkan satu langkah ke nilai sebelumnya. Setelah itu, algoritma memeriksa apakah state tersebut sudah ada di frontier dengan melihat pada *frontierCosts*.

Jika state belum ada di frontier, maka ia langsung ditambahkan bersama dengan nilai *f*, *g*, dan *h* yang sesuai. Namun, jika state tersebut sudah ada di frontier tetapi

memiliki nilai *g* yang lebih tinggi dari jalur baru yang ditemukan, maka node lama akan dihapus terlebih dahulu dari frontier menggunakan metode khusus, lalu digantikan dengan node baru yang menawarkan jalur lebih efisien. Sebaliknya, apabila state tersebut sudah ada dan memiliki biaya *g* yang lebih rendah, maka *successor* baru itu akan diabaikan karena dianggap tidak menawarkan jalur yang lebih baik.

13. Pencarian melalui IDA*

a. Inisialisasi dan Konsep Dasar

```
public class IDAStar extends SearchAlgorithm {
    private int heuristicType;
    private int threshold;
    private int nextThreshold;

    // Heuristic types
    public static final int HEURISTIC_MANHATTAN = 1;
    public static final int HEURISTIC_BLOCKING_PIECES = 2;

    /**
     * Constructor for IDA*
     * @param heuristicType Type of heuristic to use (1 or 2)
     */
    public IDAStar(int heuristicType) {
        this.heuristicType = heuristicType;
    }

    /**
     * Calculate heuristic value based on selected heuristic type
     * @param state Game state to evaluate
     * @return Heuristic value
     */
    private int calculateHeuristic(RushHourGame state) {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return state.getHeuristic1();
            case HEURISTIC_BLOCKING_PIECES:
                return state.getHeuristic2();
            default:
                return state.getHeuristic1();
        }
    }
}
```

IDA* menggunakan threshold yaitu batas nilai f untuk iterasi saat ini dan nextThreshold berisi nilai f terkecil yang melebihi batas saat ini, digunakan sebagai batas untuk iterasi berikutnya. IDA* juga menggunakan 2 tipe heuristik.

b. Implementasi Algoritma Utama

```
@Override
public List<RushHourGame> solve(RushHourGame initial) {
    resetCounters();
    long startTime = System.currentTimeMillis();

    // Initialize threshold with initial heuristic value
    threshold = calculateHeuristic(initial);

    while (threshold != Integer.MAX_VALUE) {
        nextThreshold = Integer.MAX_VALUE;
        SearchNode result = depthLimitedSearch(initial, 0, new
ArrayList<>());

        if (result != null) {
            // Solution found
            executionTime = System.currentTimeMillis() - startTime;
            solution = buildSolution(result);
            return solution;
        }

        threshold = nextThreshold;
    }

    executionTime = System.currentTimeMillis() - startTime;
    return null; // No solution found
}
```

Proses dimulai dengan menetapkan batas awal berdasarkan nilai heuristik dari state awal. Setelah itu, algoritma melakukan serangkaian pencarian berulang, di mana setiap pencarian menggunakan metode *depth-first search* yang dibatasi oleh nilai f (hasil dari $g + h$). Jika selama pencarian nilai f dari suatu node melebihi batas saat ini, maka node tersebut tidak dieksplorasi lebih lanjut (*pruned*).

Jika tidak ditemukan solusi dalam satu iterasi, maka batas pencarian akan diperbarui menjadi *nextThreshold*, yaitu nilai f terkecil yang melebihi ambang batas sebelumnya. Proses ini terus diulang, dan setiap kali ambang batas dinaikkan secara dinamis berdasarkan hasil pencarian terakhir. Pencarian hanya akan berhenti jika solusi ditemukan atau ambang batas mencapai *Integer.MAX_VALUE*, yang menandakan bahwa tidak ada solusi yang bisa ditemukan dalam ruang pencarian yang tersedia.

Setiap kali pencarian dilakukan untuk satu ambang batas tertentu, IDA* menjalankan pencarian secara rekursif dengan metode *depthLimitedSearch*. Pada tahap ini, state awal selalu diasumsikan memiliki nilai g sebesar nol karena belum ada langkah yang diambil. Untuk mendeteksi kemungkinan siklus atau perulangan, algoritma juga menggunakan daftar kosong di awal untuk mencatat jalur eksplorasi.

c. Implementasi Pencarian Depth-Limited

```
private SearchNode depthLimitedSearch(RushHourGame state, int g,
List<RushHourGame> path) {
    nodesVisited++;

    int h = calculateHeuristic(state);
    int f = g + h;

    // If f exceeds threshold, update next threshold and prune
    if (f > threshold) {
        nextThreshold = Math.min(nextThreshold, f);
        return null;
    }

    // Check if goal reached
    if (state.isGoalState()) {
        // Create solution node
        SearchNode solutionNode = new SearchNode(state, null, g, h,
"Goal");

        // Build parent nodes for solution path
        SearchNode current = solutionNode;
        for (int i = path.size() - 1; i >= 0; i--) {
            RushHourGame prevState = path.get(i);
            String action = i > 0 ? getAction(path.get(i - 1), prevState)
: "Initial";
            SearchNode parent = new SearchNode(prevState, null, i,
calculateHeuristic(prevState), action);
            current.parent = parent;
            current = parent;
        }

        return solutionNode;
    }

    // Expand current node
    List<RushHourGame> nextStates = state.getNextStates();

    // Sort by f-value for better pruning (optional optimization)
    nextStates.sort((s1, s2) -> {
        int f1 = (g + 1) + calculateHeuristic(s1);
        int f2 = (g + 1) + calculateHeuristic(s2);
        return Integer.compare(f1, f2);
    });

    for (RushHourGame nextState : nextStates) {
        // Check for cycles in path
        if (!path.contains(nextState)) {
            path.add(state); // Add current state to path
            SearchNode result = depthLimitedSearch(nextState, g + 1,
path);
            path.remove(path.size() - 1); // Remove state from path
```

```
            if (result != null) {
                // Update parent link
                if (result.parent == null &&
result.state.equals(nextState)) {
                    String action = getAction(state, nextState);
                    result.parent = new SearchNode(state, null, g, h,
action);
                }
                return result;
            }
        }
    }

    return null; // No solution found at this threshold
}
```

Setiap state yang dievaluasi akan dihitung nilai *f*-nya, yaitu hasil penjumlahan antara *g* (biaya dari awal ke titik saat ini) dan *h* (estimasi jarak ke tujuan). Jika nilai *f* ini melebihi ambang batas (*threshold*) yang sedang digunakan, maka eksplorasi terhadap node tersebut dihentikan. Namun, sebelum benar-benar dihentikan, nilai *f* tersebut dibandingkan dengan *nextThreshold*, dan jika lebih kecil, maka nilainya disimpan sebagai kandidat ambang batas untuk iterasi selanjutnya.Setelah melewati perhitungan awal, algoritma kemudian memeriksa apakah state saat ini sudah merupakan *goal state*. Jika ya, maka proses pencarian dihentikan dan jalur solusi mulai dibentuk dengan menelusuri kembali urutan langkah yang telah tersimpan dalam path. Untuk mempercepat proses pencarian dan mengurangi eksplorasi yang tidak perlu, semua *successor* dari state saat ini diurutkan berdasarkan nilai *f*-nya. Tujuannya adalah agar state yang lebih menjanjikan (dengan nilai *f* lebih kecil) diuji terlebih dahulu, sehingga kemungkinan pemangkasan (pruning) lebih cepat terjadi.

Salah satu fitur penting dari *depthLimitedSearch* adalah deteksi siklus. Jika ada successor yang sudah muncul dalam *path* tersebut, maka ia akan diabaikan agar tidak terjadi pengulangan yang tidak perlu. Ini sangat penting dalam strategi *depth-first* untuk menghindari eksplorasi tak terbatas. Setiap *successor* yang valid kemudian dieksplorasi secara rekursif. Sebelum proses pencarian dilanjutkan ke anak node, state saat ini ditambahkan ke dalam *path*. Setelah pencarian rekursif selesai, entri tersebut dihapus kembali sebagai bagian dari proses *backtracking*. Jika pada akhirnya solusi ditemukan, maka algoritma akan memperbarui referensi ke node induk di sepanjang jalur tersebut.

BAB III
ANALISIS ALGORITMA

Dalam konteks algoritma pencarian, f(n) merupakan fungsi evaluasi yang digunakan untuk menentukan prioritas eksplorasi node. Sedangkan g(n) didefinisikan sebagai biaya sebenarnya dari node awal (start node) ke node n dalam pohon pencarian. Dalam algoritma UCS (Uniform Cost Search), fungsi evaluasi f(n) = g(n), yang berarti UCS hanya memperhitungkan biaya perjalanan yang telah dilalui. Untuk Greedy Best-First Search, f(n) = h(n), dimana h(n) adalah estimasi biaya dari node n ke goal state, tanpa mempertimbangkan biaya perjalanan yang telah dilalui. Pada A* dan IDA*, f(n) = g(n) + h(n), yang merupakan kombinasi dari biaya perjalanan sejauh ini dan estimasi biaya ke tujuan, sehingga menyeimbangkan eksplorasi dan pemanfaatan informasi heuristik.

Heuristik yang digunakan pada algoritma A* dikatakan admissible jika untuk setiap node n, $h(n) \leq h^*(n)$, dimana $h^*(n)$ adalah biaya sebenarnya untuk mencapai goal dari node n. Dengan kata lain, heuristik admissible tidak pernah melebih-lebihkan (overestimate) biaya perjalanan sebenarnya ke tujuan. Dalam konteks penyelesaian Rush Hour, heuristik Manhattan distance (H1) dapat dianggap admissible karena jarak Manhattan selalu lebih kecil atau sama dengan jumlah langkah minimum yang dibutuhkan untuk memindahkan mobil dari posisi saat ini ke pintu keluar. Namun, heuristik blocking pieces + distance (H2) mungkin tidak selalu admissible karena penambahan faktor penalti untuk piece yang menghalangi bisa menghasilkan nilai yang melebihi biaya sebenarnya, tergantung pada bagaimana penalti tersebut diterapkan.

Dalam penyelesaian Rush Hour, algoritma UCS akan sama dengan BFS jika semua langkah pergerakan mobil memiliki biaya yang sama. Hal ini karena UCS memperluas node berdasarkan biaya perjalanan terendah dari start state, sedangkan BFS memperluas node berdasarkan jumlah langkah terkecil dari start state. Ketika semua langkah memiliki biaya yang sama (misalnya, setiap pergerakan mobil dihitung sebagai satu langkah), maka urutan ekspansi node dan path yang dihasilkan akan identik antara UCS dan BFS. Namun, jika ada perbedaan biaya antar langkah (misalnya, jika menggeser mobil yang berbeda memiliki biaya yang berbeda), maka UCS dan BFS akan menghasilkan urutan ekspansi dan solusi yang berbeda.

Secara teoritis, algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada penyelesaian Rush Hour, terutama untuk puzzle yang kompleks. Hal ini karena A* menggunakan informasi heuristik untuk mengarahkan pencarian ke arah tujuan, sehingga dapat mengurangi jumlah node yang perlu dieksplorasi. UCS tidak memiliki informasi tentang seberapa dekat suatu state dengan tujuan, sehingga cenderung mengeksplorasi node dalam pola melingkar dari start state, termasuk banyak node yang mungkin menjauhi tujuan. Dengan heuristik yang baik, A* dapat langsung "mengarah" ke tujuan, mengeksplorasi jalur yang menjanjikan terlebih dahulu. Namun, efisiensi A* sangat bergantung pada kualitas heuristik yang digunakan. Dengan heuristik yang buruk atau tidak informatif, kinerja A* bisa mendekati atau bahkan lebih buruk dari UCS.

Greedy Best-First Search tidak menjamin solusi optimal untuk penyelesaian Rush Hour. Algoritma ini hanya mempertimbangkan nilai heuristik (estimasi jarak ke tujuan) dan mengabaikan biaya perjalanan sejauh ini. Akibatnya, Greedy Best-First Search cenderung memilih jalur yang tampak menjanjikan dalam jangka pendek, tetapi tidak mempertimbangkan total biaya perjalanan. Hal ini dapat menyebabkan algoritma terjebak dalam jalur yang terlihat dekat ke tujuan secara heuristik tetapi sebenarnya membutuhkan banyak langkah. Contohnya dalam pencarian rute di Romania (dapat diakses pada link ini), Greedy Best-First Search menemukan rute Arad→Sibiu→Fagaras→Bucharest dengan total biaya 450, padahal rute optimal adalah Arad→Sibiu→Rimniciu Vilcea→Pitesti→Bucharest dengan biaya 418.

IDA* menggabungkan keunggulan ruang dari Iterative Deepening Search dengan panduan heuristik dari A*. Algoritma ini melakukan serangkaian pencarian depth-first dengan batas yang ditingkatkan berdasarkan nilai f(n). Keunggulan utama IDA* adalah efisiensi memori, dengan kompleksitas ruang hanya O(d) dimana d adalah kedalaman solusi. Ini jauh lebih baik daripada A* yang memiliki kompleksitas ruang O(b^d), terutama untuk puzzle dengan ruang state yang sangat besar. Namun, IDA* mungkin melakukan kalkulasi berulang karena tidak menyimpan state yang telah dikunjungi antara iterasi, yang dapat menyebabkan waktu eksekusi yang lebih lama. Dalam konteks Rush Hour dengan banyak mobil atau papan yang besar, IDA* menjadi pilihan yang baik karena efisiensi memorinya dan tetap menjamin solusi optimal jika heuristik admissible.

Dalam penyelesaian Rush Hour Puzzle, pemilihan algoritma pathfinding harus mempertimbangkan karakteristik dan kebutuhan spesifik dari permasalahan. UCS menjamin solusi optimal tetapi tidak efisien untuk puzzle kompleks. Greedy Best-First Search cepat tetapi tidak menjamin optimalitas. A* menyeimbangkan optimalitas dan efisiensi jika heuristik yang digunakan admissible, menjadikannya pilihan populer untuk banyak kasus. IDA* mengatasi keterbatasan memori A* dengan mengorbankan waktu komputasi. Penggunaan heuristik yang efektif dan admissible seperti Manhattan distance sangat penting untuk kinerja algoritma pencarian terinformasi dalam menemukan solusi optimal dengan efisien.

BAB IV

SOURCE CODE PROGRAM

SOURCE CODE

4.1 AStar.java

```
package src;

import java.util.*;

public class AStar extends SearchAlgorithm {
    private int heuristicType;

    // Heuristic types
    public static final int HEURISTIC_MANHATTAN = 1;
    public static final int HEURISTIC_BLOCKING_PIECES = 2;


    public AStar(int heuristicType) {
        this.heuristicType = heuristicType;
    }

    @Override
    public List<RushHourGame> solve(RushHourGame initial) {
        resetCounters();
        long startTime = System.currentTimeMillis();

        // Priority queue ordered by f = g + h
        PriorityQueue<SearchNode> frontier = new PriorityQueue<>(new
Comparator<SearchNode>() {
            @Override
            public int compare(SearchNode n1, SearchNode n2) {
                if (n1.f != n2.f) {
                    return Integer.compare(n1.f, n2.f);
                }
                // Tie-breaking: prefer node with lower h value
                if (n1.h != n2.h) {
                    return Integer.compare(n1.h, n2.h);
                }
                // Second tie-breaking: prefer newer nodes
                return 0;
            }
        });

        Set<RushHourGame> explored = new HashSet<>();
        Map<RushHourGame, Integer> frontierCosts = new HashMap<>(); // Track
best g values in frontier

        // Calculate initial heuristic
        int h = calculateHeuristic(initial);
        SearchNode initialNode = new SearchNode(initial, null, 0, h, "Initial");
        frontier.add(initialNode);
```

```
        frontierCosts.put(initial, 0);

        while (!frontier.isEmpty()) {
            SearchNode node = frontier.poll();

            // Remove from frontier costs when popped
            frontierCosts.remove(node.state);
            nodesVisited++;

            // Check if goal state reached
            if (node.state.isGoalState()) {
                executionTime = System.currentTimeMillis() - startTime;
                solution = buildSolution(node);
                return solution;
            }

            explored.add(node.state);

            // Expand current node
            for (RushHourGame nextState : node.state.getNextStates()) {
                int newG = node.g + 1; // Each move has cost 1

                if (!explored.contains(nextState)) {
                    Integer frontierG = frontierCosts.get(nextState);

                    if (frontierG == null) {
                        // New state not in frontier
                        h = calculateHeuristic(nextState);
                        String action = getAction(node.state, nextState);
                        SearchNode newNode = new SearchNode(nextState, node,
newG, h, action);
                        frontier.add(newNode);
                        frontierCosts.put(nextState, newG);
                    } else if (newG < frontierG) {
                        // Found better path to state in frontier
                        // Remove old node (we need to do this manually in Java)
                        removeFromFrontier(frontier, nextState);
                        frontierCosts.remove(nextState);

                        // Add new better node
                        h = calculateHeuristic(nextState);
                        String action = getAction(node.state, nextState);
                        SearchNode newNode = new SearchNode(nextState, node,
newG, h, action);
                        frontier.add(newNode);
                        frontierCosts.put(nextState, newG);
                    }
                }
            }
        }

        executionTime = System.currentTimeMillis() - startTime;
        return null; // No solution found
    }

    private void removeFromFrontier(PriorityQueue<SearchNode> frontier,
```

```
RushHourGame state) {
        Iterator<SearchNode> iterator = frontier.iterator();
        while (iterator.hasNext()) {
            SearchNode node = iterator.next();
            if (node.state.equals(state)) {
                iterator.remove();
                break;
            }
        }
    }

    private int calculateHeuristic(RushHourGame state) {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return state.getHeuristic1();
            case HEURISTIC_BLOCKING_PIECES:
                return state.getHeuristic2();
            default:
                return state.getHeuristic1();
        }
    }

    @Override
    public String getAlgorithmName() {
        return "A* Search";
    }

    @Override
    public String getAlgorithmDescription() {
        String heuristicName = getHeuristicName();
        return "A* combines path cost (g) and heuristic (h) to find optimal
solution efficiently. " +
                "Current heuristic: " + heuristicName + ". " +
                "Guarantees optimal solution when heuristic is admissible.";
    }

    public String getHeuristicName() {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return "Manhattan Distance (H1)";
            case HEURISTIC_BLOCKING_PIECES:
                return "Blocking Pieces + Distance (H2)";
            default:
                return "Unknown";
        }
    }

    public int getHeuristicType() {
        return heuristicType;
    }

    public void setHeuristicType(int heuristicType) {
        this.heuristicType = heuristicType;
    }

    public Map<String, Object> getStatistics() {
```

```
        Map<String, Object> stats = new HashMap<>();
        stats.put("algorithm", getAlgorithmName());
        stats.put("heuristic", getHeuristicName());
        stats.put("heuristic_type", heuristicType);
        stats.put("nodes_visited", nodesVisited);
        stats.put("execution_time", executionTime);
        stats.put("solution_length", solution != null ? solution.size() - 1 :
-1);
        stats.put("optimal", isHeuristicAdmissible()); // Optimal if heuristic
is admissible
        return stats;
    }

    public boolean isHeuristicAdmissible() {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return true; // Manhattan distance is admissible for Rush Hour
            case HEURISTIC_BLOCKING_PIECES:
                return false; // May overestimate, making it non-admissible
            default:
                return false;
        }
    }

    @Override
    public void printSolution() {
        System.out.println("=== " + getAlgorithmName() + " Results ===");
        System.out.println(getAlgorithmDescription());
        System.out.println();

        if (solution == null || solution.isEmpty()) {
            System.out.println("No solution found!");
            System.out.println("Nodes visited: " + nodesVisited);
            System.out.println("Execution time: " + executionTime + " ms");
            System.out.println("Heuristic used: " + getHeuristicName());
            System.out.println("Heuristic admissible: " +
isHeuristicAdmissible());
            return;
        }

        System.out.println("Solution found!");
        System.out.println("Nodes visited: " + nodesVisited);
        System.out.println("Execution time: " + executionTime + " ms");
        System.out.println("Solution length: " + (solution.size() - 1) + "
moves");
        System.out.println("Heuristic used: " + getHeuristicName());
        System.out.println("Heuristic admissible: " + isHeuristicAdmissible());
        System.out.println("Status: " + (isHeuristicAdmissible() ? "OPTIMAL" :
"NON-OPTIMAL"));
        System.out.println();

        // Print solution path with f, g, h values
        System.out.println("Papan Awal");
        printColoredBoard(solution.get(0), null);
        int h0 = calculateHeuristic(solution.get(0));
        System.out.println("(g=0, h=" + h0 + ", f=" + h0 + ")");
```

```
        System.out.println();

        // Print each move with f, g, h values
        for (int i = 1; i < solution.size(); i++) {
            int g = i;
            int h = calculateHeuristic(solution.get(i));
            int f = g + h;
            System.out.println("Gerakan " + i + ": " + solutionActions.get(i -
1) +
                    " (g=" + g + ", h=" + h + ", f=" + f + ")");
            printColoredBoard(solution.get(i), solutionActions.get(i - 1));
            System.out.println();
        }
    }

    private void printColoredBoard(RushHourGame game, String action) {
        char[][] board = game.getBoard();
        int rows = game.getRows();
        int cols = game.getCols();

        // ANSI color codes
        String RESET = "\u001B[0m";
        String RED = "\u001B[31m";      // Primary piece
        String GREEN = "\u001B[32m";    // Exit
        String YELLOW = "\u001B[33m";   // Moved piece
        String BLUE = "\u001B[34m";     // Other pieces

        char movedPiece = '.';
        if (action != null && !action.equals("Initial")) {
            movedPiece = action.charAt(0);
        }

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                char c = board[i][j];

                if (c == game.getPrimaryPiece()) {
                    System.out.print(RED + c + RESET);
                } else if (c == game.getExit()) {
                    System.out.print(GREEN + c + RESET);
                } else if (c == movedPiece && c != '.') {
                    System.out.print(YELLOW + c + RESET);
                } else if (c != '.' && c != game.getExit()) {
                    System.out.print(BLUE + c + RESET);
                } else {
                    System.out.print(c);
                }
            }
            System.out.println();
        }
    }

    public String analyzeAlgorithm() {
        if (solution == null) {
            return "No solution found - cannot analyze algorithm.";
        }
```

```
        StringBuilder analysis = new StringBuilder();
        analysis.append("A* Analysis with
").append(getHeuristicName()).append(":\n");

        // Calculate f, g, h values for each step
        List<Integer> gValues = new ArrayList<>();
        List<Integer> hValues = new ArrayList<>();
        List<Integer> fValues = new ArrayList<>();

        for (int i = 0; i < solution.size(); i++) {
            int g = i;
            int h = calculateHeuristic(solution.get(i));
            int f = g + h;
            gValues.add(g);
            hValues.add(h);
            fValues.add(f);
        }

        // Analyze heuristic behavior
        analysis.append("- Initial f=").append(fValues.get(0)).append(" (g=0,
h=").append(hValues.get(0)).append(")\n");
        analysis.append("- Final f=").append(fValues.get(fValues.size() - 1))
                .append(" (g=").append(gValues.get(gValues.size() - 1))
                .append(", h=").append(hValues.get(hValues.size() -
1)).append(")\n");

        // Check if heuristic is consistent
        boolean consistent = true;
        for (int i = 1; i < solution.size(); i++) {
            if (hValues.get(i - 1) > hValues.get(i) + 1) {
                consistent = false;
                break;
            }
        }

        analysis.append("- Heuristic consistent: ").append(consistent ? "Yes" :
"No").append("\n");
        analysis.append("- Heuristic admissible:
").append(isHeuristicAdmissible() ? "Yes" : "No").append("\n");
        analysis.append("- Solution optimal: ").append(isHeuristicAdmissible() ?
"Yes" : "Possibly No").append("\n");

        // Efficiency analysis
        double avgBranchingFactor = (double) nodesVisited / solution.size();
        analysis.append("- Average effective branching factor:
").append(String.format("%.2f", avgBranchingFactor)).append("\n");

        return analysis.toString();
    }

    public String compareWithUCS() {
        StringBuilder comparison = new StringBuilder();
        comparison.append("A* vs UCS Theoretical Comparison:\n");

        if (isHeuristicAdmissible()) {
```

```
            comparison.append("- A* should be more efficient than UCS (fewer
nodes expanded)\n");
            comparison.append("- Both find optimal solution\n");
            comparison.append("- A* uses heuristic to guide search toward
goal\n");
        } else {
            comparison.append("- A* may be faster but might not find optimal
solution\n");
            comparison.append("- UCS guarantees optimal solution\n");
            comparison.append("- Non-admissible heuristic can mislead A*\n");
        }

        comparison.append("- A* memory usage similar to UCS\n");
        comparison.append("- A* has overhead of heuristic calculation\n");

        return comparison.toString();
    }

    public boolean guaranteesOptimalSolution() {
        return isHeuristicAdmissible();
    }
}
```

4.2 GameAnimation.java

```
package src;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.List;

public class GameAnimation {
    private GamePanel gamePanel;
    private List<RushHourGame> solution;
    private List<String> actions;
    private int currentStep;
    private boolean isPlaying;
    private Timer animationTimer;
    private AnimationListener listener;

    // Animation settings
    private int animationSpeed; // Speed from 1 (slow) to 10 (fast)
    private static final int BASE_DELAY = 2000; // Base delay in milliseconds (2
seconds)
    private static final int MIN_DELAY = 200;   // Minimum delay (0.2 seconds)

    // Animation states
    private enum AnimationState {
        STOPPED, PLAYING, PAUSED, COMPLETED
    }

    private AnimationState currentState;


    public GameAnimation(GamePanel gamePanel) {
```

```
        this.gamePanel = gamePanel;
        this.currentStep = 0;
        this.isPlaying = false;
        this.animationSpeed = 5; // Default middle speed
        this.currentState = AnimationState.STOPPED;

        // Create animation timer
        initializeTimer();
    }


    private void initializeTimer() {
        int delay = calculateDelay();
        animationTimer = new Timer(delay, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (currentState == AnimationState.PLAYING) {
                    nextStep();
                }
            }
        });
    }

    public void setSolution(List<RushHourGame> solution, List<String> actions) {
        if (solution == null || solution.isEmpty()) {
                 throw new IllegalArgumentException("Solution cannot be null or
empty");
        }

        if (actions == null || actions.size() != solution.size() - 1) {
             throw new IllegalArgumentException("Actions list must have one less
element than solution list");
        }

        this.solution = solution;
        this.actions = actions;
        reset();
    }

    public void play() {
        if (solution == null || solution.isEmpty()) {
            return;
        }

               if  (currentState  !=  AnimationState.PLAYING  &&  currentState  !=
AnimationState.COMPLETED) {
            currentState = AnimationState.PLAYING;
            isPlaying = true;

            if (listener != null) {
                listener.onAnimationStarted();
            }

            // Update timer delay based on current speed
            animationTimer.setDelay(calculateDelay());
            animationTimer.start();
```

```
        }
    }

    public void pause() {
        if (currentState == AnimationState.PLAYING) {
            currentState = AnimationState.PAUSED;
            isPlaying = false;
            animationTimer.stop();
        }
    }

    public void stop() {
        currentState = AnimationState.STOPPED;
        isPlaying = false;
        animationTimer.stop();
        reset();
    }

    public void stepForward() {
        if (solution == null || solution.isEmpty()) {
            return;
        }

        if (currentStep < solution.size() - 1) {
            currentStep++;
            updateDisplay();

            if (listener != null) {
                listener.onMoveChanged(currentStep, solution.size() - 1);
            }

            // Check if we've reached the end
            if (currentStep >= solution.size() - 1) {
                // Don't reset - just stop the timer and mark as completed
                currentState = AnimationState.COMPLETED;
                isPlaying = false;
                animationTimer.stop();

                if (listener != null) {
                    listener.onAnimationComplete();
                }
            }
        }
    }

    public void stepBackward() {
        if (solution == null || solution.isEmpty()) {
            return;
        }

        if (currentStep > 0) {
            currentStep--;

            // If we were completed and step back, we're no longer completed
            if (currentState == AnimationState.COMPLETED) {
                currentState = AnimationState.STOPPED;
```

```
        }

        updateDisplay();

        if (listener != null) {
            listener.onMoveChanged(currentStep, solution.size() - 1);
        }
    }
}

public void reset() {
    currentStep = 0;
    currentState = AnimationState.STOPPED;
    isPlaying = false;
    animationTimer.stop();

    updateDisplay();

    if (listener != null && solution != null) {
        listener.onMoveChanged(currentStep, solution.size() - 1);
    }
}

public void setSpeed(int speed) {
    this.animationSpeed = Math.max(1, Math.min(10, speed));

    // Update timer delay if animation is running
    if (animationTimer != null) {
        animationTimer.setDelay(calculateDelay());
    }
}

public int getSpeed() {
    return animationSpeed;
}

public void setAnimationListener(AnimationListener listener) {
    this.listener = listener;
}

public int getCurrentStep() {
    return currentStep;
}

public int getTotalSteps() {
    return solution != null ? solution.size() - 1 : 0;
}

public boolean isPlaying() {
    return isPlaying;
}

public boolean isPaused() {
    return currentState == AnimationState.PAUSED;
}
```

```
    public boolean isCompleted() {
        return currentState == AnimationState.COMPLETED;
    }

    public boolean hasNextStep() {
        return solution != null && currentStep < solution.size() - 1;
    }

    public boolean hasPreviousStep() {
        return solution != null && currentStep > 0;
    }

    public String getCurrentAction() {
        if (actions != null && currentStep > 0 && currentStep <= actions.size())
{
            return actions.get(currentStep - 1);
        }
        return null;
    }

    public void jumpToStep(int step) {
        if (solution == null || solution.isEmpty()) {
            return;
        }

        step = Math.max(0, Math.min(step, solution.size() - 1));
        currentStep = step;

        // Update state based on the step we jumped to
        if (step >= solution.size() - 1) {
            currentState = AnimationState.COMPLETED;
            isPlaying = false;
            animationTimer.stop();
        } else if (currentState == AnimationState.COMPLETED) {
            currentState = AnimationState.STOPPED;
        }

        updateDisplay();

        if (listener != null) {
            listener.onMoveChanged(currentStep, solution.size() - 1);
        }
    }

    private int calculateDelay() {
        // Convert speed (1-10) to delay (2000-200 ms)
        // Speed 1 = 2000ms, Speed 10 = 200ms
             int delay = BASE_DELAY - ((animationSpeed - 1) * (BASE_DELAY -
MIN_DELAY) / 9);
        return Math.max(MIN_DELAY, delay);
    }

    private void nextStep() {
        if (hasNextStep()) {
            stepForward();
        } else {
```

```
            // Animation complete - don't reset, just stop and mark as completed
            currentState = AnimationState.COMPLETED;
            isPlaying = false;
            animationTimer.stop();

            if (listener != null) {
                listener.onAnimationComplete();
            }
        }
    }

    private void updateDisplay() {
        if (solution == null || solution.isEmpty()) {
            return;
        }

        // Set the current game state
        RushHourGame currentGameState = solution.get(currentStep);
        gamePanel.setGame(currentGameState);

        // Highlight the piece that moved (if any)
        if (currentStep > 0 && actions != null && currentStep <= actions.size())
{
            String action = actions.get(currentStep - 1);
            gamePanel.setLastAction(action);
        } else {
            gamePanel.setLastAction(null);
        }

        // Force repaint
        gamePanel.repaint();
    }

    public String getAnimationInfo() {
        if (solution == null) {
            return "No solution loaded";
        }

        StringBuilder info = new StringBuilder();
                                                                info.append("Step:
").append(currentStep).append("/").append(solution.size() - 1);
        info.append(" | State: ").append(currentState);
        info.append(" | Speed: ").append(animationSpeed).append("/10");

        if (currentStep > 0 && actions != null && currentStep <= actions.size())
{
                info.append(" | Last Action: ").append(actions.get(currentStep -
1));
        }

        return info.toString();
    }

    public String getProgressBar(int width) {
        if (solution == null || solution.isEmpty()) {
            return "[" + " ".repeat(width) + "]";
```

```
        }

          int progress = (int) ((double) currentStep / (solution.size() - 1) *
width);
        progress = Math.max(0, Math.min(progress, width));

        StringBuilder bar = new StringBuilder("[");
        for (int i = 0; i < width; i++) {
            if (i < progress) {
                bar.append("=");
            } else if (i == progress) {
                bar.append(">");
            } else {
                bar.append(" ");
            }
        }
        bar.append("]");

        return bar.toString();
    }

    public String getStepDetails() {
        if (solution == null || currentStep >= solution.size()) {
            return "No step information available";
        }

        StringBuilder details = new StringBuilder();
        details.append("Step ").append(currentStep).append(":\n");

        if (currentStep == 0) {
            details.append("Initial state");
        } else {
            details.append("Action: ").append(actions.get(currentStep - 1));
        }

        return details.toString();
    }

    public void dispose() {
        if (animationTimer != null) {
            animationTimer.stop();
            animationTimer = null;
        }
        currentState = AnimationState.STOPPED;
        isPlaying = false;
    }
}
```

4.3 GamePanel.java

```
package src;

import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
```

```java
import java.util.List;
import java.util.Map;


public class GamePanel extends JPanel {
    private RushHourGame game;
    private int cellSize;
    private int boardMargin;
    private String lastAction;
    private char highlightedPiece;

    // Colors for different elements
    private static final Color BACKGROUND_COLOR = Color.WHITE;
    private static final Color BOARD_COLOR = Color.LIGHT_GRAY;
    private static final Color CELL_COLOR = Color.WHITE;
    private static final Color PRIMARY_PIECE_COLOR = Color.RED;
    private static final Color EXIT_COLOR = Color.GREEN;
    private static final Color MOVED_PIECE_COLOR = Color.YELLOW;
    private static final Color OTHER_PIECE_COLOR = Color.BLUE;
    private static final Color GRID_COLOR = Color.GRAY;
    private static final Color WALL_COLOR = Color.DARK_GRAY;

    // Piece type indicators
    private enum PieceType {
        PRIMARY, EXIT, MOVED, OTHER, EMPTY
    }


    public GamePanel() {
        this.game = null;
        this.boardMargin = 50;
        this.lastAction = null;
        this.highlightedPiece = '.';

        // Set preferred size
        setPreferredSize(new Dimension(600, 600));
        setBackground(BACKGROUND_COLOR);

        // Add mouse listener for interaction (optional - for debugging/testing)
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                if (game != null) {
                    Point cell = getCellFromPoint(e.getPoint());
                    if (cell != null) {
                        onCellClicked(cell.x, cell.y);
                    }
                }
            }
        });
    }


    public void setGame(RushHourGame game) {
        this.game = game;
        this.lastAction = null;
```

```
        this.highlightedPiece = '.';
        repaint();
    }


    public void setLastAction(String action) {
        this.lastAction = action;
        this.highlightedPiece = '.';

        if (action != null && !action.equals("Initial")) {
            String[] parts = action.split("-");
            if (parts.length > 0) {
                this.highlightedPiece = parts[0].charAt(0);
            }
        }
        repaint();
    }

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

        if (game == null) {
            drawEmptyBoard(g);
            return;
        }

        Graphics2D g2d = (Graphics2D) g.create();
                                g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

        calculateDimensions();

        // Draw board background and borders
        drawBoardBackground(g2d);
        drawBoardWalls(g2d);

        // Draw grid and cells
        drawGrid(g2d);
        drawCells(g2d);

        // Draw exit indicator
        drawExit(g2d);

        // Draw game information
        drawGameInfo(g2d);

        g2d.dispose();
    }

    private void drawEmptyBoard(Graphics g) {
        g.setColor(Color.GRAY);
        g.setFont(new Font("Arial", Font.BOLD, 24));
        FontMetrics fm = g.getFontMetrics();
        String text = "Load a puzzle to begin";
        int x = (getWidth() - fm.stringWidth(text)) / 2;
```

```
        int y = getHeight() / 2;
        g.drawString(text, x, y);
    }

    private void calculateDimensions() {
        if (game == null) return;

        int availableWidth = getWidth() - (2 * boardMargin);
          int availableHeight = getHeight() - (2 * boardMargin) - 100; // 100px
for info area

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

        cellSize = Math.min(availableWidth / cols, availableHeight / rows);
        cellSize = Math.max(cellSize, 20); // Minimum cell size
    }

    private void drawBoardBackground(Graphics2D g2d) {
        int rows = game.getRows();
        int cols = game.getCols();

        // Calculate board position
        int boardWidth = cols * cellSize;
        int boardHeight = rows * cellSize;
        int boardX = (getWidth() - boardWidth) / 2;
        int boardY = boardMargin;

        // Draw board background
        g2d.setColor(BOARD_COLOR);
        g2d.fillRect(boardX - 5, boardY - 5, boardWidth + 10, boardHeight + 10);
    }

    private void drawBoardWalls(Graphics2D g2d) {
        int rows = game.getRows();
        int cols = game.getCols();

        int boardWidth = cols * cellSize;
        int boardHeight = rows * cellSize;
        int boardX = (getWidth() - boardWidth) / 2;
        int boardY = boardMargin;

        g2d.setColor(WALL_COLOR);
        g2d.setStroke(new BasicStroke(4));

        // Draw walls - but leave gap for exit
        int exitRow = game.getExitRow();
        int exitCol = game.getExitCol();

        // Top wall
        g2d.drawLine(boardX, boardY, boardX + boardWidth, boardY);

        // Bottom wall
        g2d.drawLine(boardX, boardY + boardHeight, boardX + boardWidth, boardY +
boardHeight);
```

```
        // Left wall
        g2d.drawLine(boardX, boardY, boardX, boardY + boardHeight);

        // Right wall - with potential gap for exit
        if (exitCol == cols && exitRow >= 0 && exitRow < rows) {
            // Gap for right exit
            int exitY = boardY + exitRow * cellSize;
                g2d.drawLine(boardX + boardWidth, boardY, boardX + boardWidth,
exitY);
                 g2d.drawLine(boardX + boardWidth, exitY + cellSize, boardX +
boardWidth, boardY + boardHeight);
        } else {
            // Full right wall
                g2d.drawLine(boardX + boardWidth, boardY, boardX + boardWidth,
boardY + boardHeight);
        }

        // Draw other exit walls if needed (top, bottom, left)
        // This extends the current implementation to support exits on all sides
    }

    private void drawGrid(Graphics2D g2d) {
        int rows = game.getRows();
        int cols = game.getCols();

        int boardWidth = cols * cellSize;
        int boardHeight = rows * cellSize;
        int boardX = (getWidth() - boardWidth) / 2;
        int boardY = boardMargin;

        g2d.setColor(GRID_COLOR);
        g2d.setStroke(new BasicStroke(1));

        // Vertical lines
        for (int i = 0; i <= cols; i++) {
            int x = boardX + i * cellSize;
            g2d.drawLine(x, boardY, x, boardY + boardHeight);
        }

        // Horizontal lines
        for (int i = 0; i <= rows; i++) {
            int y = boardY + i * cellSize;
            g2d.drawLine(boardX, y, boardX + boardWidth, y);
        }
    }

    private void drawCells(Graphics2D g2d) {
        char[][] board = game.getBoard();
        int rows = game.getRows();
        int cols = game.getCols();

        int boardWidth = cols * cellSize;
        int boardX = (getWidth() - boardWidth) / 2;
        int boardY = boardMargin;

        for (int i = 0; i < rows; i++) {
```

```
            for (int j = 0; j < cols; j++) {
                char cell = board[i][j];
                int x = boardX + j * cellSize;
                int y = boardY + i * cellSize;

                // Determine piece type
                PieceType type = getPieceType(cell);

                // Draw cell background
                g2d.setColor(CELL_COLOR);
                g2d.fillRect(x + 1, y + 1, cellSize - 2, cellSize - 2);

                // Draw piece
                if (cell != '.') {
                    drawPiece(g2d, x, y, cell, type);
                }
            }
        }
    }

    private PieceType getPieceType(char piece) {
        if (piece == '.') {
            return PieceType.EMPTY;
        } else if (piece == game.getPrimaryPiece()) {
            return PieceType.PRIMARY;
        } else if (piece == game.getExit()) {
            return PieceType.EXIT;
        } else if (piece == highlightedPiece) {
            return PieceType.MOVED;
        } else {
            return PieceType.OTHER;
        }
    }

     private void drawPiece(Graphics2D g2d, int x, int y, char piece, PieceType
type) {
        Color pieceColor;

        switch (type) {
            case PRIMARY:
                pieceColor = PRIMARY_PIECE_COLOR;
                break;
            case EXIT:
                pieceColor = EXIT_COLOR;
                break;
            case MOVED:
                pieceColor = MOVED_PIECE_COLOR;
                break;
            case OTHER:
                pieceColor = OTHER_PIECE_COLOR;
                break;
            default:
                return; // Don't draw empty cells
        }

        // Draw piece with rounded corners
```

```
        g2d.setColor(pieceColor);
        g2d.fillRoundRect(x + 3, y + 3, cellSize - 6, cellSize - 6, 8, 8);

        // Draw piece outline
        g2d.setColor(pieceColor.darker());
        g2d.setStroke(new BasicStroke(2));
        g2d.drawRoundRect(x + 3, y + 3, cellSize - 6, cellSize - 6, 8, 8);

        // Draw piece letter
        g2d.setColor(Color.WHITE);
        Font font = new Font("Arial", Font.BOLD, cellSize / 3);
        g2d.setFont(font);
        FontMetrics fm = g2d.getFontMetrics();
        int textX = x + (cellSize - fm.charWidth(piece)) / 2;
        int textY = y + (cellSize + fm.getAscent()) / 2;
        g2d.drawString(String.valueOf(piece), textX, textY);
    }

    private void drawExit(Graphics2D g2d) {
        int exitRow = game.getExitRow();
        int exitCol = game.getExitCol();
        int rows = game.getRows();
        int cols = game.getCols();

        int boardWidth = cols * cellSize;
        int boardHeight = rows * cellSize;
        int boardX = (getWidth() - boardWidth) / 2;
        int boardY = boardMargin;

        g2d.setColor(EXIT_COLOR);
        g2d.setStroke(new BasicStroke(6));

        if (exitCol == cols && exitRow >= 0 && exitRow < rows) {
            // Right exit
            int y = boardY + exitRow * cellSize + cellSize / 2;
            int x = boardX + cols * cellSize;

            // Draw arrow pointing right
            g2d.drawLine(x, y, x + 30, y);
            g2d.drawLine(x + 20, y - 10, x + 30, y);
            g2d.drawLine(x + 20, y + 10, x + 30, y);

            // Draw EXIT text
            g2d.setColor(EXIT_COLOR);
            g2d.setFont(new Font("Arial", Font.BOLD, 14));
            g2d.drawString("EXIT", x + 35, y + 5);
        } else if (exitCol == -1 && exitRow >= 0 && exitRow < rows) {
            // Left exit
            int y = boardY + exitRow * cellSize + cellSize / 2;
            int x = boardX;

            // Draw arrow pointing left
            g2d.drawLine(x, y, x - 30, y);
            g2d.drawLine(x - 20, y - 10, x - 30, y);
            g2d.drawLine(x - 20, y + 10, x - 30, y);
```

```
            // Draw EXIT text
            g2d.setColor(EXIT_COLOR);
            g2d.setFont(new Font("Arial", Font.BOLD, 14));
            g2d.drawString("EXIT", x - 65, y + 5);
        } else if (exitRow == -1 && exitCol >= 0 && exitCol < cols) {
            // Top exit
            int x = boardX + exitCol * cellSize + cellSize / 2;
            int y = boardY;

            // Draw arrow pointing up
            g2d.drawLine(x, y, x, y - 30);
            g2d.drawLine(x - 10, y - 20, x, y - 30);
            g2d.drawLine(x + 10, y - 20, x, y - 30);

            // Draw EXIT text
            g2d.setColor(EXIT_COLOR);
            g2d.setFont(new Font("Arial", Font.BOLD, 14));
            g2d.drawString("EXIT", x - 20, y - 35);
        } else if (exitRow == rows && exitCol >= 0 && exitCol < cols) {
            // Bottom exit
            int x = boardX + exitCol * cellSize + cellSize / 2;
            int y = boardY + rows * cellSize;

            // Draw arrow pointing down
            g2d.drawLine(x, y, x, y + 30);
            g2d.drawLine(x - 10, y + 20, x, y + 30);
            g2d.drawLine(x + 10, y + 20, x, y + 30);

            // Draw EXIT text
            g2d.setColor(EXIT_COLOR);
            g2d.setFont(new Font("Arial", Font.BOLD, 14));
            g2d.drawString("EXIT", x - 20, y + 45);
        }
    }

    private void drawGameInfo(Graphics2D g2d) {
        if (game == null) return;

        int y = getHeight() - 60;
        g2d.setColor(Color.BLACK);
        g2d.setFont(new Font("Arial", Font.PLAIN, 14));

        // Game info
        String info = String.format("Board: %dx%d | Pieces: %d | Exit: (%d,%d)",
            game.getRows(), game.getCols(),
            game.getPieces().size(),
            game.getExitRow(), game.getExitCol());
        g2d.drawString(info, 10, y);

        // Last action info
        if (lastAction != null && !lastAction.equals("Initial")) {
            y += 20;
            g2d.drawString("Last Action: " + lastAction, 10, y);
        }

        // Legend
```

```
        y += 20;
        drawLegend(g2d, 10, y);
    }

    private void drawLegend(Graphics2D g2d, int x, int y) {
        int legendSize = 15;
        int spacing = 100;

        // Primary piece
        g2d.setColor(PRIMARY_PIECE_COLOR);
        g2d.fillRect(x, y, legendSize, legendSize);
        g2d.setColor(Color.BLACK);
        g2d.drawString("Primary (P)", x + legendSize + 5, y + 12);

        // Other pieces
        x += spacing;
        g2d.setColor(OTHER_PIECE_COLOR);
        g2d.fillRect(x, y, legendSize, legendSize);
        g2d.setColor(Color.BLACK);
        g2d.drawString("Other Pieces", x + legendSize + 5, y + 12);

        // Moved piece
        x += spacing;
        g2d.setColor(MOVED_PIECE_COLOR);
        g2d.fillRect(x, y, legendSize, legendSize);
        g2d.setColor(Color.BLACK);
        g2d.drawString("Last Moved", x + legendSize + 5, y + 12);

        // Exit
        x += spacing;
        g2d.setColor(EXIT_COLOR);
        g2d.fillRect(x, y, legendSize, legendSize);
        g2d.setColor(Color.BLACK);
        g2d.drawString("Exit", x + legendSize + 5, y + 12);
    }

    private Point getCellFromPoint(Point point) {
        if (game == null) return null;

        int rows = game.getRows();
        int cols = game.getCols();
        int boardWidth = cols * cellSize;
        int boardHeight = rows * cellSize;
        int boardX = (getWidth() - boardWidth) / 2;
        int boardY = boardMargin;

        int x = point.x - boardX;
        int y = point.y - boardY;

        if (x >= 0 && x < boardWidth && y >= 0 && y < boardHeight) {
            int col = x / cellSize;
            int row = y / cellSize;
            return new Point(row, col);
        }

        return null;
```

```
    }

    private void onCellClicked(int row, int col) {
        if (game == null) return;

        char[][] board = game.getBoard();
        if (row >= 0 && row < board.length && col >= 0 && col < board[0].length)
{
            char cell = board[row][col];
              System.out.println("Clicked cell (" + row + ", " + col + "): " +
cell);
        }
    }

    public int getCellSize() {
        return cellSize;
    }

    public Point getBoardOffset() {
        if (game == null) return new Point(0, 0);

        int cols = game.getCols();
        int boardWidth = cols * cellSize;
        int boardX = (getWidth() - boardWidth) / 2;
        int boardY = boardMargin;

        return new Point(boardX, boardY);
    }

    public void recalculateDimensions() {
        calculateDimensions();
        repaint();
    }
}
```

4.4 GBFS.java

```
package src;

import java.util.*;

public class GBFS extends SearchAlgorithm {
    private int heuristicType;

    // Heuristic types
    public static final int HEURISTIC_MANHATTAN = 1;
    public static final int HEURISTIC_BLOCKING_PIECES = 2;

    public GBFS(int heuristicType) {
        this.heuristicType = heuristicType;
    }

    @Override
    public List<RushHourGame> solve(RushHourGame initial) {
        resetCounters();
        long startTime = System.currentTimeMillis();
```

```
        // Priority queue ordered by heuristic value (h only)
                PriorityQueue<SearchNode>  frontier  =  new  PriorityQueue<>(new
Comparator<SearchNode>() {
            @Override
            public int compare(SearchNode n1, SearchNode n2) {
                if (n1.h != n2.h) {
                    return Integer.compare(n1.h, n2.h);
                }
                        //  Tie-breaking:  prefer  newer  nodes  (LIFO  for  better
performance)
                return 0;
            }
        });

        Set<RushHourGame> explored = new HashSet<>();
         Set<RushHourGame> frontierStates = new HashSet<>(); // Track states in
frontier

        // Calculate initial heuristic
        int h = calculateHeuristic(initial);
        frontier.add(new SearchNode(initial, null, 0, h, "Initial"));
        frontierStates.add(initial);

        while (!frontier.isEmpty()) {
            SearchNode node = frontier.poll();
            frontierStates.remove(node.state);
            nodesVisited++;

            // Check if goal state reached
            if (node.state.isGoalState()) {
                executionTime = System.currentTimeMillis() - startTime;
                solution = buildSolution(node);
                return solution;
            }

            explored.add(node.state);

            // Expand current node
            for (RushHourGame nextState : node.state.getNextStates()) {
                                        if   (!explored.contains(nextState)   &&
!frontierStates.contains(nextState)) {
                    // Calculate heuristic for new state
                    h = calculateHeuristic(nextState);
                    String action = getAction(node.state, nextState);

                    // GBFS doesn't track g value, but we set it for consistency
                     frontier.add(new SearchNode(nextState, node, node.g + 1, h,
action));
                    frontierStates.add(nextState);
                }
            }
        }

        executionTime = System.currentTimeMillis() - startTime;
        return null; // No solution found
```

```
    }

    private int calculateHeuristic(RushHourGame state) {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return state.getHeuristic1();
            case HEURISTIC_BLOCKING_PIECES:
                return state.getHeuristic2();
            default:
                return state.getHeuristic1();
        }
    }

    @Override
    public String getAlgorithmName() {
        return "Greedy Best First Search (GBFS)";
    }

    @Override
    public String getAlgorithmDescription() {
        String heuristicName = getHeuristicName();
        return "GBFS uses only the heuristic function to guide search toward the
goal. " +
                "Current heuristic: " + heuristicName + ". " +
                "It's fast but doesn't guarantee optimal solution.";
    }

    public String getHeuristicName() {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return "Manhattan Distance (H1)";
            case HEURISTIC_BLOCKING_PIECES:
                return "Blocking Pieces + Distance (H2)";
            default:
                return "Unknown";
        }
    }

    public int getHeuristicType() {
        return heuristicType;
    }

    public void setHeuristicType(int heuristicType) {
        this.heuristicType = heuristicType;
    }

    public Map<String, Object> getStatistics() {
        Map<String, Object> stats = new HashMap<>();
        stats.put("algorithm", getAlgorithmName());
        stats.put("heuristic", getHeuristicName());
        stats.put("heuristic_type", heuristicType);
        stats.put("nodes_visited", nodesVisited);
        stats.put("execution_time", executionTime);
          stats.put("solution_length", solution != null ? solution.size() - 1 :
-1);
        stats.put("optimal", false); // GBFS doesn't guarantee optimal solution
```

```
        return stats;
    }

    @Override
    public void printSolution() {
        System.out.println("=== " + getAlgorithmName() + " Results ===");
        System.out.println(getAlgorithmDescription());
        System.out.println();

        if (solution == null || solution.isEmpty()) {
            System.out.println("No solution found!");
            System.out.println("Nodes visited: " + nodesVisited);
            System.out.println("Execution time: " + executionTime + " ms");
            System.out.println("Heuristic used: " + getHeuristicName());
            return;
        }

        System.out.println("Solution found!");
        System.out.println("Nodes visited: " + nodesVisited);
        System.out.println("Execution time: " + executionTime + " ms");
           System.out.println("Solution length: " + (solution.size() - 1) + "
moves");
        System.out.println("Heuristic used: " + getHeuristicName());
         System.out.println("Status: NON-OPTIMAL (GBFS doesn't guarantee optimal
solution)");
        System.out.println();

        // Print solution path
        System.out.println("Papan Awal");
        printColoredBoard(solution.get(0), null);
        System.out.println();

        // Print each move with heuristic values
        for (int i = 1; i < solution.size(); i++) {
            int hValue = calculateHeuristic(solution.get(i));
              System.out.println("Gerakan " + i + ": " + solutionActions.get(i -
1) +
                     " (h=" + hValue + ")");
            printColoredBoard(solution.get(i), solutionActions.get(i - 1));
            System.out.println();
        }
    }

    private void printColoredBoard(RushHourGame game, String action) {
        char[][] board = game.getBoard();
        int rows = game.getRows();
        int cols = game.getCols();

        // ANSI color codes
        String RESET = "\u001B[0m";
        String RED = "\u001B[31m";      // Primary piece
        String GREEN = "\u001B[32m";    // Exit
        String YELLOW = "\u001B[33m";   // Moved piece
        String BLUE = "\u001B[34m";     // Other pieces

        char movedPiece = '.';
```

```
        if (action != null && !action.equals("Initial")) {
            movedPiece = action.charAt(0);
        }

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                char c = board[i][j];

                if (c == game.getPrimaryPiece()) {
                    System.out.print(RED + c + RESET);
                } else if (c == game.getExit()) {
                    System.out.print(GREEN + c + RESET);
                } else if (c == movedPiece && c != '.') {
                    System.out.print(YELLOW + c + RESET);
                } else if (c != '.' && c != game.getExit()) {
                    System.out.print(BLUE + c + RESET);
                } else {
                    System.out.print(c);
                }
            }
            System.out.println();
        }
    }

    public String analyzeHeuristic() {
        if (solution == null) {
            return "No solution found - cannot analyze heuristic.";
        }

        StringBuilder analysis = new StringBuilder();
                                    analysis.append("Heuristic    Analysis    for
").append(getHeuristicName()).append(":\n");

        // Calculate average heuristic value throughout solution
        int totalH = 0;
        for (RushHourGame state : solution) {
            totalH += calculateHeuristic(state);
        }
        double avgH = (double) totalH / solution.size();

                                 analysis.append("-    Average    heuristic    value:
").append(String.format("%.2f", avgH)).append("\n");
                                      analysis.append("-     Initial     heuristic:
").append(calculateHeuristic(solution.get(0))).append("\n");
                                        analysis.append("-     Final     heuristic:
").append(calculateHeuristic(solution.get(solution.size() - 1))).append("\n");

        // Check if heuristic generally decreases
        int decreases = 0;
        int increases = 0;
        for (int i = 1; i < solution.size(); i++) {
            int prevH = calculateHeuristic(solution.get(i - 1));
            int currH = calculateHeuristic(solution.get(i));
            if (currH < prevH) decreases++;
            else if (currH > prevH) increases++;
        }
```

```
          analysis.append("- Heuristic decreases: ").append(decreases).append(" 
times\n");
          analysis.append("- Heuristic increases: ").append(increases).append(" 
times\n");

        if (decreases > increases) {
              analysis.append("- The heuristic generally guides toward the goal 
effectively.\n");
        } else {
               analysis.append("- The heuristic may not be very informative for 
this problem.\n");
        }

        return analysis.toString();
    }

    public boolean isHeuristicAdmissible() {
          // Heuristic 1 (Manhattan distance) is generally admissible for Rush 
Hour
               // Heuristic 2 might overestimate in some cases, making it 
non-admissible
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return true; // Manhattan distance is admissible
            case HEURISTIC_BLOCKING_PIECES:
                   return false; // May overestimate due to the blocking pieces 
factor
            default:
                return false;
        }
    }


    public boolean guaranteesOptimalSolution() {
        return false;
    }
}
```

4.5 IDAStar.java

```
package src;

import java.util.*;


public class IDAStar extends SearchAlgorithm {
    private int heuristicType;
    private int threshold;
    private int nextThreshold;

    // Heuristic types
    public static final int HEURISTIC_MANHATTAN = 1;
    public static final int HEURISTIC_BLOCKING_PIECES = 2;

    public IDAStar(int heuristicType) {
```

```
        this.heuristicType = heuristicType;
    }

    @Override
    public List<RushHourGame> solve(RushHourGame initial) {
        resetCounters();
        long startTime = System.currentTimeMillis();

        // Initialize threshold with initial heuristic value
        threshold = calculateHeuristic(initial);

        while (threshold != Integer.MAX_VALUE) {
            nextThreshold = Integer.MAX_VALUE;
                      SearchNode  result  =  depthLimitedSearch(initial,  0,  new
ArrayList<>());

            if (result != null) {
                // Solution found
                executionTime = System.currentTimeMillis() - startTime;
                solution = buildSolution(result);
                return solution;
            }

            threshold = nextThreshold;
        }

        executionTime = System.currentTimeMillis() - startTime;
        return null; // No solution found
    }

          private  SearchNode  depthLimitedSearch(RushHourGame  state,  int  g,
List<RushHourGame> path) {
        nodesVisited++;

        int h = calculateHeuristic(state);
        int f = g + h;

        // If f exceeds threshold, update next threshold and prune
        if (f > threshold) {
            nextThreshold = Math.min(nextThreshold, f);
            return null;
        }

        // Check if goal reached
        if (state.isGoalState()) {
            // Create solution node
            SearchNode solutionNode = new SearchNode(state, null, g, h, "Goal");

            // Build parent nodes for solution path
            SearchNode current = solutionNode;
            for (int i = path.size() - 1; i >= 0; i--) {
                RushHourGame prevState = path.get(i);
                 String action = i > 0 ? getAction(path.get(i - 1), prevState) :
"Initial";
                       SearchNode  parent  =  new  SearchNode(prevState,  null,  i,
calculateHeuristic(prevState), action);
```

```
                current.parent = parent;
                current = parent;
            }

            return solutionNode;
        }

        // Expand current node
        List<RushHourGame> nextStates = state.getNextStates();

        // Sort by f-value for better pruning (optional optimization)
        nextStates.sort((s1, s2) -> {
            int f1 = (g + 1) + calculateHeuristic(s1);
            int f2 = (g + 1) + calculateHeuristic(s2);
            return Integer.compare(f1, f2);
        });

        for (RushHourGame nextState : nextStates) {
            // Check for cycles in path
            if (!path.contains(nextState)) {
                path.add(state); // Add current state to path
                SearchNode result = depthLimitedSearch(nextState, g + 1, path);
                path.remove(path.size() - 1); // Remove state from path

                if (result != null) {
                    // Update parent link
                    if (result.parent == null && result.state.equals(nextState))
{
                        String action = getAction(state, nextState);
                              result.parent = new SearchNode(state, null, g, h,
action);
                    }
                    return result;
                }
            }
        }

        return null; // No solution found at this threshold
    }

    private int calculateHeuristic(RushHourGame state) {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return state.getHeuristic1();
            case HEURISTIC_BLOCKING_PIECES:
                return state.getHeuristic2();
            default:
                return state.getHeuristic1();
        }
    }

    @Override
    public String getAlgorithmName() {
        return "Iterative Deepening A* (IDA*)";
    }
```

```
    @Override
    public String getAlgorithmDescription() {
        String heuristicName = getHeuristicName();
         return "IDA* combines iterative deepening with A* heuristic guidance. "
+
                  "Uses minimal memory (space complexity O(d)) while maintaining
A* optimality. " +
                "Current heuristic: " + heuristicName + ".";
    }

    public String getHeuristicName() {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return "Manhattan Distance (H1)";
            case HEURISTIC_BLOCKING_PIECES:
                return "Blocking Pieces + Distance (H2)";
            default:
                return "Unknown";
        }
    }

    public int getHeuristicType() {
        return heuristicType;
    }

    public void setHeuristicType(int heuristicType) {
        this.heuristicType = heuristicType;
    }

    public boolean isHeuristicAdmissible() {
        switch (heuristicType) {
            case HEURISTIC_MANHATTAN:
                return true; // Manhattan distance is admissible
            case HEURISTIC_BLOCKING_PIECES:
                return false; // May overestimate
            default:
                return false;
        }
    }

    public Map<String, Object> getStatistics() {
        Map<String, Object> stats = new HashMap<>();
        stats.put("algorithm", getAlgorithmName());
        stats.put("heuristic", getHeuristicName());
        stats.put("heuristic_type", heuristicType);
        stats.put("nodes_visited", nodesVisited);
        stats.put("execution_time", executionTime);
          stats.put("solution_length", solution != null ? solution.size() - 1 :
-1);
        stats.put("optimal", isHeuristicAdmissible());
        stats.put("space_complexity", "O(d)"); // Linear in solution depth
        return stats;
    }

    @Override
    public void printSolution() {
```

```
        System.out.println("=== " + getAlgorithmName() + " Results ===");
        System.out.println(getAlgorithmDescription());
        System.out.println();

        if (solution == null || solution.isEmpty()) {
            System.out.println("No solution found!");
            System.out.println("Nodes visited: " + nodesVisited);
            System.out.println("Execution time: " + executionTime + " ms");
            System.out.println("Heuristic used: " + getHeuristicName());
                                System.out.println("Heuristic   admissible:   "   +
isHeuristicAdmissible());
            return;
        }

        System.out.println("Solution found!");
        System.out.println("Nodes visited: " + nodesVisited);
        System.out.println("Execution time: " + executionTime + " ms");
           System.out.println("Solution  length:  "  +  (solution.size()  -  1)  +  "
moves");
        System.out.println("Heuristic used: " + getHeuristicName());
        System.out.println("Heuristic admissible: " + isHeuristicAdmissible());
         System.out.println("Status: " + (isHeuristicAdmissible() ? "OPTIMAL" :
"NON-OPTIMAL"));
        System.out.println("Space Complexity: O(d) - Linear in solution depth");
        System.out.println();

        // Print solution path with threshold information
        System.out.println("Papan Awal");
        printColoredBoard(solution.get(0), null);
        int h0 = calculateHeuristic(solution.get(0));
        System.out.println("(g=0, h=" + h0 + ", f=" + h0 + ")");
        System.out.println();

        // Print each move
        for (int i = 1; i < solution.size(); i++) {
            int g = i;
            int h = calculateHeuristic(solution.get(i));
            int f = g + h;
              System.out.println("Gerakan " + i + ": " + solutionActions.get(i -
1) +
                    " (g=" + g + ", h=" + h + ", f=" + f + ")");
            printColoredBoard(solution.get(i), solutionActions.get(i - 1));
            System.out.println();
        }
    }

    private void printColoredBoard(RushHourGame game, String action) {
        char[][] board = game.getBoard();
        int rows = game.getRows();
        int cols = game.getCols();

        // ANSI color codes
        String RESET = "\u001B[0m";
        String RED = "\u001B[31m";      // Primary piece
        String GREEN = "\u001B[32m";    // Exit
        String YELLOW = "\u001B[33m";   // Moved piece
```

```
        String BLUE = "\u001B[34m";     // Other pieces

        char movedPiece = '.';
        if (action != null && !action.equals("Initial")) {
            movedPiece = action.charAt(0);
        }

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                char c = board[i][j];

                if (c == game.getPrimaryPiece()) {
                    System.out.print(RED + c + RESET);
                } else if (c == game.getExit()) {
                    System.out.print(GREEN + c + RESET);
                } else if (c == movedPiece && c != '.') {
                    System.out.print(YELLOW + c + RESET);
                } else if (c != '.' && c != game.getExit()) {
                    System.out.print(BLUE + c + RESET);
                } else {
                    System.out.print(c);
                }
            }
            System.out.println();
        }
    }

    public String analyzeAlgorithm() {
        if (solution == null) {
            return "No solution found - cannot analyze algorithm.";
        }

        StringBuilder analysis = new StringBuilder();
                                        analysis.append("IDA*     Analysis     with
").append(getHeuristicName()).append(":\n");

        // Calculate memory usage advantage
        int solutionDepth = solution.size() - 1;
                                         analysis.append("-     Solution     depth:
").append(solutionDepth).append("\n");
                                       analysis.append("-     Space     complexity:
O(").append(solutionDepth).append(") vs O(b^d) for A*\n");

        // Analyze threshold growth
        analysis.append("- Final threshold: ").append(threshold).append("\n");

        // Check if heuristic is helpful
        int initialH = calculateHeuristic(solution.get(0));
        if (initialH > 0) {
                  analysis.append("-  Heuristic  provides  guidance  (initial  h  =
").append(initialH).append(")\n");
        } else {
            analysis.append("- Heuristic provides minimal guidance\n");
        }

        // Efficiency comparison
```

```
        analysis.append("- Nodes visited: ").append(nodesVisited).append("\n");
                        analysis.append("-  Average  nodes  per  solution  step:
").append(String.format("%.2f",         (double)         nodesVisited         /
solutionDepth)).append("\n");

        // Admissibility check
                                analysis.append("-    Heuristic    admissible:
").append(isHeuristicAdmissible() ? "Yes" : "No").append("\n");
        analysis.append("- Solution optimal: ").append(isHeuristicAdmissible() ?
"Yes" : "Possibly No").append("\n");

        return analysis.toString();
    }

    public String compareWithOtherAlgorithms() {
        StringBuilder comparison = new StringBuilder();
        comparison.append("IDA* vs Other Algorithms:\n");

        comparison.append("\nIDA* vs A*:\n");
        comparison.append("- Space: IDA* uses O(d), A* uses O(b^d)\n");
              comparison.append("-  Time:  IDA*  may  revisit  nodes,  potentially
slower\n");
               comparison.append("-  Optimality:  Both  optimal  with  admissible
heuristic\n");

        comparison.append("\nIDA* vs UCS:\n");
        if (isHeuristicAdmissible()) {
                        comparison.append("-  IDA*  should  be  faster  (guided  by
heuristic)\n");
            comparison.append("- Both find optimal solution\n");
        } else {
            comparison.append("- UCS guarantees optimality, IDA* may not\n");
        }
        comparison.append("- IDA* uses less memory than UCS\n");

        comparison.append("\nIDA* vs GBFS:\n");
               comparison.append("-  IDA*  guarantees  optimality  (if  heuristic
admissible)\n");
        comparison.append("- GBFS is typically faster but non-optimal\n");
        comparison.append("- IDA* uses less memory than GBFS\n");

        comparison.append("\nIDA* Advantages:\n");
        comparison.append("- Minimal memory usage\n");
        comparison.append("- Optimal solution (with admissible heuristic)\n");
        comparison.append("- No need to store explored states\n");

        comparison.append("\nIDA* Disadvantages:\n");
        comparison.append("- May revisit the same states multiple times\n");
        comparison.append("- Can be slower than A* for easier problems\n");
               comparison.append("-  Performance  depends  heavily  on  heuristic
quality\n");

        return comparison.toString();
    }

    public boolean guaranteesOptimalSolution() {
```

```
        return isHeuristicAdmissible();
    }
}
```

4.6 RushHourGame.java

```
package src;

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

public class RushHourGame {
    private int rows, cols;
    private char[][] board;
    private char primaryPiece = 'P';
    private char exit = 'K';
    protected int exitRow, exitCol;
    protected int primaryRow, primaryCol;
    private Map<Character, List<int[]>> pieces;

    public RushHourGame(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        this.board = new char[rows][cols];
        this.pieces = new HashMap<>();
    }

    // Copy constructor for creating new states
    public RushHourGame(RushHourGame other) {
        this.rows = other.rows;
        this.cols = other.cols;
        this.board = new char[rows][cols];
        this.pieces = new HashMap<>();

        // Deep copy board
        for (int i = 0; i < rows; i++) {
            System.arraycopy(other.board[i], 0, this.board[i], 0, cols);
        }

        // Deep copy pieces
         for (Map.Entry<Character, List<int[]>> entry : other.pieces.entrySet())
{
            List<int[]> newPieces = new ArrayList<>();
            for (int[] piece : entry.getValue()) {
                newPieces.add(piece.clone());
            }
            this.pieces.put(entry.getKey(), newPieces);
        }

        this.exitRow = other.exitRow;
        this.exitCol = other.exitCol;
        this.primaryRow = other.primaryRow;
        this.primaryCol = other.primaryCol;
    }
```

```
    public static RushHourGame loadFromFile(String filename) throws IOException
{
        BufferedReader reader = new BufferedReader(new FileReader(filename));

        try {
            // Read dimensions
            String[] dimensions = reader.readLine().split(" ");
            int rows = Integer.parseInt(dimensions[0]);
            int cols = Integer.parseInt(dimensions[1]);

            // Read number of pieces (not including primary piece)
            int numPieces = Integer.parseInt(reader.readLine());

            RushHourGame game = new RushHourGame(rows, cols);

            // Read all lines to find the board and exit
            List<String> allLines = new ArrayList<>();
            String line;
            while ((line = reader.readLine()) != null) {
                allLines.add(line);
            }

            // Check for exit at the top (before the board)
            boolean exitFound = false;
            int startLine = 0;

            // Check if first line is just "K" (exit at top)
            if (allLines.size() > 0 && allLines.get(0).trim().equals("K")) {
                game.exitRow = -1;
                game.exitCol = cols / 2; // Place at center of top border
                exitFound = true;
                startLine = 1;
            }

            // Read the board
            for (int i = 0; i < rows; i++) {
                if (startLine + i >= allLines.size()) {
                    throw new IOException("Not enough lines for board");
                }

                String boardLine = allLines.get(startLine + i);

                // Check for exit on left side
                if (boardLine.startsWith("K")) {
                    game.exitRow = i;
                    game.exitCol = -1;
                    exitFound = true;
                          boardLine = boardLine.substring(1); // Remove K from
beginning
                }

                // Check for exit on right side
                if (boardLine.endsWith("K")) {
                    game.exitRow = i;
                    game.exitCol = cols;
                    exitFound = true;
```

```
                    boardLine = boardLine.substring(0, boardLine.length() - 1);
// Remove K from end
                }

                // Process the board line
                if (boardLine.length() != cols) {
                      throw new IOException("Row " + (i + 1) + " has incorrect
length. Expected " + cols + ", got " + boardLine.length());
                }

                for (int j = 0; j < cols; j++) {
                    char c = boardLine.charAt(j);
                    game.board[i][j] = c;

                    if (c != '.') {
                        if (!game.pieces.containsKey(c)) {
                            game.pieces.put(c, new ArrayList<>());
                        }
                        game.pieces.get(c).add(new int[]{i, j});

                        if (c == 'P') {
                            game.primaryRow = i;
                            game.primaryCol = j;
                        }
                    }
                }
            }

            // Check for exit at the bottom (after the board)
            if (startLine + rows < allLines.size()) {
                String bottomLine = allLines.get(startLine + rows);
                if (bottomLine.trim().equals("K")) {
                    game.exitRow = rows;
                    game.exitCol = cols / 2; // Place at center of bottom border
                    exitFound = true;
                }
            }

            if (!exitFound) {
                    throw new IOException("No exit 'K' found outside the board
border");
            }

            return game;
        } finally {
            reader.close();
        }
    }
    public boolean isGoalState() {
        // Check if primary piece can exit
        List<int[]> primaryPositions = pieces.get('P');
        if (primaryPositions == null) return false;

        // Determine orientation of primary piece
        boolean isHorizontal = primaryPositions.size() > 1 &&
                primaryPositions.get(0)[0] == primaryPositions.get(1)[0];
```

```
        if (isHorizontal) {
            // Primary piece is horizontal, exit should be on the same row
            if (exitRow != primaryRow) return false;

            // Check if we can slide the primary piece to the exit
            int minCol = Integer.MAX_VALUE, maxCol = Integer.MIN_VALUE;
            for (int[] pos : primaryPositions) {
                minCol = Math.min(minCol, pos[1]);
                maxCol = Math.max(maxCol, pos[1]);
            }

            // Check if exit is reachable
            if (exitCol == cols) {
                // Exit is at right border
                if (maxCol == cols - 1) {
                    // Primary piece is already at the edge, goal reached
                    return true;
                } else {
                    // Check if path to right edge is clear
                    for (int j = maxCol + 1; j < cols; j++) {
                        if (board[exitRow][j] != '.') {
                            return false;
                        }
                    }
                    return true;
                }
            } else if (exitCol == -1) {
                // Exit is at left edge
                for (int j = exitCol + 1; j < minCol; j++) {
                    if (board[exitRow][j] != '.') return false;
                }
                return true;
             }else if (exitCol > maxCol) {
                // Exit is to the right, check if path is clear
                for (int j = maxCol + 1; j <= exitCol; j++) {
                    if (board[exitRow][j] != '.') {
                        return false;
                    }
                }
                return true;
            } else if (exitCol < minCol) {
                // Exit is to the left, check if path is clear
                for (int j = exitCol; j < minCol; j++) {
                    if (board[exitRow][j] != '.') {
                        return false;
                    }
                }
                return true;
            }
        } else {
            // Primary piece is vertical, exit should be on the same column
            if (exitCol != primaryCol) return false;

            // Check if we can slide the primary piece to the exit
            int minRow = Integer.MAX_VALUE, maxRow = Integer.MIN_VALUE;
```

```
            for (int[] pos : primaryPositions) {
                minRow = Math.min(minRow, pos[0]);
                maxRow = Math.max(maxRow, pos[0]);
            }

            // Check if exit is reachable
            if (exitRow == rows) {
                // Exit is at bottom border
                if (maxRow == rows - 1) {
                    // Primary piece is already at the edge, goal reached
                    return true;
                } else {
                    // Check if path to bottom edge is clear
                    for (int i = maxRow + 1; i < rows; i++) {
                        if (board[i][exitCol] != '.') {
                            return false;
                        }
                    }
                    return true;
                }
            } else if (exitRow == -1) {
                // Exit is at top
                for (int i = exitRow + 1; i < minRow; i++) {
                    if (board[i][exitCol] != '.') return false;
                }
                return true;
            }else if (exitRow > maxRow) {
                // Exit is below, check if path is clear
                for (int i = maxRow + 1; i <= exitRow; i++) {
                    if (board[i][exitCol] != '.') {
                        return false;
                    }
                }
                return true;
            } else if (exitRow < minRow) {
                // Exit is above, check if path is clear
                for (int i = exitRow; i < minRow; i++) {
                    if (board[i][exitCol] != '.') {
                        return false;
                    }
                }
                return true;
            }
        }

        return false;
    }

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

        for (Map.Entry<Character, List<int[]>> entry : pieces.entrySet()) {
            char piece = entry.getKey();
            List<int[]> positions = entry.getValue();

            if (positions.size() < 2) continue;
```

```
            // Determine if piece is horizontal or vertical
            boolean isHorizontal = positions.get(0)[0] == positions.get(1)[0];

            if (isHorizontal) {
                // Try moving left and right
                tryMoveHorizontal(nextStates, piece, -1); // left
                tryMoveHorizontal(nextStates, piece, 1);  // right
            } else {
                // Try moving up and down
                tryMoveVertical(nextStates, piece, -1); // up
                tryMoveVertical(nextStates, piece, 1);  // down
            }
        }

        return nextStates;
    }

     private void tryMoveHorizontal(List<RushHourGame> nextStates, char piece,
int direction) {
    List<int[]> positions = pieces.get(piece);
    int row = positions.get(0)[0];

    int minCol = Integer.MAX_VALUE, maxCol = Integer.MIN_VALUE;
    for (int[] pos : positions) {
        minCol = Math.min(minCol, pos[1]);
        maxCol = Math.max(maxCol, pos[1]);
    }

    int step = 1;
    while (true) {
        int newCol = (direction > 0) ? maxCol + step : minCol - step;

        // Allow primary piece to exit if at the border
        if (piece == 'P' && exitRow == row && newCol == exitCol) {
            RushHourGame newState = new RushHourGame(this);
            newState.primaryRow = row;
            newState.primaryCol = exitCol;
            nextStates.add(newState);
            break;
        }

        // Check bounds
        if (newCol < 0 || newCol >= cols) break;

        // Check all positions along the path
        boolean clear = true;
        for (int s = 1; s <= step; s++) {
            int colToCheck = (direction > 0) ? maxCol + s : minCol - s;
              if (colToCheck < 0 || colToCheck >= cols || board[row][colToCheck]
!= '.') {
                clear = false;
                break;
            }
        }
```

```
        if (!clear) break;

        // Create new state
        RushHourGame newState = new RushHourGame(this);

        // Clear old positions
        for (int[] pos : positions) {
            newState.board[pos[0]][pos[1]] = '.';
        }

        // Create updated positions
        List<int[]> newPositions = new ArrayList<>();
        for (int[] pos : positions) {
            int newY = pos[1] + direction * step;
            newState.board[row][newY] = piece;
            newPositions.add(new int[]{row, newY});
        }

        newState.pieces.put(piece, newPositions);
        if (piece == 'P') {
            newState.primaryRow = row;
            newState.primaryCol = newPositions.get(0)[1];
        }

        nextStates.add(newState);
        step++;
    }
}

    private void tryMoveVertical(List<RushHourGame> nextStates, char piece, int
direction) {
    List<int[]> positions = pieces.get(piece);
    int col = positions.get(0)[1];

    int minRow = Integer.MAX_VALUE, maxRow = Integer.MIN_VALUE;
    for (int[] pos : positions) {
        minRow = Math.min(minRow, pos[0]);
        maxRow = Math.max(maxRow, pos[0]);
    }

    int step = 1;
    while (true) {
        int newRow = (direction > 0) ? maxRow + step : minRow - step;

        // Allow primary piece to exit if at the border
        if (piece == 'P' && exitCol == col && newRow == exitRow) {
            RushHourGame newState = new RushHourGame(this);
            newState.primaryRow = exitRow;
            newState.primaryCol = col;
            nextStates.add(newState);
            break;
        }

        // Check bounds
        if (newRow < 0 || newRow >= rows) break;
```

```
        // Check all positions along the path
        boolean clear = true;
        for (int s = 1; s <= step; s++) {
            int rowToCheck = (direction > 0) ? maxRow + s : minRow - s;
              if (rowToCheck < 0 || rowToCheck >= rows || board[rowToCheck][col]
!= '.') {
                clear = false;
                break;
            }
        }

        if (!clear) break;

        // Create new state
        RushHourGame newState = new RushHourGame(this);

        // Clear old positions
        for (int[] pos : positions) {
            newState.board[pos[0]][pos[1]] = '.';
        }

        // Create updated positions
        List<int[]> newPositions = new ArrayList<>();
        for (int[] pos : positions) {
            int newX = pos[0] + direction * step;
            newState.board[newX][col] = piece;
            newPositions.add(new int[]{newX, col});
        }

        newState.pieces.put(piece, newPositions);
        if (piece == 'P') {
            newState.primaryRow = newPositions.get(0)[0];
            newState.primaryCol = col;
        }

        nextStates.add(newState);
        step++;
    }
}

    // Calculate heuristic 1: Manhattan distance from primary piece to exit
    public int getHeuristic1() {
        if (isGoalState()) return 0;

        List<int[]> primaryPositions = pieces.get('P');
        if (primaryPositions == null) return Integer.MAX_VALUE;

        // Determine if primary piece is horizontal or vertical
        boolean isHorizontal = primaryPositions.size() > 1 &&
                primaryPositions.get(0)[0] == primaryPositions.get(1)[0];

        if (isHorizontal) {
            // Primary piece is horizontal
            int minCol = Integer.MAX_VALUE, maxCol = Integer.MIN_VALUE;
            for (int[] pos : primaryPositions) {
                minCol = Math.min(minCol, pos[1]);
```

```
                maxCol = Math.max(maxCol, pos[1]);
            }

            // Calculate distance to exit
            if (exitRow == primaryRow) {
                if (exitCol == cols) {
                    // Exit is at right border
                    return cols - maxCol - 1;
                } else if (exitCol > maxCol) {
                    return exitCol - maxCol;
                } else if (exitCol < minCol) {
                    return minCol - exitCol;
                }
            }
             return Math.abs(exitRow - primaryRow) + Math.min(Math.abs(exitCol -
minCol), Math.abs(exitCol - maxCol));
        } else {
            // Primary piece is vertical
            int minRow = Integer.MAX_VALUE, maxRow = Integer.MIN_VALUE;
            for (int[] pos : primaryPositions) {
                minRow = Math.min(minRow, pos[0]);
                maxRow = Math.max(maxRow, pos[0]);
            }

            // Calculate distance to exit
            if (exitCol == primaryCol) {
                if (exitRow == rows) {
                    // Exit is at bottom border
                    return rows - maxRow - 1;
                } else if (exitRow > maxRow) {
                    return exitRow - maxRow;
                } else if (exitRow < minRow) {
                    return minRow - exitRow;
                }
            }
             return Math.abs(exitCol - primaryCol) + Math.min(Math.abs(exitRow -
minRow), Math.abs(exitRow - maxRow));
        }
    }

    // Calculate heuristic 2: Number of blocking pieces + distance
    public int getHeuristic2() {
        if (isGoalState()) return 0;

        List<int[]> primaryPositions = pieces.get('P');
        if (primaryPositions == null) return Integer.MAX_VALUE;

        // Determine if primary piece is horizontal or vertical
        boolean isHorizontal = primaryPositions.size() > 1 &&
                primaryPositions.get(0)[0] == primaryPositions.get(1)[0];

        int blockingPieces = 0;
        int distance = 0;

        if (isHorizontal) {
            // Primary piece is horizontal
```

```
            int minCol = Integer.MAX_VALUE, maxCol = Integer.MIN_VALUE;
            for (int[] pos : primaryPositions) {
                minCol = Math.min(minCol, pos[1]);
                maxCol = Math.max(maxCol, pos[1]);
            }

            // Count blocking pieces and calculate distance
            if (exitRow == primaryRow) {
                if (exitCol == cols) {
                    // Exit is at right border
                    distance = cols - maxCol - 1;
                    for (int j = maxCol + 1; j < cols; j++) {
                        if (board[exitRow][j] != '.') {
                            blockingPieces++;
                        }
                    }
                } else if (exitCol > maxCol) {
                    distance = exitCol - maxCol;
                    for (int j = maxCol + 1; j < exitCol; j++) {
                        if (board[exitRow][j] != '.') {
                            blockingPieces++;
                        }
                    }
                } else if (exitCol < minCol) {
                    distance = minCol - exitCol;
                    for (int j = exitCol + 1; j < minCol; j++) {
                        if (board[exitRow][j] != '.') {
                            blockingPieces++;
                        }
                    }
                }
            } else {
                distance = Math.abs(exitRow - primaryRow);
                      blockingPieces = 1; // Primary piece needs to be moved
vertically first
            }
        } else {
            // Primary piece is vertical
            int minRow = Integer.MAX_VALUE, maxRow = Integer.MIN_VALUE;
            for (int[] pos : primaryPositions) {
                minRow = Math.min(minRow, pos[0]);
                maxRow = Math.max(maxRow, pos[0]);
            }

            // Count blocking pieces and calculate distance
            if (exitCol == primaryCol) {
                if (exitRow == rows) {
                    // Exit is at bottom border
                    distance = rows - maxRow - 1;
                    for (int i = maxRow + 1; i < rows; i++) {
                        if (board[i][exitCol] != '.') {
                            blockingPieces++;
                        }
                    }
                } else if (exitRow > maxRow) {
                    distance = exitRow - maxRow;
```

```
                    for (int i = maxRow + 1; i < exitRow; i++) {
                        if (board[i][exitCol] != '.') {
                            blockingPieces++;
                        }
                    }
                } else if (exitRow < minRow) {
                    distance = minRow - exitRow;
                    for (int i = exitRow + 1; i < minRow; i++) {
                        if (board[i][exitCol] != '.') {
                            blockingPieces++;
                        }
                    }
                }
            } else {
                distance = Math.abs(exitCol - primaryCol);
                       blockingPieces = 1; // Primary piece needs to be moved
horizontally first
            }
        }

        return blockingPieces * 2 + distance;
    }

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

        RushHourGame other = (RushHourGame) 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 (board[i][j] != other.board[i][j]) return false;
            }
        }

        return true;
    }

    @Override
    public int hashCode() {
        int hash = rows * 31 + cols;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                hash = hash * 31 + board[i][j];
            }
        }
        return hash;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < rows; i++) {
```

```
            for (int j = 0; j < cols; j++) {
                sb.append(board[i][j]);
            }
            sb.append('\n');
        }
        return sb.toString();
    }

    // Getters
    public int getRows() { return rows; }
    public int getCols() { return cols; }
    public char[][] getBoard() { return board; }
    public char getPrimaryPiece() { return primaryPiece; }
    public char getExit() { return exit; }
    public int getExitRow() { return exitRow; }
    public int getExitCol() { return exitCol; }
    public int getPrimaryRow() { return primaryRow; }
    public int getPrimaryCol() { return primaryCol; }
    public Map<Character, List<int[]>> getPieces() { return pieces; }

    // Setter methods
    public void setExitPosition(int row, int col) {
        this.exitRow = row;
        this.exitCol = col;
    }

    public void setPrimaryPosition(int row, int col) {
        this.primaryRow = row;
        this.primaryCol = col;
    }

    public void clearPieces() {
        this.pieces.clear();
    }

    public void addPiecePosition(char piece, int row, int col) {
        if (!pieces.containsKey(piece)) {
            pieces.put(piece, new ArrayList<>());
        }
        pieces.get(piece).add(new int[]{row, col});
    }

    // Removes the primary piece ('P') from the board and pieces map
    public void removePrimaryPiece() {
        List<int[]> primaryPositions = pieces.get('P');
        if (primaryPositions != null) {
            for (int[] pos : primaryPositions) {
                board[pos[0]][pos[1]] = '.';
            }
            pieces.remove('P');
        }
    }
}
```

4.7 RushHourGUI.java

```
package src;

import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.filechooser.FileNameExtensionFilter;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import javax.swing.UnsupportedLookAndFeelException;


public class RushHourGUI extends JFrame {
    // Main components
    private RushHourGame currentGame;
    private List<RushHourGame> currentSolution;
    private List<String> currentActions;
    private SearchAlgorithm currentAlgorithm;

    // GUI Components
    private GamePanel gamePanel;
    private GameAnimation gameAnimation;

    // Control Panel Components
    private JComboBox<String> algorithmComboBox;
    private JComboBox<String> heuristicComboBox;
    private JButton loadButton;
    private JButton solveButton;
    private JButton resetButton;
    private JButton saveButton; // New Save button
    private JLabel statusLabel;

    // Animation Control Components
    private JButton playButton;
    private JButton pauseButton;
    private JButton stepButton;
    private JButton stepBackButton;
    private JSlider speedSlider;
    private JLabel moveLabel;

    // Results Panel Components
    private JTextArea resultsTextArea;
    private JLabel nodesVisitedLabel;
    private JLabel executionTimeLabel;
    private JLabel solutionLengthLabel;

    // Store panels as instance variables
    private JPanel controlPanel;
    private JPanel animationPanel;
    private JPanel resultsPanel;
```

```
    // Layout constants
    private static final int WINDOW_WIDTH = 1200;
    private static final int WINDOW_HEIGHT = 800;
    private static final int GAME_PANEL_SIZE = 600;

    public RushHourGUI() {
        initializeGUI();
        setupEventHandlers();
    }


    private void initializeGUI() {
        setTitle("Rush Hour Puzzle Solver");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
        setLocationRelativeTo(null);

        // Set layout first
        setLayout(new BorderLayout());

        // Create main panels
        createGamePanel();
        createControlPanel();
        createAnimationControlPanel();
        createResultsPanel();

        // Setup main layout - add all panels
        setupMainLayout();

        // Force validation
        validate();

        // Initialize with disabled state
        updateGUIState(false);
    }


    private void createGamePanel() {
        gamePanel = new GamePanel();
                        gamePanel.setPreferredSize(new   Dimension(GAME_PANEL_SIZE,
GAME_PANEL_SIZE));
                  gamePanel.setBorder(BorderFactory.createTitledBorder("Rush  Hour
Board"));

        gameAnimation = new GameAnimation(gamePanel);
    }

    private void createControlPanel() {
        controlPanel = new JPanel(new GridBagLayout());
        controlPanel.setBorder(BorderFactory.createTitledBorder("Controls"));
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 5, 5, 5);

        // Algorithm selection
        gbc.gridx = 0; gbc.gridy = 0;
```

```
        controlPanel.add(new JLabel("Algorithm:"), gbc);

        gbc.gridx = 1;
        algorithmComboBox = new JComboBox<>(new String[]{
                "UCS (Uniform Cost Search)",
                "GBFS (Greedy Best First Search)",
                "A* (A-Star)",
                "IDA* (Iterative Deepening A*)"
        });
        controlPanel.add(algorithmComboBox, gbc);

        // Heuristic selection
        gbc.gridx = 0; gbc.gridy = 1;
        controlPanel.add(new JLabel("Heuristic:"), gbc);

        gbc.gridx = 1;
        heuristicComboBox = new JComboBox<>(new String[]{
                "H1 - Manhattan Distance",
                "H2 - Blocking Pieces + Distance"
        });
        heuristicComboBox.setEnabled(false);
        controlPanel.add(heuristicComboBox, gbc);

        // Buttons
        gbc.gridx = 0; gbc.gridy = 2;
        loadButton = new JButton("Load Puzzle");
        System.out.println("Load button created");
        controlPanel.add(loadButton, gbc);

        gbc.gridx = 1;
        solveButton = new JButton("Solve Puzzle");
        solveButton.setEnabled(false);
        controlPanel.add(solveButton, gbc);

        gbc.gridx = 0; gbc.gridy = 3;
        resetButton = new JButton("Reset");
        controlPanel.add(resetButton, gbc);

        // Add the new Save Solution button
        gbc.gridx = 1; gbc.gridy = 3;
        saveButton = new JButton("Save Solution");
        saveButton.setEnabled(false); // Initially disabled
        controlPanel.add(saveButton, gbc);

        // Status label
        gbc.gridx = 0; gbc.gridy = 4; gbc.gridwidth = 2;
        statusLabel = new JLabel("Ready - Load a puzzle to begin");
        statusLabel.setHorizontalAlignment(SwingConstants.CENTER);
        controlPanel.add(statusLabel, gbc);
    }

    private void createAnimationControlPanel() {
        animationPanel = new JPanel(new FlowLayout());
            animationPanel.setBorder(BorderFactory.createTitledBorder("Animation
Controls"));
```

```
        // Control buttons
        playButton = new JButton("▶ Play");
        pauseButton = new JButton("⏸ Pause");
        stepBackButton = new JButton("⏮ Step Back");
        stepButton = new JButton("⏭ Step Forward");

        // Speed control
        speedSlider = new JSlider(1, 10, 5);
        speedSlider.setMinorTickSpacing(1);
        speedSlider.setMajorTickSpacing(5);
        speedSlider.setPaintTicks(true);
        speedSlider.setPaintLabels(true);

        // Move information
        moveLabel = new JLabel("Move: 0/0");

        // Add components
        animationPanel.add(stepBackButton);
        animationPanel.add(playButton);
        animationPanel.add(pauseButton);
        animationPanel.add(stepButton);
        animationPanel.add(new JLabel("Speed:"));
        animationPanel.add(speedSlider);
        animationPanel.add(moveLabel);

        // Initially disable all animation controls
        setAnimationControlsEnabled(false);
    }

    private void createResultsPanel() {
        resultsPanel = new JPanel(new BorderLayout());
        resultsPanel.setBorder(BorderFactory.createTitledBorder("Results"));

        // Statistics labels
        JPanel statsPanel = new JPanel(new GridLayout(3, 1));
        nodesVisitedLabel = new JLabel("Nodes Visited: -");
        executionTimeLabel = new JLabel("Execution Time: -");
        solutionLengthLabel = new JLabel("Solution Length: -");

        statsPanel.add(nodesVisitedLabel);
        statsPanel.add(executionTimeLabel);
        statsPanel.add(solutionLengthLabel);

        // Results text area
        resultsTextArea = new JTextArea(5, 20);
        resultsTextArea.setEditable(false);
        resultsTextArea.setFont(new Font(Font.MONOSPACED, Font.PLAIN, 12));
        JScrollPane scrollPane = new JScrollPane(resultsTextArea);

        resultsPanel.add(statsPanel, BorderLayout.NORTH);
        resultsPanel.add(scrollPane, BorderLayout.CENTER);
    }

    private void setupMainLayout() {
        // Create center panel for game display
        JPanel centerPanel = new JPanel(new FlowLayout());
```

```
        centerPanel.add(gamePanel);

        // Add all panels to the main frame
        add(centerPanel, BorderLayout.CENTER);
        add(controlPanel, BorderLayout.WEST);
        add(animationPanel, BorderLayout.SOUTH);
        add(resultsPanel, BorderLayout.EAST);

        System.out.println("Layout setup complete");
    }

    private void setupEventHandlers() {
        // Algorithm selection handler
        algorithmComboBox.addActionListener(e -> {
            int selected = algorithmComboBox.getSelectedIndex();
            // UCS doesn't use heuristics, others do
            heuristicComboBox.setEnabled(selected != 0);
        });

        // Load button handler
        loadButton.addActionListener(e -> loadPuzzle());

        // Solve button handler
        solveButton.addActionListener(e -> solvePuzzle());

        // Reset button handler
        resetButton.addActionListener(e -> resetGame());

        // Save button handler
        saveButton.addActionListener(e -> saveSolution());

        // Animation control handlers
        playButton.addActionListener(e -> gameAnimation.play());
        pauseButton.addActionListener(e -> gameAnimation.pause());
        stepButton.addActionListener(e -> gameAnimation.stepForward());
        stepBackButton.addActionListener(e -> gameAnimation.stepBackward());

        // Speed slider handler
        speedSlider.addChangeListener(e -> {
            int speed = speedSlider.getValue();
            gameAnimation.setSpeed(speed);
        });

        // Updated animation event handler with better handling of the last step
        gameAnimation.setAnimationListener(new AnimationListener() {
            @Override
            public void onMoveChanged(int currentMove, int totalMoves) {
                moveLabel.setText("Move: " + currentMove + "/" + totalMoves);

                // Update results text area with current move
                 if (currentActions != null && currentMove > 0 && currentMove <=
currentActions.size()) {
                    String action = currentActions.get(currentMove - 1);
                    resultsTextArea.append("Move " + currentMove + ": " + action
+ "\n");
```

```
resultsTextArea.setCaretPosition(resultsTextArea.getDocument().getLength());
                }

                // Special handling for the final move
                if (currentMove == totalMoves) {
                       statusLabel.setText("Solution complete! P has reached the
exit.");
                } else if (currentMove == 0) {
                    statusLabel.setText("Initial state");
                } else {
                             statusLabel.setText("Playing animation... Step " +
currentMove + "/" + totalMoves);
                }
            }

            @Override
            public void onAnimationComplete() {
                setAnimationControlsEnabled(true);
                // Give user time to see the final state
                  statusLabel.setText("Animation complete - Solution shown. Use
Reset to return to start.");
            }

            @Override
            public void onAnimationStarted() {
                statusLabel.setText("Playing animation...");
            }
        });
    }

    private void loadPuzzle() {
        JFileChooser fileChooser = new JFileChooser();
            fileChooser.setFileFilter(new FileNameExtensionFilter("Text files",
"txt"));
        fileChooser.setCurrentDirectory(new File("."));

        int result = fileChooser.showOpenDialog(this);
        if (result == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fileChooser.getSelectedFile();
            try {
                                                                currentGame    =
RushHourIO.loadGameFromFile(selectedFile.getAbsolutePath());
                gamePanel.setGame(currentGame);
                gamePanel.repaint();

                // Reset solution-related data
                currentSolution = null;
                currentActions = null;
                currentAlgorithm = null;

                updateGUIState(true);
                statusLabel.setText("Puzzle loaded: " + selectedFile.getName());

                // Clear previous results
                clearResults();
```

```
                // Disable the save button since we don't have a solution yet
                saveButton.setEnabled(false);

            } catch (IOException ex) {
                JOptionPane.showMessageDialog(this,
                        "Error loading puzzle:\n" + ex.getMessage(),
                        "Load Error",
                        JOptionPane.ERROR_MESSAGE);
                statusLabel.setText("Error loading puzzle");
            }
        }
    }

    private void solvePuzzle() {
        if (currentGame == null) {
            JOptionPane.showMessageDialog(this,
                    "Please load a puzzle first!",
                    "No Puzzle",
                    JOptionPane.WARNING_MESSAGE);
            return;
        }

        // Get selected algorithm and heuristic
        int algorithmIndex = algorithmComboBox.getSelectedIndex();
        int heuristicIndex = heuristicComboBox.getSelectedIndex() + 1;

        // Create algorithm instance
        currentAlgorithm = createAlgorithm(algorithmIndex, heuristicIndex);
        SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>() {
            @Override
            protected Void doInBackground() throws Exception {
                statusLabel.setText("Solving puzzle...");
                solveButton.setEnabled(false);
                    saveButton.setEnabled(false); // Disable save button during
solving

                // Create a copy of the game for solving
                RushHourGame gameCopy = new RushHourGame(currentGame);
                currentSolution = currentAlgorithm.solve(gameCopy);
                currentActions = currentAlgorithm.getSolutionActions();

                if (currentSolution != null && !currentSolution.isEmpty()) {
                                    List<RushHourGame>  solutionWithExit  =  new
ArrayList<>(currentSolution);
                                          List<String>  actionsWithExit  =  new
ArrayList<>(currentActions);
                                              RushHourGame  finalState  =  new
RushHourGame(solutionWithExit.get(solutionWithExit.size() - 1));
                    finalState.removePrimaryPiece();
                    solutionWithExit.add(finalState);
                    actionsWithExit.add("EXIT");
                    currentSolution = solutionWithExit;
                    currentActions = actionsWithExit;
                }
```

```
                return null;
            }

            @Override
            protected void done() {
                solveButton.setEnabled(true);

                if (currentSolution != null && !currentSolution.isEmpty()) {
                    displayResults();
                    setupAnimation();
                    statusLabel.setText("Solution found! Ready to animate.");
                    saveButton.setEnabled(true);
                } else {
                    statusLabel.setText("No solution found.");
                    saveButton.setEnabled(false);

                    JOptionPane.showMessageDialog(RushHourGUI.this,
                            "No solution found for this puzzle.",
                            "No Solution",
                            JOptionPane.INFORMATION_MESSAGE);
                }
            }
        };

        worker.execute();
    }

    private void saveSolution() {
                if  (currentSolution  ==  null  ||  currentSolution.isEmpty()  ||
currentAlgorithm == null) {
            JOptionPane.showMessageDialog(this,
                    "No solution available to save!",
                    "No Solution",
                    JOptionPane.WARNING_MESSAGE);
            return;
        }

        JFileChooser fileChooser = new JFileChooser();
            fileChooser.setFileFilter(new  FileNameExtensionFilter("Text  files",
"txt"));
        fileChooser.setCurrentDirectory(new File("."));

        // Generate a default filename based on algorithm
           String defaultFilename = RushHourIO.generateOutputFilename("puzzle",
currentAlgorithm);
        fileChooser.setSelectedFile(new File(defaultFilename));

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

            // Add .txt extension if not present
            String filePath = selectedFile.getAbsolutePath();
            if (!filePath.toLowerCase().endsWith(".txt")) {
                filePath += ".txt";
                selectedFile = new File(filePath);
```

```
            }

            // Check if file exists and confirm overwrite
            if (selectedFile.exists()) {
                int response = JOptionPane.showConfirmDialog(this,
                        "File already exists. Overwrite?",
                        "Confirm Overwrite",
                        JOptionPane.YES_NO_OPTION);
                if (response != JOptionPane.YES_OPTION) {
                    return;
                }
            }

            try {
                 RushHourIO.writeSolutionToFile(currentSolution, currentActions,
currentAlgorithm, filePath);
                                statusLabel.setText("Solution  saved  to:  "  +
selectedFile.getName());
            } catch (IOException ex) {
                JOptionPane.showMessageDialog(this,
                        "Error saving solution:\n" + ex.getMessage(),
                        "Save Error",
                        JOptionPane.ERROR_MESSAGE);
                statusLabel.setText("Error saving solution");
            }
        }
    }

          private  SearchAlgorithm  createAlgorithm(int  algorithmIndex,  int
heuristicIndex) {
        switch (algorithmIndex) {
            case 0:
                return new UCS();
            case 1:
                return new GBFS(heuristicIndex);
            case 2:
                return new AStar(heuristicIndex);
            case 3:
                return new IDAStar(heuristicIndex);
            default:
                return new UCS();
        }
    }

    private void displayResults() {
        if (currentAlgorithm == null) return;

        // Update statistics labels
                           nodesVisitedLabel.setText("Nodes   Visited:   "   +
currentAlgorithm.getNodesVisited());
                          executionTimeLabel.setText("Execution   Time:   "   +
currentAlgorithm.getExecutionTime() + " ms");
                         solutionLengthLabel.setText("Solution   Length:   "   +
(currentSolution.size() - 1) + " moves");

        // Display algorithm information in text area
```

```
        resultsTextArea.setText("");
                                    resultsTextArea.append("Algorithm:    "    +
currentAlgorithm.getAlgorithmName() + "\n");
                                  resultsTextArea.append("Description:    "    +
currentAlgorithm.getAlgorithmDescription() + "\n\n");
        resultsTextArea.append("Solution Steps:\n");
    }


    private void setupAnimation() {
        if (currentSolution != null && currentActions != null) {
            gameAnimation.setSolution(currentSolution, currentActions);
            setAnimationControlsEnabled(true);
            moveLabel.setText("Move: 0/" + (currentSolution.size() - 1));
        }
    }


    private void resetGame() {
        if (currentGame != null) {
            gamePanel.setGame(currentGame);
            gamePanel.repaint();

            if (gameAnimation != null) {
                gameAnimation.reset();
                      moveLabel.setText("Move: 0/" + (currentSolution != null ?
currentSolution.size() - 1 : 0));
            }

            // Clear only the solution steps from results, keep the statistics
            if (currentAlgorithm != null) {
                resultsTextArea.setText("");
                                     resultsTextArea.append("Algorithm:   "   +
currentAlgorithm.getAlgorithmName() + "\n");
                                    resultsTextArea.append("Description:   "   +
currentAlgorithm.getAlgorithmDescription() + "\n\n");
                resultsTextArea.append("Solution Steps:\n");
            }

            statusLabel.setText("Game reset to initial state");
        }
    }

    private void clearResults() {
        nodesVisitedLabel.setText("Nodes Visited: -");
        executionTimeLabel.setText("Execution Time: -");
        solutionLengthLabel.setText("Solution Length: -");
        resultsTextArea.setText("");
        moveLabel.setText("Move: 0/0");
        setAnimationControlsEnabled(false);
    }

    private void updateGUIState(boolean puzzleLoaded) {
        solveButton.setEnabled(puzzleLoaded);
        resetButton.setEnabled(puzzleLoaded);
        // Save button should only be enabled when there's a solution
```

```
            saveButton.setEnabled(puzzleLoaded && currentSolution != null &&
!currentSolution.isEmpty());
    }

    private void setAnimationControlsEnabled(boolean enabled) {
        playButton.setEnabled(enabled);
        pauseButton.setEnabled(enabled);
        stepButton.setEnabled(enabled);
        stepBackButton.setEnabled(enabled);
        speedSlider.setEnabled(enabled);
    }


    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            try {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
            } catch (Exception e) {
                e.printStackTrace();
            }

            RushHourGUI gui = new RushHourGUI();
            gui.setVisible(true);

            // Additional debug
            System.out.println("GUI visible");
        });
    }
}

interface AnimationListener {
    void onMoveChanged(int currentMove, int totalMoves);
    void onAnimationComplete();
    void onAnimationStarted();
}
```

4.8 RushHourIO.java

```
package src;

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

public class RushHourIO {
        public static RushHourGame loadGameFromFile(String filename) throws
IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filename));

        try {
            // Read dimensions
            String[] dimensions = reader.readLine().trim().split("\\s+");
            if (dimensions.length != 2) {
                  throw new IOException("Invalid format: First line must contain
two integers (rows cols)");
            }
```

```
            int rows = Integer.parseInt(dimensions[0]);
            int cols = Integer.parseInt(dimensions[1]);

            if (rows <= 0 || cols <= 0 || rows > 20 || cols > 20) {
                    throw new IOException("Invalid dimensions: Must be between
1-20");
            }

            // Read number of pieces (excluding primary piece)
            String numPiecesStr = reader.readLine().trim();
            int numPieces = Integer.parseInt(numPiecesStr);

            if (numPieces < 0 || numPieces > 50) {
                throw new IOException("Invalid number of pieces: Must be between
0-50");
            }

            // Create game instance
            RushHourGame game = new RushHourGame(rows, cols);
            char[][] board = game.getBoard();
            boolean exitFound = false;

            // Detect top exit before reading the board
            String peekLine = reader.readLine();
                    if (peekLine != null && peekLine.trim().length() == 1 &&
peekLine.trim().charAt(0) == 'K') {
                int topExitCol = peekLine.indexOf('K');
                game.setExitPosition(-1, topExitCol);
                exitFound = true;
                peekLine = reader.readLine();
            }

            // Read all lines after header and possible top exit
            List<String> allLines = new ArrayList<>();
            if (peekLine != null) allLines.add(peekLine);
            String line;
            while ((line = reader.readLine()) != null) {
                allLines.add(line);
            }

            // Find possible bottom exit line (a line with only 'K' and spaces)
            int bottomExitIdx = -1;
            for (int i = 0; i < allLines.size(); i++) {
                String l = allLines.get(i);
                if (l.trim().length() == 1 && l.trim().charAt(0) == 'K') {
                    bottomExitIdx = i;
                    break;
                }
            }

                  int boardEndIdx = (bottomExitIdx == -1) ? allLines.size() :
bottomExitIdx;
            List<String> boardLines = allLines.subList(0, boardEndIdx);

            // Validate number of board lines
```

```
            if (boardLines.size() != rows) {
                        throw new IOException("Number of board lines (" +
boardLines.size() + ") does not match specified rows (" + rows + ")");
            }

            // Validate each board line's length
            for (int i = 0; i < boardLines.size(); i++) {
                String bl = boardLines.get(i);
                int expected1 = cols;
                int expected2 = cols + 1;
                if (bl.length() != expected1 && bl.length() != expected2) {
                       throw new IOException("Row " + (i + 1) + " has incorrect
length: expected " + expected1 + " or " + expected2 + ", got " + bl.length());
                }
            }

            // Fill the board as before
            for (int i = 0; i < rows; i++) {
                String bl = boardLines.get(i);
                if (bl.length() == cols + 1 && bl.charAt(0) == 'K') {
                    // Exit on left
                    for (int j = 1; j <= cols; j++) {
                        board[i][j - 1] = bl.charAt(j);
                    }
                    game.setExitPosition(i, -1);
                    exitFound = true;
                } else if (bl.length() == cols + 1 && bl.charAt(cols) == 'K') {
                    // Exit on right
                    for (int j = 0; j < cols; j++) {
                        board[i][j] = bl.charAt(j);
                    }
                    game.setExitPosition(i, cols);
                    exitFound = true;
                } else if (bl.length() == cols + 1 && bl.charAt(0) == ' ') {
                    for (int j = 1; j <= cols; j++) {
                        board[i][j - 1] = bl.charAt(j);
                    }
                } else if (bl.length() == cols) {
                    for (int j = 0; j < cols; j++) {
                        board[i][j] = bl.charAt(j);
                    }
                } else {
                       throw new IOException("Row " + (i + 1) + " has incorrect
length or invalid 'K' placement");
                }
            }

            // Handle bottom exit if present
            if (bottomExitIdx != -1) {
                String bottomLine = allLines.get(bottomExitIdx);
                int colK = bottomLine.indexOf('K');
                game.setExitPosition(rows, colK);
                exitFound = true;
                for (int i = bottomExitIdx + 1; i < allLines.size(); i++) {
                    if (!allLines.get(i).trim().isEmpty()) {
```

```
                        throw new IOException("Extra non-empty line found after
board and exit: '" + allLines.get(i) + "'");
                    }
                }
            } else {
                for (int i = boardEndIdx; i < allLines.size(); i++) {
                    if (!allLines.get(i).trim().isEmpty()) {
                        throw new IOException("Extra non-empty line found after
board: '" + allLines.get(i) + "'");
                    }
                }
            }

            if (!exitFound) {
                    throw new IOException("No exit 'K' found outside the board
boundaries");
            }

            // Validate and initialize game state
            validateAndInitializeGame(game);

            // Validate number of unique pieces (excluding 'P')
            Set<Character> uniquePieces = new HashSet<>();
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                    char c = board[i][j];
                    if (c != '.' && c != 'P') {
                        uniquePieces.add(c);
                    }
                }
            }
            if (uniquePieces.size() != numPieces) {
                  throw new IOException("Number of unique pieces (excluding 'P')
in board (" + uniquePieces.size() + ") does not match specified number (" +
numPieces + ")");
            }

            return game;

        } catch (NumberFormatException e) {
            throw new IOException("Invalid number format in input file", e);
        } finally {
            reader.close();
        }
    }

      private static void validateAndInitializeGame(RushHourGame game) throws
IOException {
        char[][] board = game.getBoard();
        int rows = game.getRows();
        int cols = game.getCols();

        boolean foundPrimary = false;
        game.clearPieces();

        // Scan board and categorize pieces
```

```
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                char c = board[i][j];

                if (c == 'P') {
                    foundPrimary = true;
                    game.addPiecePosition(c, i, j);
                     if (game.getPrimaryRow() == 0 && game.getPrimaryCol() == 0)
{
                        game.setPrimaryPosition(i, j);
                    }
                } else if (c == 'K') {
                    throw new IOException("Exit 'K' found inside the board at ("
+ i + ", " + j + "). Exit must be outside the board.");
                } else if (c != '.' && c != 'K') {
                    // Regular piece
                    game.addPiecePosition(c, i, j);
                }
            }
        }

        // Validate required elements
        if (!foundPrimary) {
            throw new IOException("No primary piece 'P' found in the board");
        }

        // Validate exit position is set
        if (game.getExitRow() == 0 && game.getExitCol() == 0) {
             throw new IOException("No exit position set. Exit 'K' must be found
outside the board.");
        }

        // Validate primary piece
        List<int[]> primaryPositions = game.getPieces().get('P');
        if (primaryPositions == null || primaryPositions.size() < 2) {
            throw new IOException("Primary piece must occupy at least 2 cells");
        }

        // Validate piece continuity and orientation for all pieces
                        for   (Map.Entry<Character,   List<int[]>>   entry   :
game.getPieces().entrySet()) {
            char piece = entry.getKey();
            List<int[]> positions = entry.getValue();

            if (positions.size() < 2) {
                    throw new IOException("Piece '" + piece + "' must occupy at
least 2 cells");
            }

            // Check if piece forms a continuous line
            if (!isValidPieceShape(positions)) {
                     throw new IOException("Piece '" + piece + "' must form a
continuous horizontal or vertical line");
            }
        }
```

```
        // Validate exit position (must be on board edge or outside)
        int exitRow = game.getExitRow();
        int exitCol = game.getExitCol();

        boolean validExit = false;

        // Check if exit is on border (outside the board)
        if (exitCol == cols && exitRow >= 0 && exitRow < rows) {
            // Right border
            validExit = true;
        } else if (exitRow == rows && exitCol >= 0 && exitCol < cols) {
            // Bottom border
            validExit = true;
        } else if (exitCol == -1 && exitRow >= 0 && exitRow < rows) {
            // Left border
            validExit = true;
        } else if (exitRow == -1 && exitCol >= 0 && exitCol < cols) {
            // Top border
            validExit = true;
        }

        if (!validExit) {
              throw new IOException("Exit must be on the board edge (outside the
board bounds). Found at (" + exitRow + ", " + exitCol + ")");
        }

        // Validate that primary piece and exit are aligned
        boolean isHorizontal = checkPrimaryOrientation(game);
        if (isHorizontal && exitRow != game.getPrimaryRow()) {
              throw new IOException("Primary piece is horizontal but exit is not
on the same row");
        } else if (!isHorizontal && exitCol != game.getPrimaryCol()) {
             throw new IOException("Primary piece is vertical but exit is not on
the same column");
        }
    }

    private static boolean checkPrimaryOrientation(RushHourGame game) {
        List<int[]> primaryPositions = game.getPieces().get('P');
        if (primaryPositions.size() < 2) return true;

        return primaryPositions.get(0)[0] == primaryPositions.get(1)[0];
    }

    private static boolean isValidPieceShape(List<int[]> positions) {
        if (positions.size() < 2) return false;

        // Sort positions
        positions.sort((a, b) -> {
            if (a[0] != b[0]) return Integer.compare(a[0], b[0]);
            return Integer.compare(a[1], b[1]);
        });

        // Check if all positions are in same row (horizontal piece)
        boolean horizontal = true;
        for (int i = 1; i < positions.size(); i++) {
```

```
            if (positions.get(i)[0] != positions.get(0)[0] ||
                    positions.get(i)[1] != positions.get(i-1)[1] + 1) {
                horizontal = false;
                break;
            }
        }

        if (horizontal) return true;

        // Check if all positions are in same column (vertical piece)
        boolean vertical = true;
        for (int i = 1; i < positions.size(); i++) {
            if (positions.get(i)[1] != positions.get(0)[1] ||
                    positions.get(i)[0] != positions.get(i-1)[0] + 1) {
                vertical = false;
                break;
            }
        }

        return vertical;
    }

       public static void writeSolutionToConsole(List<RushHourGame> solution,
List<String> actions, SearchAlgorithm algorithm) {
        if (solution == null || solution.isEmpty()) {
            System.out.println("No solution found!");
            return;
        }

        // Print header
          System.out.println("=== " + algorithm.getAlgorithmName() + " Solution
===");
        System.out.println("Nodes visited: " + algorithm.getNodesVisited());
        System.out.println("Execution time: " + algorithm.getExecutionTime() + "
ms");
           System.out.println("Solution length: " + (solution.size() - 1) + "
moves");
        System.out.println();

        // Print initial state
        System.out.println("Papan Awal");
        printColoredBoard(solution.get(0), null);
        System.out.println();

        // Print each move
        for (int i = 1; i < solution.size(); i++) {
            System.out.println("Gerakan " + i + ": " + actions.get(i - 1));
            printColoredBoard(solution.get(i), actions.get(i - 1));
            System.out.println();
        }
    }


        public static void writeSolutionToFile(List<RushHourGame> solution,
List<String> actions,
```

```
                                                    SearchAlgorithm algorithm, String
filename) throws IOException {
        BufferedWriter writer = new BufferedWriter(new FileWriter(filename));

        try {
            // Write header
                writer.write("=== " + algorithm.getAlgorithmName() + " Solution
===\n");
                 writer.write("Nodes visited: " + algorithm.getNodesVisited() +
"\n");
              writer.write("Execution time: " + algorithm.getExecutionTime() + "
ms\n");
                 writer.write("Solution length: " + (solution.size() - 1) + "
moves\n");
            writer.write("\n");

            if (solution == null || solution.isEmpty()) {
                writer.write("No solution found!\n");
                return;
            }

            // Write initial state
            writer.write("Papan Awal\n");
            writer.write(solution.get(0).toString());
            writer.write("\n");

            // Write each move
            for (int i = 1; i < solution.size(); i++) {
                writer.write("Gerakan " + i + ": " + actions.get(i - 1) + "\n");
                writer.write(solution.get(i).toString());
                writer.write("\n");
            }
        } finally {
            writer.close();
        }
    }

    private static void printColoredBoard(RushHourGame game, String action) {
        char[][] board = game.getBoard();
        int rows = game.getRows();
        int cols = game.getCols();

        // ANSI color codes
        String RESET = "\u001B[0m";
        String RED = "\u001B[31m";      // Primary piece
        String GREEN = "\u001B[32m";
        String YELLOW = "\u001B[33m";   // Moved piece
        String BLUE = "\u001B[34m";     // Other pieces

        char movedPiece = '.';
        if (action != null && !action.equals("Initial")) {
            movedPiece = action.charAt(0);
        }

        // Print board with border to show exit
        System.out.print("  ");
```

```java
        for (int j = 0; j < cols; j++) {
            System.out.print(j % 10);
        }
        System.out.println();

        for (int i = 0; i < rows; i++) {
            System.out.print(i + " ");
            for (int j = 0; j < cols; j++) {
                char c = board[i][j];

                if (c == game.getPrimaryPiece()) {
                    System.out.print(RED + c + RESET);
                } else if (c != '.' && c == movedPiece) {
                    System.out.print(YELLOW + c + RESET);
                } else if (c != '.' && c != game.getExit()) {
                    System.out.print(BLUE + c + RESET);
                } else {
                    System.out.print(c);
                }
            }

            // Show exit marker if it's on this row
            if (game.getExitRow() == i && game.getExitCol() == cols) {
                System.out.print(GREEN + " K" + RESET);
            }

            System.out.println();
        }
    }


    public static String getUserInput(String prompt) {
        Scanner scanner = new Scanner(System.in);
        System.out.print(prompt);
        return scanner.nextLine().trim();
    }


    public static int getAlgorithmChoice() {
        Scanner scanner = new Scanner(System.in);
        int choice = 0;

        while (choice < 1 || choice > 4) {
            System.out.println("\nSelect algorithm:");
            System.out.println("1. UCS (Uniform Cost Search)");
            System.out.println("2. GBFS (Greedy Best First Search)");
            System.out.println("3. A* (A-Star)");
            System.out.println("4. IDA* (Iterative Deepening A*)");
            System.out.print("Choice (1-4): ");

            try {
                choice = Integer.parseInt(scanner.nextLine().trim());
                if (choice < 1 || choice > 4) {
                    System.out.println("Invalid choice. Please enter 1-4.");
                }
            } catch (NumberFormatException e) {
```

```
                System.out.println("Invalid input. Please enter a number.");
            }
        }

        return choice;
    }


    public static int getHeuristicChoice() {
        Scanner scanner = new Scanner(System.in);
        int choice = 0;

        while (choice < 1 || choice > 2) {
            System.out.println("\nSelect heuristic:");
            System.out.println("1. Manhattan Distance (H1)");
            System.out.println("2. Blocking Pieces + Distance (H2)");
            System.out.print("Choice (1-2): ");

            try {
                choice = Integer.parseInt(scanner.nextLine().trim());
                if (choice < 1 || choice > 2) {
                    System.out.println("Invalid choice. Please enter 1-2.");
                }
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a number.");
            }
        }

        return choice;
    }

    public static boolean isValidFile(String filename) {
        File file = new File(filename);
        return file.exists() && file.isFile() && file.canRead();
    }

    public static String generateOutputFilename(String baseName, SearchAlgorithm
algorithm) {
                                        String     timestamp     =     new
java.text.SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
        String algName = algorithm.getAlgorithmName().replaceAll("[^a-zA-Z0-9]",
"_");
           return "solution_" + baseName + "_" + algName + "_" + timestamp +
".txt";
    }

      public static void printComparisonTable(Map<String, Map<String, Object>>
results) {
        System.out.println("\n=== Algorithm Comparison ===");
                       System.out.println("Algorithm\t\tNodes\tTime(ms)\tSolution
Length\tOptimal");
        System.out.println("----------------------------------------");

         for (Map.Entry<String, Map<String, Object>> entry : results.entrySet())
{
            String alg = entry.getKey();
```

```
            Map<String, Object> stats = entry.getValue();

            System.out.printf("%-15s\t%d\t%d\t\t%d\t\t%s\n",
                    alg,
                    (Integer) stats.get("nodes_visited"),
                    (Long) stats.get("execution_time"),
                    (Integer) stats.get("solution_length"),
                    (Boolean) stats.get("optimal") ? "Yes" : "No");
        }
    }
}
```

4.9 RushHourSolver.java

```
package src;

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

public class RushHourSolver {
    private RushHourGame initialGame;
    private Map<String, SearchAlgorithm> algorithms;
    private Map<String, Map<String, Object>> results;

    public RushHourSolver() {
        this.algorithms = new HashMap<>();
        this.results = new HashMap<>();
        initializeAlgorithms();
    }

    private void initializeAlgorithms() {
        algorithms.put("UCS", new UCS());
        algorithms.put("GBFS", new GBFS(1)); // Default to heuristic 1
        algorithms.put("A*", new AStar(1));   // Default to heuristic 1
        algorithms.put("IDA*", new IDAStar(1)); // Default to heuristic 1
    }

    public static void main(String[] args) {
        RushHourSolver solver = new RushHourSolver();

        try {
            // Load test case
            String filename;
            if (args.length > 0) {
                filename = args[0];
            } else {
                  filename = RushHourIO.getUserInput("Enter test case filename:
");
            }

            // Validate file exists
            while (!RushHourIO.isValidFile(filename)) {
                      System.out.println("File not found or not readable: " +
filename);
                  filename = RushHourIO.getUserInput("Enter test case filename:
");
```

```
            }

            // Load game from file
            System.out.println("Loading test case from: " + filename);
            solver.loadGame(filename);

            // Interactive menu
            solver.runInteractiveMode();

        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }

    public void loadGame(String filename) throws IOException {
        this.initialGame = RushHourIO.loadGameFromFile(filename);
        System.out.println("Game loaded successfully!");
            System.out.println("Board  size:  " + initialGame.getRows()  + "x"  +
initialGame.getCols());
                            System.out.println("Number   of   pieces:   "   +
initialGame.getPieces().size());

        // Display initial board
        System.out.println("\nInitial board:");
        System.out.println(initialGame);
    }

    private void runInteractiveMode() {
        Scanner scanner = new Scanner(System.in);
        boolean continueRunning = true;

        while (continueRunning) {
            System.out.println("\n=== Rush Hour Solver Menu ===");
            System.out.println("1. Run single algorithm");
            System.out.println("2. Run all algorithms (comparison)");
            System.out.println("3. Display current board");
            System.out.println("4. Load new test case");
            System.out.println("5. View previous results");
            System.out.println("6. Exit");
            System.out.print("Choice: ");

            try {
                int choice = Integer.parseInt(scanner.nextLine().trim());

                switch (choice) {
                    case 1:
                        runSingleAlgorithm();
                        break;
                    case 2:
                        runAllAlgorithms();
                        break;
                    case 3:
                        displayCurrentBoard();
                        break;
                    case 4:
```

```
                        loadNewTestCase();
                        break;
                    case 5:
                        displayPreviousResults();
                        break;
                    case 6:
                        continueRunning = false;
                        break;
                    default:
                        System.out.println("Invalid choice. Please enter 1-6.");
                }
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a number.");
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
            }
        }

        System.out.println("Thank you for using Rush Hour Solver!");
    }

    private void runSingleAlgorithm() {
        if (initialGame == null) {
                  System.out.println("No  game  loaded.  Please  load  a  test  case
first.");
            return;
        }

        // Get algorithm choice
        int algorithmChoice = RushHourIO.getAlgorithmChoice();
        SearchAlgorithm algorithm = selectAlgorithm(algorithmChoice);

        // Get heuristic choice if needed
        if (requiresHeuristic(algorithmChoice)) {
            int heuristicChoice = RushHourIO.getHeuristicChoice();
            setAlgorithmHeuristic(algorithm, heuristicChoice);
        }

        // Run algorithm
        System.out.println("\nRunning " + algorithm.getAlgorithmName() + "...");
        runAlgorithm(algorithm);

        // Display results
        displayResults(algorithm);

        // Save to file option
        saveResultsOption(algorithm);
    }

    private void runAllAlgorithms() {
        if (initialGame == null) {
                  System.out.println("No  game  loaded.  Please  load  a  test  case
first.");
            return;
        }
```

```
        System.out.println("Running all algorithms...");

        // Run UCS
        System.out.println("\n1. Running UCS...");
        UCS ucs = new UCS();
        runAlgorithm(ucs);
        results.put("UCS", ucs.getStatistics());

        // Run both heuristics for GBFS
        System.out.println("\n2. Running GBFS with H1...");
        GBFS gbfs1 = new GBFS(1);
        runAlgorithm(gbfs1);
        results.put("GBFS-H1", gbfs1.getStatistics());

        System.out.println("\n3. Running GBFS with H2...");
        GBFS gbfs2 = new GBFS(2);
        runAlgorithm(gbfs2);
        results.put("GBFS-H2", gbfs2.getStatistics());

        // Run both heuristics for A*
        System.out.println("\n4. Running A* with H1...");
        AStar aStar1 = new AStar(1);
        runAlgorithm(aStar1);
        results.put("A*-H1", aStar1.getStatistics());

        System.out.println("\n5. Running A* with H2...");
        AStar aStar2 = new AStar(2);
        runAlgorithm(aStar2);
        results.put("A*-H2", aStar2.getStatistics());

        // Run both heuristics for IDA*
        System.out.println("\n6. Running IDA* with H1...");
        IDAStar idaStar1 = new IDAStar(1);
        runAlgorithm(idaStar1);
        results.put("IDA*-H1", idaStar1.getStatistics());

        System.out.println("\n7. Running IDA* with H2...");
        IDAStar idaStar2 = new IDAStar(2);
        runAlgorithm(idaStar2);
        results.put("IDA*-H2", idaStar2.getStatistics());

        // Display comparison
        RushHourIO.printComparisonTable(results);

        // Save comparison to file
        saveComparisonOption();
    }

    private void runAlgorithm(SearchAlgorithm algorithm) {
        long startTime = System.currentTimeMillis();

        // Create a copy of the initial game for the algorithm
        RushHourGame gameCopy = new RushHourGame(initialGame);

        // Solve the puzzle
        List<RushHourGame> solution = algorithm.solve(gameCopy);
```

```
        // Record statistics
        System.out.println("Algorithm: " + algorithm.getAlgorithmName());
        System.out.println("Nodes visited: " + algorithm.getNodesVisited());
        System.out.println("Execution time: " + algorithm.getExecutionTime() + "
ms");

        if (solution != null && !solution.isEmpty()) {
              System.out.println("Solution found! Length: " + (solution.size() -
1) + " moves");
        } else {
            System.out.println("No solution found.");
        }
    }

    private void displayResults(SearchAlgorithm algorithm) {
                                if    (algorithm.getSolution()    !=    null    &&
!algorithm.getSolution().isEmpty()) {
            // Prepare solution and actions with final step
                                    List<RushHourGame>   solutionWithExit   =   new
ArrayList<>(algorithm.getSolution());
                                        List<String>   actionsWithExit   =   new
ArrayList<>(algorithm.getSolutionActions());
                                            RushHourGame   finalState   =   new
RushHourGame(solutionWithExit.get(solutionWithExit.size() - 1));
            finalState.removePrimaryPiece();
            solutionWithExit.add(finalState);
            actionsWithExit.add("EXIT");

            System.out.println("\n=== Detailed Solution ===");
            RushHourIO.writeSolutionToConsole(solutionWithExit, actionsWithExit,
algorithm);

            // Additional analysis for specific algorithms
            if (algorithm instanceof AStar) {
                                        System.out.println("\n"   +   ((AStar)
algorithm).analyzeAlgorithm());
            } else if (algorithm instanceof GBFS) {
                                            System.out.println("\n"   +   ((GBFS)
algorithm).analyzeHeuristic());
            } else if (algorithm instanceof IDAStar) {
                                       System.out.println("\n"   +   ((IDAStar)
algorithm).analyzeAlgorithm());
            }
        }
    }

    private void displayCurrentBoard() {
        if (initialGame == null) {
                  System.out.println("No  game  loaded.  Please  load  a  test  case
first.");
            return;
        }

        System.out.println("\nCurrent board:");
        System.out.println(initialGame);
```

```
        // Display game information
            System.out.println("Board size: " + initialGame.getRows() + "x" +
initialGame.getCols());
                                System.out.println("Number of pieces: " +
initialGame.getPieces().size());
                        System.out.println("Primary piece position: Row " +
(initialGame.getPrimaryRow() + 1) +
                ", Col " + (initialGame.getPrimaryCol() + 1));
         System.out.println("Exit position: Row " + (initialGame.getExitRow() +
1) +
                ", Col " + (initialGame.getExitCol() + 1));
    }


    private void loadNewTestCase() {
        try {
                 String filename = RushHourIO.getUserInput("Enter new test case
filename: ");

            while (!RushHourIO.isValidFile(filename)) {
                      System.out.println("File not found or not readable: " +
filename);
                  filename = RushHourIO.getUserInput("Enter test case filename:
");
            }

            loadGame(filename);

            // Clear previous results
            results.clear();
        } catch (IOException e) {
            System.err.println("Error loading game: " + e.getMessage());
        }
    }

    private void displayPreviousResults() {
        if (results.isEmpty()) {
            System.out.println("No previous results available.");
            return;
        }

        System.out.println("\n=== Previous Results ===");
        RushHourIO.printComparisonTable(results);
    }

    private SearchAlgorithm selectAlgorithm(int choice) {
        switch (choice) {
            case 1:
                return new UCS();
            case 2:
                return new GBFS(1);
            case 3:
                return new AStar(1);
            case 4:
                return new IDAStar(1);
```

```
            default:
                return new UCS();
        }
    }

    private boolean requiresHeuristic(int algorithmChoice) {
        // UCS doesn't use heuristic, others do
        return algorithmChoice != 1;
    }

          private  void  setAlgorithmHeuristic(SearchAlgorithm  algorithm,  int
heuristicChoice) {
        if (algorithm instanceof GBFS) {
            ((GBFS) algorithm).setHeuristicType(heuristicChoice);
        } else if (algorithm instanceof AStar) {
            ((AStar) algorithm).setHeuristicType(heuristicChoice);
        } else if (algorithm instanceof IDAStar) {
            ((IDAStar) algorithm).setHeuristicType(heuristicChoice);
        }
    }

    private void saveResultsOption(SearchAlgorithm algorithm) {
                                if    (algorithm.getSolution()    ==    null    ||
algorithm.getSolution().isEmpty()) {
            return;
        }

          String input = RushHourIO.getUserInput("Save solution to file? (y/n):
");

        if (input.toLowerCase().startsWith("y")) {
            try {
                        String filenameInputted = RushHourIO.getUserInput("Enter
filename (without extension): ");
                String filename = "output/" + filenameInputted + ".txt";
                // Prepare solution and actions with final step
                                List<RushHourGame>  solutionWithExit  =  new
ArrayList<>(algorithm.getSolution());
                                        List<String>  actionsWithExit  =  new
ArrayList<>(algorithm.getSolutionActions());
                                            RushHourGame  finalState  =  new
RushHourGame(solutionWithExit.get(solutionWithExit.size() - 1));
                finalState.removePrimaryPiece();
                solutionWithExit.add(finalState);
                actionsWithExit.add("EXIT");
                                RushHourIO.writeSolutionToFile(solutionWithExit,
actionsWithExit, algorithm, filename);
                System.out.println("Solution saved to: " + filename);
            } catch (IOException e) {
                System.err.println("Error saving file: " + e.getMessage());
            }
        }
    }

    private void saveComparisonOption() {
```

```
        String input = RushHourIO.getUserInput("Save comparison results to file?
(y/n): ");

        if (input.toLowerCase().startsWith("y")) {
              // This would require additional implementation to save comparison
results
              System.out.println("Comparison save feature will be implemented in
future version.");
        }
    }

    public RushHourGame loadGameFromFile(String filename) throws IOException {
        this.initialGame = RushHourIO.loadGameFromFile(filename);
        return this.initialGame;
    }

    public List<RushHourGame> solve(String algorithmType, int heuristicType) {
        if (initialGame == null) {
                        throw  new  IllegalStateException("No  game  loaded.  Call
loadGameFromFile first.");
        }

        SearchAlgorithm algorithm;

        switch (algorithmType.toUpperCase()) {
            case "UCS":
                algorithm = new UCS();
                break;
            case "GBFS":
                algorithm = new GBFS(heuristicType);
                break;
            case "A*":
                algorithm = new AStar(heuristicType);
                break;
            case "IDA*":
                algorithm = new IDAStar(heuristicType);
                break;
            default:
                     throw new IllegalArgumentException("Unknown algorithm: " +
algorithmType);
        }

        RushHourGame gameCopy = new RushHourGame(initialGame);
        return algorithm.solve(gameCopy);
    }

    public Map<String, Map<String, Object>> getResults() {
        return new HashMap<>(results);
    }
}
```

4.10 SearchAlgoritm.java

```
package src;

import java.util.*;
```

```java
public abstract class SearchAlgorithm {
    protected int nodesVisited;
    protected long executionTime;
    protected List<RushHourGame> solution;
    protected List<String> solutionActions;

    public abstract List<RushHourGame> solve(RushHourGame initial);

    public int getNodesVisited() {
        return nodesVisited;
    }


    public long getExecutionTime() {
        return executionTime;
    }

    public List<RushHourGame> getSolution() {
        return solution;
    }

    public List<String> getSolutionActions() {
        return solutionActions;
    }

    protected void resetCounters() {
        nodesVisited = 0;
        executionTime = 0;
        solution = null;
        solutionActions = null;
    }


    protected List<RushHourGame> buildSolution(SearchNode goalNode) {
        List<RushHourGame> path = new ArrayList<>();
        List<String> actions = new ArrayList<>();
        SearchNode current = goalNode;

        while (current != null) {
            path.add(0, current.state);
            if (current.action != null && !current.action.equals("Initial")) {
                actions.add(0, current.action);
            }
            current = current.parent;
        }

        solutionActions = actions;
        return path;
    }

    protected String getAction(RushHourGame from, RushHourGame to) {
    // Find which piece moved
    for (char piece : from.getPieces().keySet()) {
        List<int[]> fromPos = from.getPieces().get(piece);
```

```
        List<int[]> toPos = to.getPieces().get(piece);

          // Handle case where piece might have exited (toPos could be null or
different size)
        if (toPos == null) {
             // Piece has completely exited the board (only primary piece can do
this)
            if (piece == 'P') {
                // Determine exit direction based on exit position
                      if (to.getExitCol() == to.getCols() && to.getExitRow() ==
from.getPrimaryRow()) {
                    return piece + "-kanan"; // Exited to the right
                } else if (to.getExitRow() == to.getRows() && to.getExitCol() ==
from.getPrimaryCol()) {
                    return piece + "-bawah"; // Exited down
                }
                // Add more exit directions if needed
            }
            continue;
        }

        if (fromPos.size() != toPos.size()) {
            // Size difference might indicate partial exit or other issue
            if (piece == 'P' && toPos.size() < fromPos.size()) {
                // Primary piece is partially exiting
                // Determine direction based on remaining positions
                if (fromPos.get(0)[0] == toPos.get(0)[0]) {
                    // Horizontal movement
                       return from.getExitCol() > from.getPrimaryCol() ? piece +
"-kanan" : piece + "-kiri";
                } else {
                    // Vertical movement
                       return from.getExitRow() > from.getPrimaryRow() ? piece +
"-bawah" : piece + "-atas";
                }
            }
            continue;
        }

        boolean moved = false;
        for (int i = 0; i < fromPos.size(); i++) {
               if (fromPos.get(i)[0] != toPos.get(i)[0] || fromPos.get(i)[1] !=
toPos.get(i)[1]) {
                moved = true;
                break;
            }
        }

        if (moved) {
            // Determine direction of movement
            int[] oldPos = fromPos.get(0);
            int[] newPos = toPos.get(0);

            if (oldPos[0] != newPos[0]) {
                // Vertical movement
                if (newPos[0] > oldPos[0]) {
```

```
                    return piece + "-bawah";
                } else {
                    return piece + "-atas";
                }
            } else {
                // Horizontal movement
                if (newPos[1] > oldPos[1]) {
                    return piece + "-kanan";
                } else {
                    return piece + "-kiri";
                }
            }
        }
    }

    return "Unknown";
}

     protected SearchNode findInFrontierWithHigherCost(PriorityQueue<SearchNode>
frontier,
                                                                RushHourGame state, int
newCost) {
        for (SearchNode n : frontier) {
            if (n.state.equals(state) && n.g > newCost) {
                return n;
            }
        }
        return null;
    }

       protected  SearchNode  findInFrontier(PriorityQueue<SearchNode>  frontier,
RushHourGame state) {
        for (SearchNode n : frontier) {
            if (n.state.equals(state)) {
                return n;
            }
        }
        return null;
    }

    public void printSolution() {
        if (solution == null || solution.isEmpty()) {
            System.out.println("No solution found!");
            return;
        }

        System.out.println("Solution found!");
        System.out.println("Nodes visited: " + nodesVisited);
        System.out.println("Execution time: " + executionTime + " ms");
           System.out.println("Solution  length:  "  +  (solution.size()  -  1)  +  "
moves");
        System.out.println();

        // Print initial state
        System.out.println("Papan Awal");
        System.out.println(solution.get(0));
```

```
        // Print each move
        for (int i = 1; i < solution.size(); i++) {
             System.out.println("Gerakan " + i + ": " + solutionActions.get(i -
1));
            System.out.println(solution.get(i));
        }
    }

    public abstract String getAlgorithmName();

    public abstract String getAlgorithmDescription();
}

class SearchNode implements Comparable<SearchNode> {
    public RushHourGame state;
    public SearchNode parent;
    public int g; // Cost from start
    public int h; // Heuristic value
    public int f; // Total cost (g + h)
    public String action; // Action taken to reach this state

      public SearchNode(RushHourGame state, SearchNode parent, int g, int h,
String action) {
        this.state = state;
        this.parent = parent;
        this.g = g;
        this.h = h;
        this.f = g + h;
        this.action = action;
    }

    @Override
    public int compareTo(SearchNode other) {
        if (this.f != other.f) {
            return Integer.compare(this.f, other.f);
        }
        // Tie-breaking: prefer node with lower h value
        return Integer.compare(this.h, other.h);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof SearchNode)) return false;
        SearchNode other = (SearchNode) obj;
        return state.equals(other.state);
    }

    @Override
    public int hashCode() {
        return state.hashCode();
    }
}
```

4.11 UCS.java

```
package src;

import java.util.*;

public class UCS extends SearchAlgorithm {

    @Override
    public List<RushHourGame> solve(RushHourGame initial) {
        resetCounters();
        long startTime = System.currentTimeMillis();

        // Priority queue ordered by g value (path cost)
                PriorityQueue<SearchNode>  frontier  =  new  PriorityQueue<>(new
Comparator<SearchNode>() {
            @Override
            public int compare(SearchNode n1, SearchNode n2) {
                if (n1.g != n2.g) {
                    return Integer.compare(n1.g, n2.g);
                }
                // Tie-breaking: prefer older nodes (FIFO)
                return 0;
            }
        });

        Set<RushHourGame> explored = new HashSet<>();

        // Initialize with starting state
        frontier.add(new SearchNode(initial, null, 0, 0, "Initial"));

        while (!frontier.isEmpty()) {
            SearchNode node = frontier.poll();
            nodesVisited++;

            // Check if goal state reached
            if (node.state.isGoalState()) {
                executionTime = System.currentTimeMillis() - startTime;
                solution = buildSolution(node);
                return solution;
            }

            explored.add(node.state);

            // Expand current node
            for (RushHourGame nextState : node.state.getNextStates()) {
                if (!explored.contains(nextState)) {
                    // Calculate new cost (each move has cost 1)
                    int newCost = node.g + 1;

                    // Check if state is already in frontier
                             SearchNode  existingNode  =  findInFrontier(frontier,
nextState);

                    if (existingNode == null) {
                        // Add new node to frontier
```

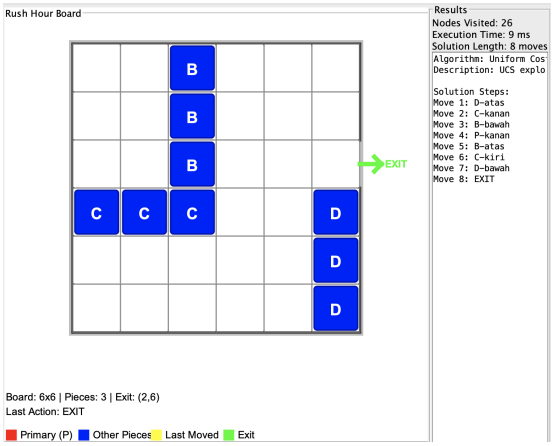
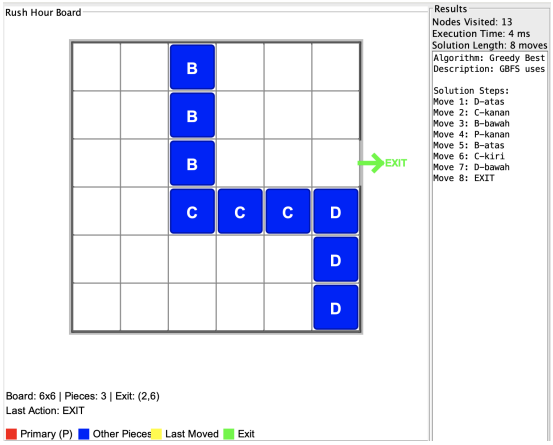
```
                        String action = getAction(node.state, nextState);
                        frontier.add(new SearchNode(nextState, node, newCost, 0,
action));
                    } else if (newCost < existingNode.g) {
                        // Found better path to existing node
                        frontier.remove(existingNode);
                        String action = getAction(node.state, nextState);
                        frontier.add(new SearchNode(nextState, node, newCost, 0,
action));
                    }
                }
            }
        }

        executionTime = System.currentTimeMillis() - startTime;
        return null; // No solution found
    }

    @Override
    public String getAlgorithmName() {
        return "Uniform Cost Search (UCS)";
    }

    @Override
    public String getAlgorithmDescription() {
          return "UCS explores nodes in order of their path cost from the start
node. " +
                  "It guarantees finding the optimal solution (minimum number of
moves) " +
                "because each move has the same cost.";
    }

    @Override
       protected  SearchNode  findInFrontier(PriorityQueue<SearchNode>  frontier,
RushHourGame state) {
        for (SearchNode n : frontier) {
            if (n.state.equals(state)) {
                return n;
            }
        }
        return null;
    }

    public Map<String, Object> getStatistics() {
        Map<String, Object> stats = new HashMap<>();
        stats.put("algorithm", getAlgorithmName());
        stats.put("nodes_visited", nodesVisited);
        stats.put("execution_time", executionTime);
          stats.put("solution_length", solution != null ? solution.size() - 1 :
-1);
         stats.put("optimal", true); // UCS always finds optimal solution if one
exists
        return stats;
    }

    @Override
```

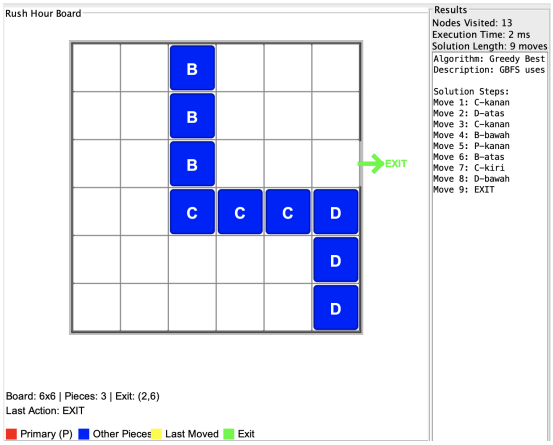
```
    public void printSolution() {
        System.out.println("=== " + getAlgorithmName() + " Results ===");
        System.out.println(getAlgorithmDescription());
        System.out.println();

        if (solution == null || solution.isEmpty()) {
            System.out.println("No solution found!");
            System.out.println("Nodes visited: " + nodesVisited);
            System.out.println("Execution time: " + executionTime + " ms");
            return;
        }

        System.out.println("Solution found!");
        System.out.println("Nodes visited: " + nodesVisited);
        System.out.println("Execution time: " + executionTime + " ms");
           System.out.println("Solution length: " + (solution.size() - 1) + "
moves");
        System.out.println("Status: OPTIMAL (UCS guarantees optimal solution)");
        System.out.println();

        // Print solution path
        System.out.println("Papan Awal");
        printColoredBoard(solution.get(0), null);
        System.out.println();

        // Print each move
        for (int i = 1; i < solution.size(); i++) {
             System.out.println("Gerakan " + i + ": " + solutionActions.get(i -
1));
            printColoredBoard(solution.get(i), solutionActions.get(i - 1));
            System.out.println();
        }
    }

    private void printColoredBoard(RushHourGame game, String action) {
        char[][] board = game.getBoard();
        int rows = game.getRows();
        int cols = game.getCols();

        // ANSI color codes
        String RESET = "\u001B[0m";
        String RED = "\u001B[31m";      // Primary piece
        String GREEN = "\u001B[32m";    // Exit
        String YELLOW = "\u001B[33m";   // Moved piece
        String BLUE = "\u001B[34m";     // Other pieces

        char movedPiece = '.';
        if (action != null && !action.equals("Initial")) {
            movedPiece = action.charAt(0);
        }

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                char c = board[i][j];

                if (c == game.getPrimaryPiece()) {
```

```
                    System.out.print(RED + c + RESET);
                } else if (c == game.getExit()) {
                    System.out.print(GREEN + c + RESET);
                } else if (c == movedPiece && c != '.') {
                    System.out.print(YELLOW + c + RESET);
                } else if (c != '.' && c != game.getExit()) {
                    System.out.print(BLUE + c + RESET);
                } else {
                    System.out.print(c);
                }
            }
            System.out.println();
        }
    }

    public boolean isEquivalentToBFS() {
        // In Rush Hour, all moves have the same cost (1), so UCS will explore
        // nodes in the same order as BFS (breadth-first order)
        return true;
    }
}
```

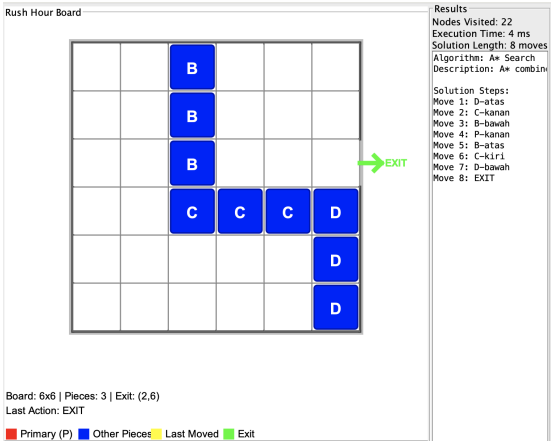
BAB V

MASUKAN DAN LUARAN PROGRAM

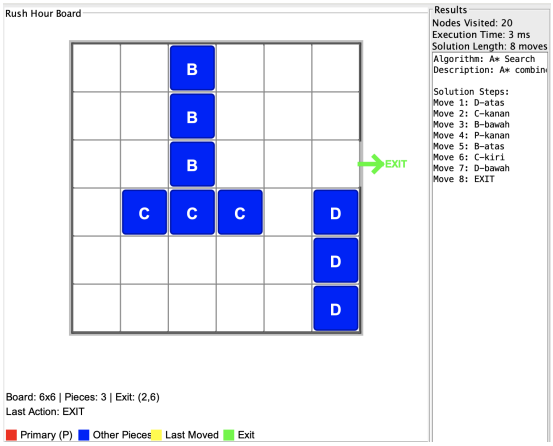
INPUT	OUTPUT
5.1 Isi test1.txt 6 6 3 ..B... ..B... PPB...K CCC..D .....D .....D	**UCS**  **GBFS-H1**  **GBFS-H2**



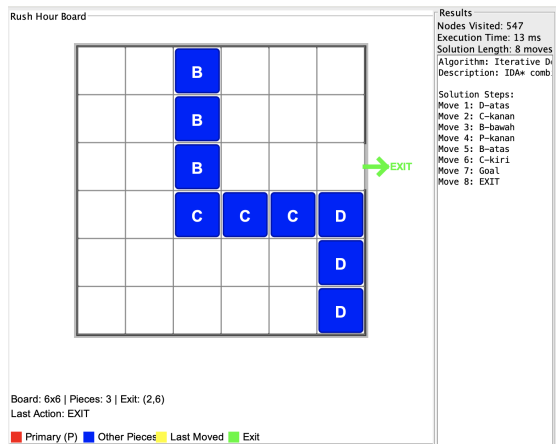
A*-H1



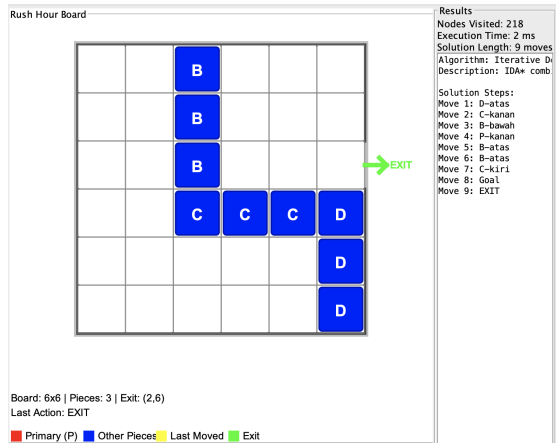
A*-H2



IDA*-H1



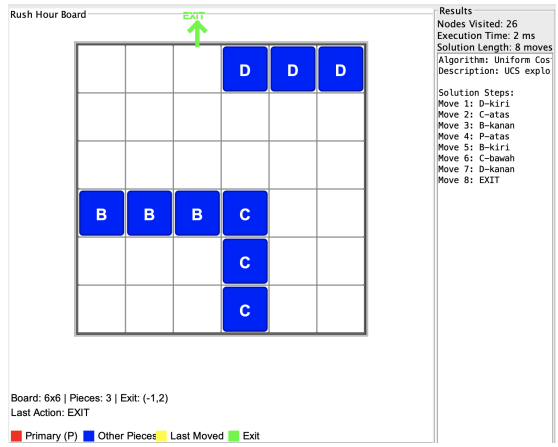
IDA*-H2



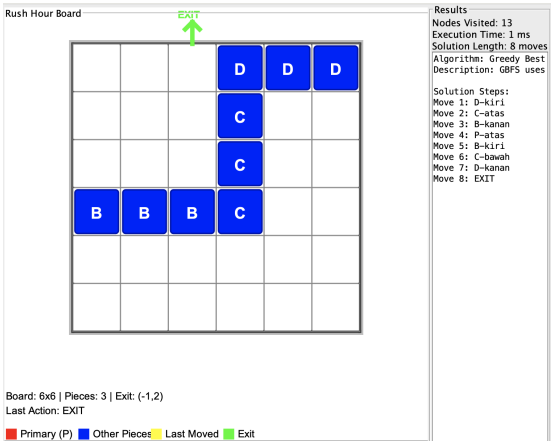
5.2 Isi test2.txt

6 6
3
  K
...DDD
.....
.....
BBBC..
..PC..
..PC..

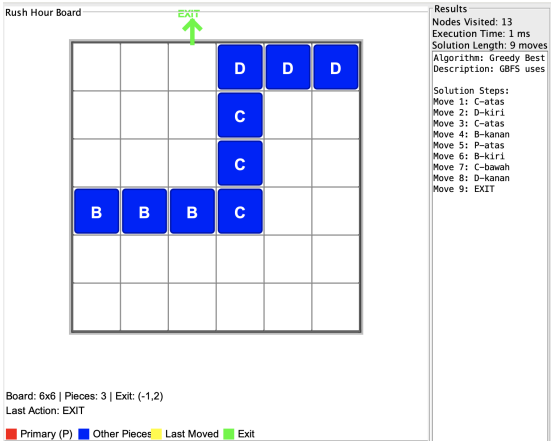
UCS



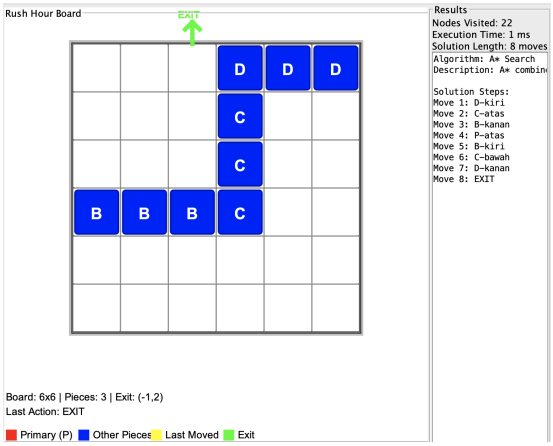
GBFS-H1



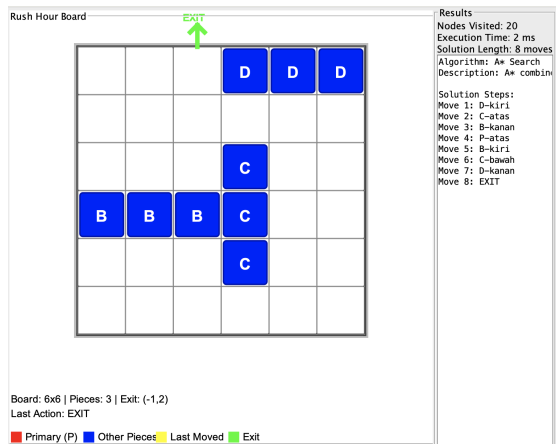
GBFS-H2



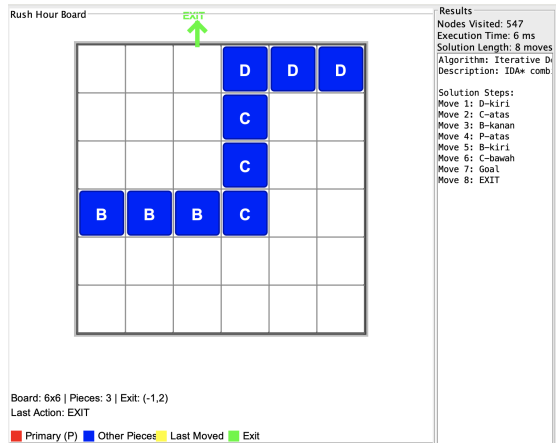
A*-H1



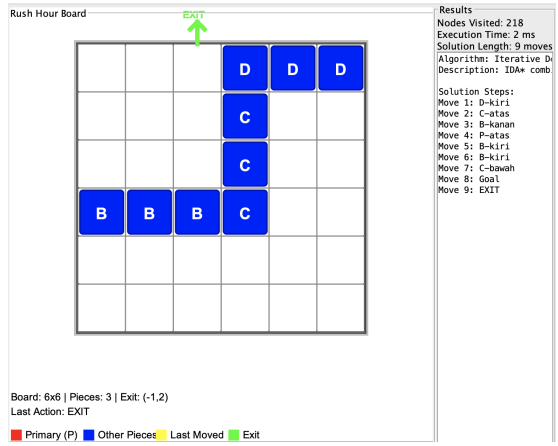
A*-H2



IDA*-H1



IDA*-H2

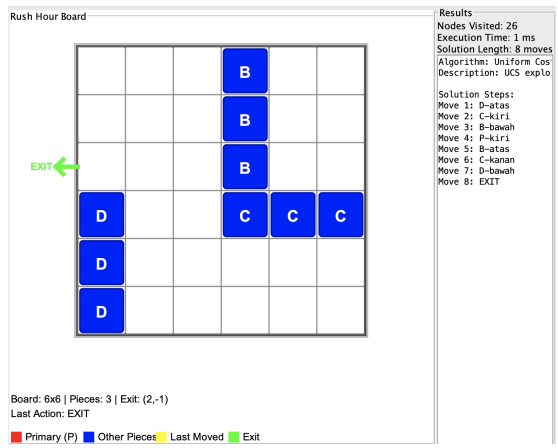


5.3 Isi test3.txt

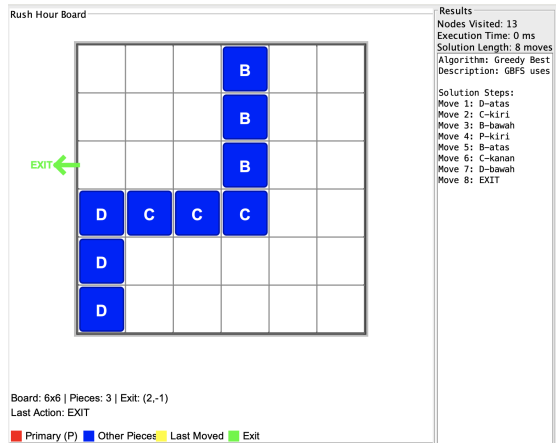
6 6
3
...B..

UCS

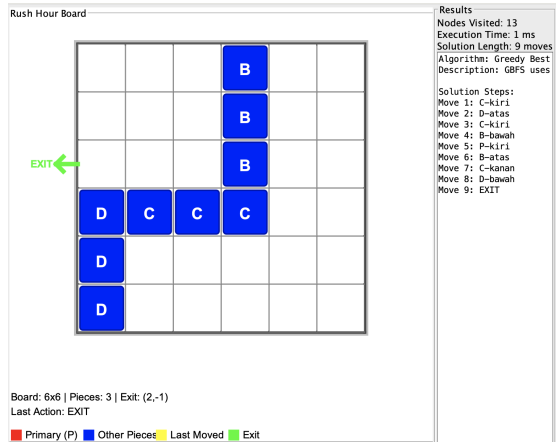
...B..
K...BPP
D..CCC
D.....
D.....



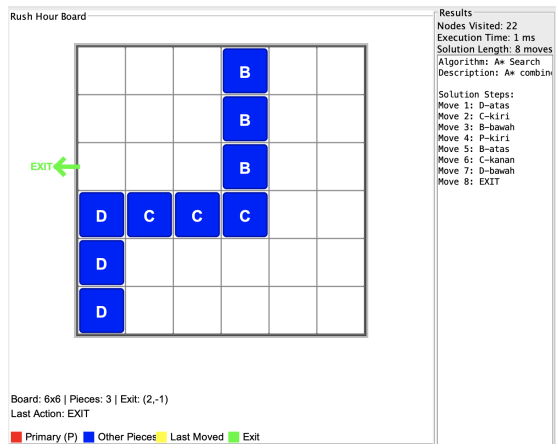
GBFS-H1



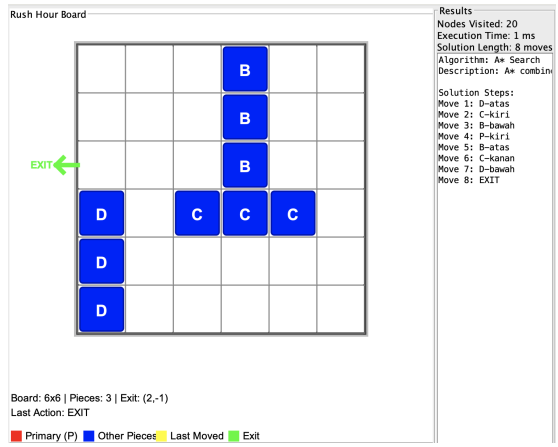
GBFS-H2



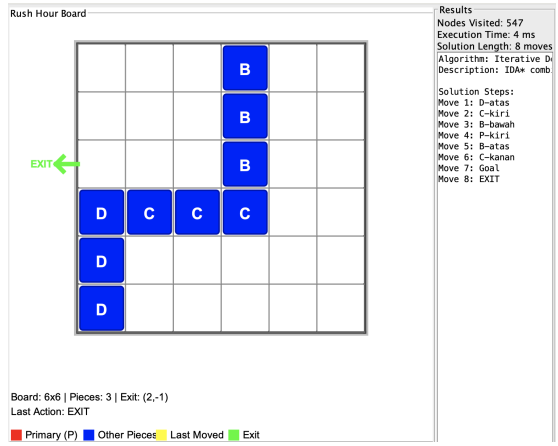
A*-H1



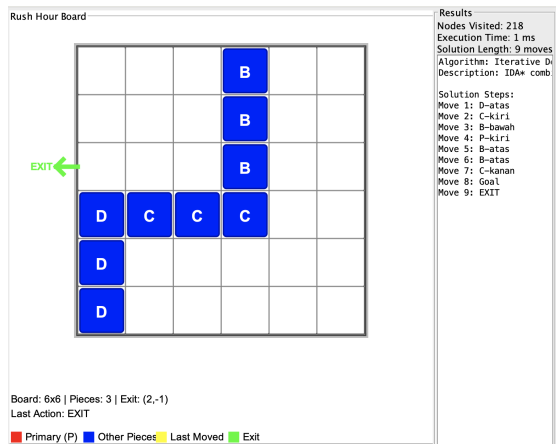
A*-H2



IDA*-H1



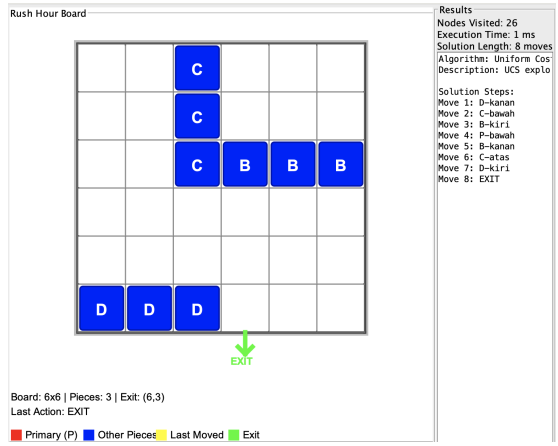
IDA*-H2



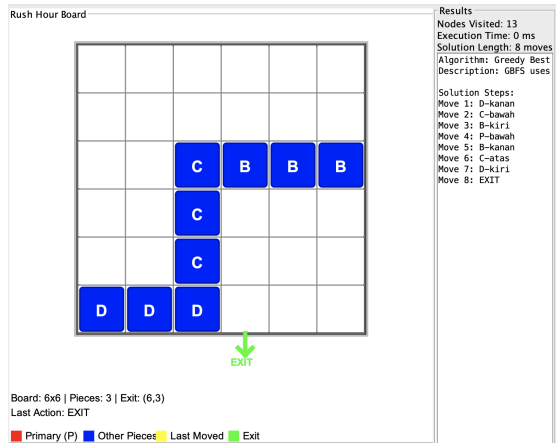
5.4 Isi test4.txt

6 6
3
..CP..
..CP..
..CBBB
.....
.....
DDD...
  K

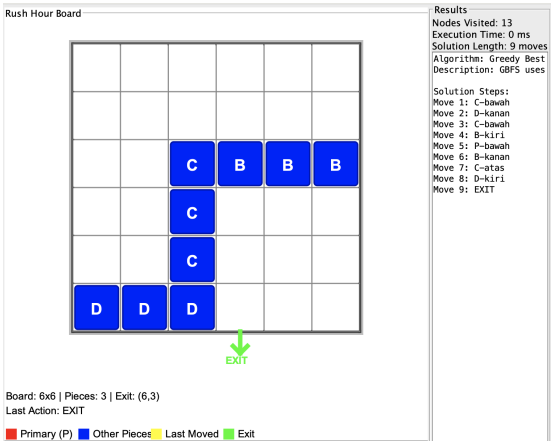
UCS



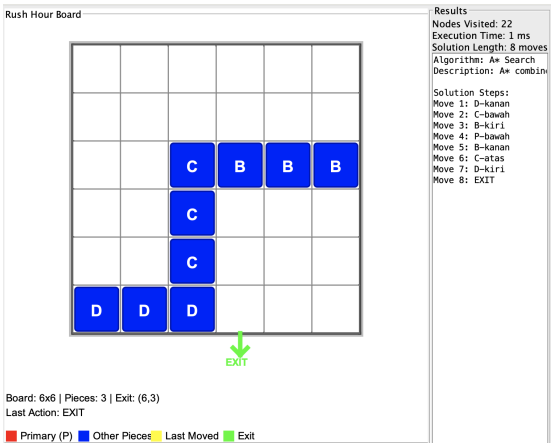
GBFS-H1



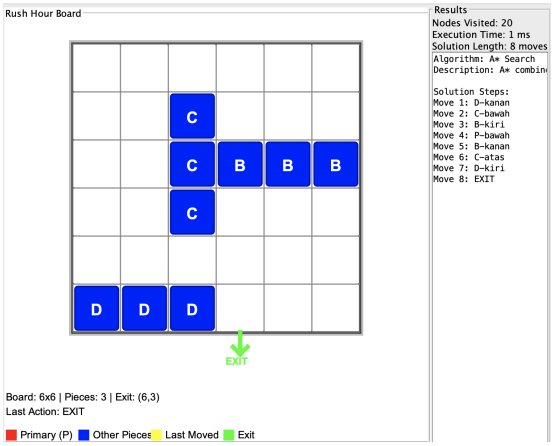
GBFS-H2



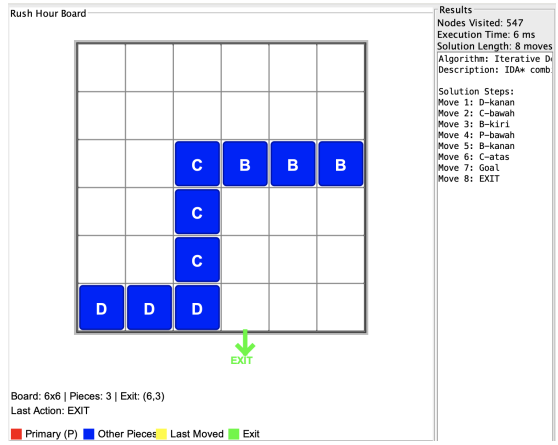
A*-H1



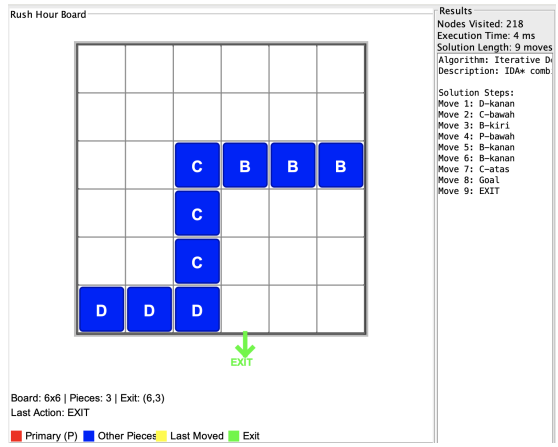
A*-H2



IDA*-H1

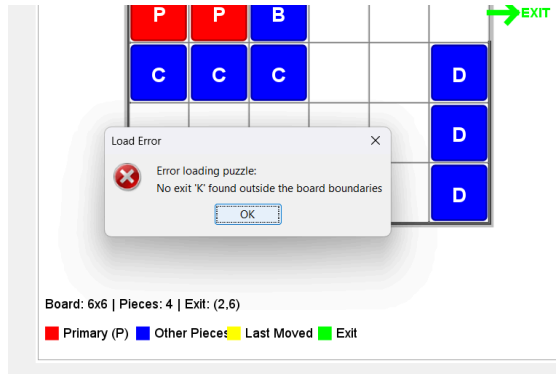


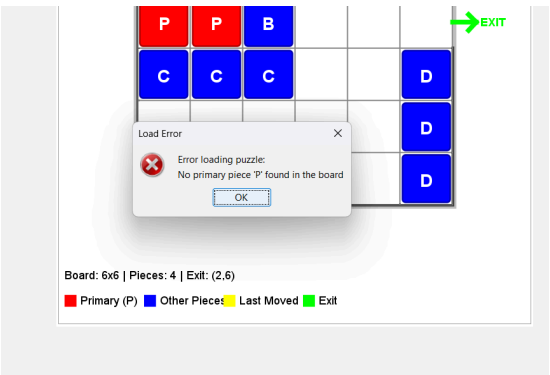
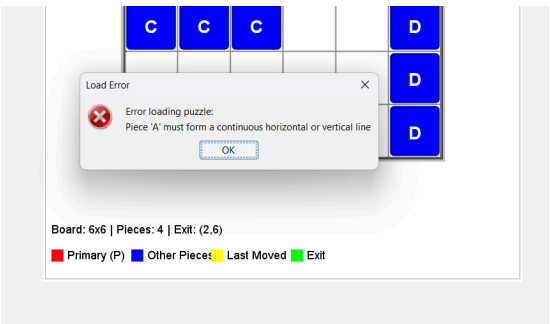
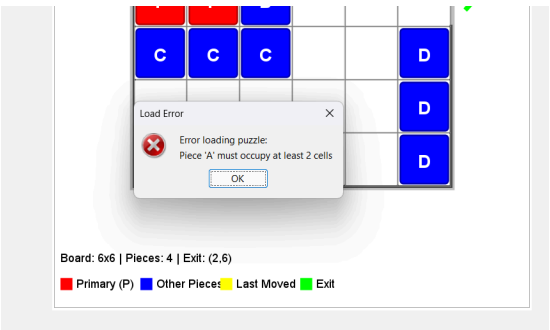
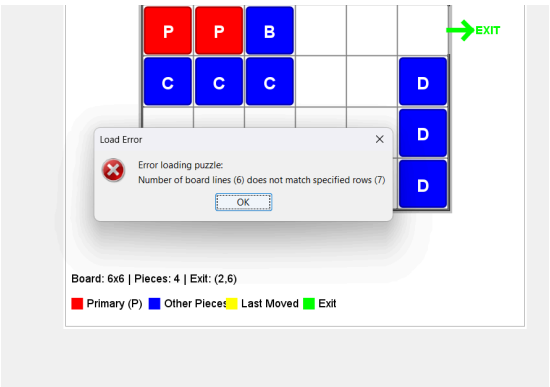
IDA*-H2



5.5 Isi test5.txt

6 6
11
AAB..F
..BCDF
GPPCDF
GH.III
GHJ...
LLJMM.



5.6 Isi test6.txt 6 6 11 AAB..F ..BCDF GEECDFK GH.III GHJ... LLJMM.	
5.7 Isi test7.txt 6 6 11 ABB..F ..ACDF GPPCDFK GH.III GHJ... LLJMM.	
5.8 Isi test8.txt 6 6 11 ABB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	
5.9 Isi test9.txt 7 6 3 ..B... ..B... PPB...K CCC..D .....D .....D	

BAB VI
ANALISIS IMPLEMENTASI

Berdasarkan percobaan yang telah dilakukan menggunakan empat algoritma pathfinding (UCS, GBFS, A*, dan IDA*) dengan dua heuristik berbeda (Manhattan Distance/H1 dan Blocking Pieces + Distance/H2) pada beberapa kasus uji Rush Hour Puzzle, terdapat beberapa pola yang terlihat.

Algoritma Uniform Cost Search (UCS) konsisten dalam menemukan solusi optimal, sebagaimana diharapkan dari sifat teoritisnya. Namun, UCS menunjukkan kelemahan dalam efisiensi pencarian untuk puzzle yang kompleks, dimana algoritma ini mengeksplorasi node dalam jumlah cukup signifikan dibandingkan dengan algoritma berbasis heuristik seperti GBFS. Meskipun demikian, waktu eksekusi UCS masih relatif kompetitif, terutama untuk puzzle dengan solusi sederhana.

Greedy Best-First Search (GBFS) dengan kedua jenis heuristik menunjukkan keunggulan signifikan dalam efisiensi eksplorasi, mengunjungi jauh lebih sedikit node dibandingkan UCS dan A*. Secara konsisten, GBFS dengan heuristik Manhattan Distance (H1) mengeksplorasi node lebih sedikit dibandingkan dengan heuristik Blocking Pieces (H2), menunjukkan bahwa heuristik sederhana sering kali lebih efisien untuk pencarian cepat. Meskipun kecepatan yang unggul, GBFS gagal menemukan solusi optimal dalam semua kasus uji, yang sesuai dengan karakteristik teoretisnya yang mengorbankan optimalitas demi efisiensi.

Algoritma A* memberikan keseimbangan antara UCS dan GBFS. Dengan heuristik Manhattan Distance (H1) yang admissible, A* berhasil menemukan solusi optimal dalam semua kasus, sambil menjaga jumlah node yang dikunjungi tetap jauh lebih rendah dibandingkan UCS. Sementara itu, A* dengan heuristik Blocking Pieces (H2) mengunjungi node lebih sedikit dibandingkan A*-H1, tetapi tidak selalu menghasilkan solusi optimal, mengkonfirmasi bahwa H2 tidak bersifat admissible. Secara keseluruhan, A* menawarkan kompromi yang efektif antara optimalitas dan efisiensi komputasi.

IDA* menunjukkan pola kinerja yang menarik. Meskipun jumlah node yang dikunjungi sangat tinggi (jauh melebihi algoritma lain), waktu eksekusinya masih relatif kompetitif berkat efisiensi memorinya. IDA* dengan heuristik H1 menjamin solusi optimal, tetapi dengan biaya komputasi yang lebih tinggi. Menariknya, IDA* dengan H2 mengunjungi node secara signifikan lebih sedikit dibandingkan dengan IDA*-H1, namun tetap gagal menjamin optimalitas, konsisten dengan karakteristik heuristik non-admissible.

Dari segi analisis kompleksitas waktu, keempat algoritma dapat dianalisis menggunakan pendekatan rekursif dan Teorema Master. Untuk UCS, GBFS, dan A*, persamaan rekursif $T(n) = b \times T(n-1) + O(1)$ dapat diterapkan, dimana b adalah faktor percabangan rata-rata dan kedalaman

rekursi adalah d (kedalaman solusi). Menggunakan Teorema Master, solusi untuk persamaan ini adalah $T(n) = O(b^d)$, mengindikasikan kompleksitas waktu eksponensial terhadap kedalaman solusi. Hal ini terkonfirmasi dalam percobaan dimana waktu eksekusi meningkat signifikan untuk puzzle dengan solusi yang lebih panjang atau kompleks.

Untuk IDA*, persamaan rekursif menjadi lebih kompleks: $T(n) = \Sigma(i=0 \text{ to } d)\ b^i$, karena algoritma melakukan multiple iterasi dengan batas kedalaman yang meningkat. Dalam kasus terburuk, ini meningkat menjadi $O(b^d)$, namun dengan overhead konstanta yang lebih besar dibandingkan algoritma lain karena adanya eksplorasi ulang node. Inilah yang menjelaskan mengapa IDA* mengunjungi node dalam jumlah jauh lebih banyak dibandingkan algoritma lain dalam percobaan kita, meskipun waktu eksekusinya tidak sebanding dengan peningkatan jumlah node tersebut.

Untuk kompleksitas ruang, UCS, GBFS, dan A* memiliki kebutuhan memori $O(b^d)$ karena mereka harus menyimpan semua node yang telah dibangkitkan namun belum dieksplorasi dalam struktur data frontier (priority queue). Hal ini terjadi karena dalam skenario terburuk, hampir semua node di level d dapat berada dalam frontier sebelum menemukan solusi. Semakin besar faktor percabangan b dan semakin dalam solusi d, semakin besar kebutuhan memori, yang bisa menjadi kendala serius pada puzzle kompleks dengan banyak state.

Sebaliknya, IDA* menunjukkan keunggulan dengan kompleksitas ruang hanya $O(d)$. Hal ini karena IDA* mengimplementasikan pencarian depth-first yang hanya menyimpan node-node dalam jalur eksplorasi saat ini, dengan maksimal d node (kedalaman solusi) yang disimpan secara bersamaan dalam memori. Ini menjelaskan mengapa IDA* tetap dapat menangani puzzle dengan ruang pencarian yang sangat besar meskipun dengan jumlah eksplorasi node yang tinggi. Dalam implementasi program kita, keuntungan ini tidak terlihat jelas karena puzzle uji masih dalam skala moderat, namun akan sangat signifikan untuk puzzle Rush Hour dengan banyak kendaraan dan ruang pencarian yang jauh lebih besar.

Terkait dengan panjang solusi, terlihat bahwa algoritma yang menjamin optimalitas (UCS, A*-H1, dan IDA*-H1) konsisten menghasilkan jumlah langkah yang sama dan minimal. Di sisi lain, algoritma dengan heuristik non-admissible atau yang tidak menjamin optimalitas (GBFS-H1, GBFS-H2, A*-H2, IDA*-H2) cenderung menghasilkan solusi dengan jumlah langkah yang bervariasi dan seringkali lebih panjang.

Kesimpulannya, tidak ada satu algoritma yang unggul dalam semua aspek. UCS menjamin optimalitas tetapi kurang efisien, GBFS sangat efisien tetapi tidak menjamin optimalitas, A* dengan heuristik admissible memberikan keseimbangan optimal antara efisiensi dan optimalitas, dan IDA* menawarkan solusi hemat memori untuk puzzle yang sangat kompleks. Pemilihan algoritma sebaiknya didasarkan pada kebutuhan spesifik aplikasi, dengan mempertimbangkan trade-off antara optimalitas, efisiensi komputasi, dan ketersediaan memori.

BAB VII
IMPLEMENTASI BONUS

8.1 Implementasi Algoritma Pathfinding Alternatif (IDA*)

Dalam implementasi Rush Hour Puzzle Solver ini, algoritma Iterative Deepening A* (IDA*) telah berhasil diimplementasikan sebagai algoritma pathfinding alternatif selain UCS, GBFS, dan A*. IDA* menggabungkan prinsip Iterative Deepening Search dengan heuristik dari A* untuk mengatasi keterbatasan memori A* konvensional. Dalam kode sumber yang dikembangkan, algoritma IDA* diimplementasikan dalam kelas IDAStar.java yang mewarisi kelas abstrak SearchAlgorithm. Implementasi ini menggunakan pendekatan iteratif dengan ambang batas (threshold) yang dimulai dari nilai heuristik node awal dan terus meningkat pada setiap iterasi. Kelas ini menggunakan metode depthLimitedSearch() untuk melakukan pencarian depth-first dengan batasan nilai f(n) pada setiap iterasi. Algoritma IDA* telah terbukti sangat berguna dalam konteks Rush Hour puzzle, terutama untuk konfigurasi kompleks dengan banyak mobil dan ruang state yang besar, karena kebutuhan memorinya hanya O(d) dibandingkan dengan O(b^d) pada A*, dimana d adalah kedalaman solusi dan b adalah faktor percabangan. Untuk penjelasan lebih rinci, tertera pada Bab II.

8.2 Implementasi Heuristik Alternatif

Proyek ini juga mengimplementasikan dua heuristik alternatif untuk algoritma pencarian terinformasi (GBFS, A*, dan IDA*). Heuristik pertama adalah Manhattan Distance (H1) yang menghitung jarak langsung dari primary piece ke pintu keluar. Dalam kode program, heuristik ini diimplementasikan dalam metode getHeuristic1() di kelas RushHourGame. Heuristik ini mempertimbangkan orientasi primary piece (horizontal atau vertikal) dan menghitung jarak terpendek yang harus ditempuh untuk mencapai pintu keluar, tanpa mempertimbangkan penghalang. Heuristik kedua adalah Blocking Pieces + Distance (H2) yang diimplementasikan dalam metode getHeuristic2(). Heuristik ini menggabungkan jarak ke pintu keluar dengan faktor penalti untuk setiap piece yang menghalangi jalur langsung primary piece ke pintu keluar. Perbedaan utama dari kedua heuristik ini adalah H1 bersifat admissible (tidak pernah overestimate) sehingga menjamin solusi optimal dengan A* dan IDA*, sementara H2 mungkin tidak admissible karena penalti untuk piece penghalang, tetapi sering kali lebih efisien dalam mengarahkan pencarian. Keduanya dapat dipilih oleh pengguna melalui antarmuka GUI, memungkinkan eksperimentasi untuk menemukan pendekatan yang paling efisien untuk berbagai jenis puzzle. Untuk penjelasan lebih rinci, tertera pada Bab II.

8.3 Implementasi Graphical User Interface (GUI)

Implementasi GUI telah berhasil dilakukan untuk meningkatkan pengalaman dalam menyelesaikan Rush Hour puzzle. Kelas utama RushHourGUI.java menggunakan framework Swing untuk menyediakan antarmuka grafis yang interaktif dan intuitif. GUI ini terdiri dari

beberapa komponen utama: panel permainan (GamePanel) untuk visualisasi papan Rush Hour, panel kontrol yang memungkinkan pengguna memilih algoritma pencarian dan heuristik, panel animasi untuk memvisualisasikan solusi secara bertahap, dan panel hasil untuk menampilkan statistik dan langkah-langkah solusi. Fitur-fitur utama GUI meliputi kemampuan untuk memuat puzzle dari file, memilih algoritma pencarian (UCS, GBFS, A*, atau IDA*) dan heuristik yang digunakan (jika berlaku), serta kontrol animasi yang komprehensif (play, pause, step forward, step backward) untuk memvisualisasikan solusi. Implementasi GUI juga mencakup fungsionalitas untuk menampilkan statistik penting seperti jumlah node yang dikunjungi, waktu eksekusi, dan panjang solusi, memberikan wawasan yang berharga tentang kinerja berbagai algoritma. Animasi solusi ditangani oleh kelas GameAnimation yang memungkinkan visualisasi langkah demi langkah dari solusi yang ditemukan, memperjelas bagaimana algoritma pathfinding bekerja dalam konteks visual.

BAB VIII
KESIMPULAN

Dalam pembuatan Rush Hour Puzzle menggunakan algoritma pathfinding, kami telah berhasil mengimplementasikan dan menganalisis empat algoritma berbeda: Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), A*, dan Iterative Deepening A* (IDA*), masing-masing dengan dua variasi heuristik. Berdasarkan hasil percobaan pada berbagai konfigurasi puzzle, kami menarik beberapa kesimpulan penting.

Pertama, kami menemukan bahwa algoritma yang berbeda menunjukkan trade-off yang jelas antara efisiensi pencarian dan optimalitas solusi. UCS dan A* dengan heuristik admissible (Manhattan Distance) secara konsisten menemukan solusi optimal, sementara GBFS menunjukkan efisiensi komputasi tertinggi dengan mengorbankan jaminan optimalitas. Hal ini menegaskan bahwa pilihan algoritma harus disesuaikan dengan prioritas aplikasi—apakah optimalitas lebih penting daripada kecepatan komputasi, atau sebaliknya.

Kedua, pemilihan heuristik memiliki dampak signifikan pada kinerja algoritma pencarian terinformasi. Heuristik Manhattan Distance (H1) yang lebih sederhana dan admissible terbukti menjamin solusi optimal untuk A* dan IDA*, sementara heuristik Blocking Pieces (H2) sering kali menghasilkan pencarian yang lebih efisien tetapi dengan solusi suboptimal. Ini menunjukkan pentingnya merancang heuristik yang tidak hanya informatif tetapi juga admissible untuk aplikasi yang membutuhkan solusi optimal.

Ketiga, analisis kompleksitas menunjukkan bahwa IDA* memiliki keunggulan signifikan dalam hal penggunaan memori dengan kompleksitas ruang O(d) dibandingkan dengan O(b^d) untuk algoritma lainnya. Meskipun IDA* mengunjungi lebih banyak node, pendekatan ini menjadi sangat berharga untuk puzzle dengan ruang pencarian yang sangat besar dimana memori menjadi kendala utama.

Sebagai kesimpulan akhir, tidak ada pendekatan "satu ukuran untuk semua" dalam penyelesaian Rush Hour Puzzle. A* dengan heuristik admissible menawarkan keseimbangan terbaik antara efisiensi dan optimalitas untuk sebagian besar kasus, tetapi GBFS lebih cocok ketika respons cepat diprioritaskan, dan IDA* unggul dalam penanganan puzzle dengan ruang pencarian yang sangat besar. Penelitian lanjutan dapat mengeksplorasi heuristik yang lebih canggih atau pendekatan hybrid yang dapat lebih efektif mengatasi variasi puzzle yang lebih kompleks.

BAB IX
LAMPIRAN

Pranala Github
https://github.com/sbimasena/Tucil3_13523053_13523118

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	V	
2. 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	
5. [Bonus] Implementasi algoritma pathfinding alternatif	V	
6. [Bonus] Implementasi 2 atau lebih heuristik alternatif	V	
7. [Bonus] Program memiliki GUI	V	