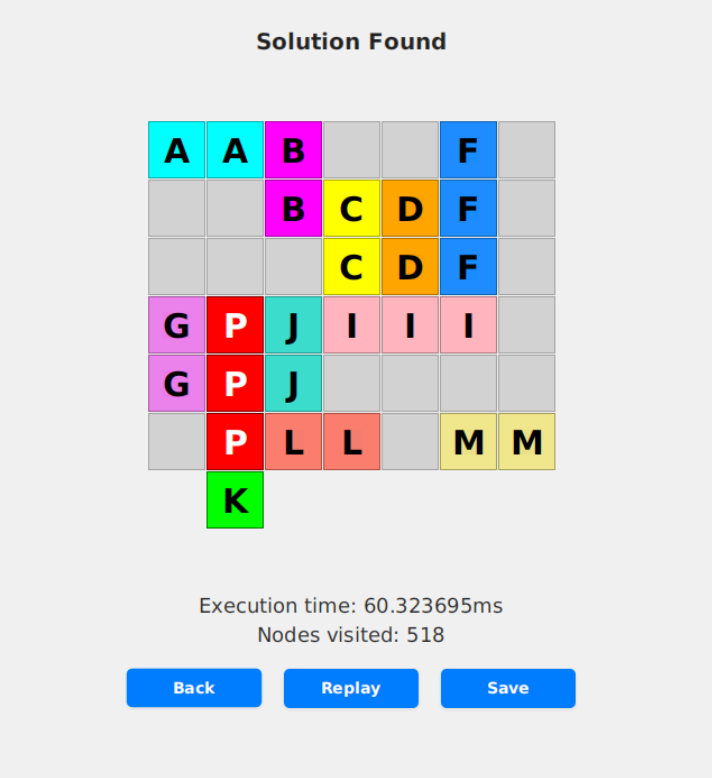
LAPORAN TUGAS KECIL 3
IF2211-Strategi Algoritma

Penerapan Algoritma GBFS, UCS, dan A pada Permainan Rush Hour*



Disusun oleh :

M Rayhan Farrukh	13523035
Ferdin Arsenarendra Purtadi	13523117

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

Daftar Isi

I. Deskripsi Masalah

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

Komponen penting dari permainan Rush Hour terdiri dari:

1. Papan – Papan merupakan tempat permainan dimainkan. Papan terdiri atas cell, yaitu sebuah singular point dari papan. Sebuah piece akan menempati cell-cell pada papan. Ketika permainan dimulai, semua piece telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi piece dan orientasi, antara horizontal atau vertikal. Hanya primary piece yang dapat digerakkan keluar papan melewati pintu keluar. Piece yang bukan primary piece tidak dapat digerakkan keluar papan. Papan memiliki satu pintu keluar yang pasti berada di dinding papan dan sejajar dengan orientasi primary piece.
2. Piece – Piece adalah sebuah kendaraan di dalam papan. Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah piece hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. Piece dapat memiliki beragam ukuran, yaitu jumlah cell yang ditempati oleh piece. Secara standar, variasi ukuran sebuah piece adalah 2-piece (menempati 2 cell) atau 3-piece (menempati 3 cell). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.
3. Primary Piece – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.
4. Pintu Keluar – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan
5. Gerakan — Gerakan yang dimaksudkan adalah pergeseran piece di dalam permainan. Piece hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu piece tidak dapat digerakkan melewati/menembus piece yang lain.

II. Analisis Algoritma

2.1 Pendahuluan

Pada program ini, ada 5 algoritma yang diimplementasikan, yaitu GBFS, UCS, A*, IDDFS, dan *Hill Climbing Search*. Dari 5 algoritma tersebut, 3 diantaranya (GBFS, A*, Hill Climbing) bersifat *informed search*, adapun UCS dan IDDFS bersifat *uninformed*. *Uninformed search,* berarti bahwa strategi pencarian algoritma tidak menggunakan informasi tambahan selain dari definisi masalah itu sendir. Adapun *informed search* berarti perhitungan pencarian dilakukan dengan memperhitungkan langkah-langkah berdasarkan nilai-nilai atau informasi tambahan yang diketahui sebelumnya, disebut dengan heuristik.

Heuristik yang digunakan pada program adalah heuristik berdasarkan kendaraan yang menghalangi (*blocking cars*), jarak ke jalan keluar (*manhattan distance to exit*), serta gabungan dari keduanya. Perhitungan heuristik dilakukan dengan sebuah fungsi evaluasi (f(n)), yaitu sebuah fungsi sebagai perhitungan estimasi total biaya. Untuk menghitung f(n) diperlukan nilai heuristik, yaitu g(n), dan h(n). g(n) adalah biaya aktual yang telah ditempuh dari simpul awal hingga saat ini. Adapun h(n) adalah biaya dari suatu simpul menuju simpul tujuan.

2.2 GBFS

Greedy Best First Search (GBFS) adalah algoritma *informed search* yang menggunakan pendekatan heuristik untuk memperkirakan seberapa dekat suatu state terhadap solusi. GBFS bersifat *informed search* karena menggunakan fungsi heuristic h(n) untuk menilai setiap node, namun tidak mempertimbangkan cost dari awal (g(n)).
Fungsi evaluasi dari GBFS adalah sebagai berikut:

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

Langkah-langkah algoritma GBFS:

- Sebuah priority queue disiapkan untuk menyimpan state, diurutkan berdasarkan nilai heuristic h(n) yang diberikan oleh fungsi heuristic.estimate(board). Kemudian state awal dimasukkan ke queue.
- Algoritma mengambil state dengan nilai heuristic terkecil dari queue (paling "dekat" ke goal menurut estimasi heuristic). Jika state tersebut merupakan solusi (primary piece bisa keluar), maka algoritma berhenti dan mengembalikan jalur solusi.
- Dari state sekarang, semua kemungkinan gerakan yang valid dihitung.Setiap hasil gerakan menghasilkan state baru dengan papan baru.
- Nilai heuristic untuk setiap state dihitung, dan state tersebut dimasukkan ke priority queue jika belum pernah dikunjungi sebelumnya.
- State yang memiliki representasi papan sama seperti sebelumnya tidak dimasukkan lagi ke antrian, untuk menghindari siklus atau perhitungan ulang yang tidak perlu.
- Proses berlanjut hingga ditemukan solusi atau queue habis (tidak ada solusi).

GBFS hanya mempertimbangkan nilai heuristic h(n) saat memilih state berikutnya. Namun, **tidak menjamin solusi optimal** karena tidak memperhitungkan total cost (g(n)). Bisa lebih cepat dari UCS atau A* karena eksplorasi lebih "agresif" menuju goal, namun bisa tersesat dalam jalur yang tampak menjanjikan tapi sebenarnya salah. Efisiensi algoritma ini sangat tergantung pada kualitas fungsi heuristik,.

2.3 UCS

Uniform Cost Search (UCS) adalah algoritma *uninformed search* yang bertujuan untuk menemukan solusi dengan biaya minimum dari titik awal menuju titik tujuan. UCS bekerja dengan meng-*expand* simpul berdasarkan *total cost* terkecil dari simpul awal, tanpa menggunakan heuristic. Hal ini membuat UCS serupa dengan algoritma Dijkstra.
Fungsi evaluasi dari UCS adalah sebagai berikut:

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

Langkah-langkah algoritma UCS:

- Sebuah *priority queue* disiapkan untuk menyimpan seluruh state papan, diurutkan berdasarkan *total cost*
- *State* awal dimasukkan ke dalam *queue* dengan cost = 0.
- Algoritma mengambil *state* dengan *cost* terendah dari *queue*.
- Jika *state* tersebut merupakan solusi, maka algoritma berhenti dan mengembalikan jalur solusi.
- Jika *state* belum mencapai tujuan, maka seluruh kemungkinan gerakan *piece* dari konfigurasi papan saat ini dihitung.
- Setiap gerakan menghasilkan *state* baru dengan *cost* bertambah satu, karena setiap langkah dianggap memiliki cost 1.
- Sebelum state baru dimasukkan ke *queue*, akan dicek apakah konfigurasi papan tersebut sudah pernah dikunjungi. Jika belum, maka ditambahkan ke *queue*.
- Proses berlanjut hingga *queue* kosong (tidak ada solusi) atau ditemukan *state* yang merupakan solusi.

Pada penyelesaian Rush Hour, UCS **bersifat sama** dengan BFS dalam hal simpul yang dibangkitkan serta hasil akhir nya. Ini karena setiap langkah pada penyelesaian Rush Hour memiliki biaya seragam yaitu satu. BFS dapat dianggap sebagai kasus khusus dari UCS, yaitu UCS pada kasus semua biaya adalah satu (atau *unweighted graph)*.

2.4 A* Search

A* (A Star) adalah algoritma *informed search* yang menggabungkan kekuatan pendekatan Uniform Cost Search (UCS) dan Greedy Best First Search (GBFS). A* mempertimbangkan baik biaya sejauh ini dari start node (g(n)) maupun estimasi biaya menuju goal (h(n)), sehingga mampu menyeimbangkan antara eksplorasi dan eksploitasi.

Fungsi evaluasi dari A* adalah sebagai berikut:

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

Langkah-langkah algoritma A*:

Inisialisasi:

- Sebuah priority *queue* disiapkan untuk menyimpan state, diurutkan berdasarkan nilai total fungsi evaluasi.
- Algoritma mengambil *state* dengan nilai f(n) terkecil dari queue.
- Jika state tersebut merupakan goal (primary piece mencapai pintu keluar), maka pencarian dihentikan dan jalur solusi dikembalikan.
- Semua gerakan valid dari state sekarang dihitung. Untuk setiap gerakan, dibuat state baru dengan g(n) dihitung dari depth (jumlah langkah dari root) dan h(n) dihitung dari heuristic.estimate(board).
- Jika representasi state belum pernah dikunjungi (visited), maka state ditambahkan ke queue. Representasi state didasarkan pada konfigurasi papan.
- Proses terus berjalan sampai queue kosong atau solusi ditemukan.

A* menjamin solusi optimal, asalkan heuristic yang digunakan bersifat admissible. Pada penyelesaian Rush Hour, A* **lebih efisien** daripada UCS karena jika heuristic yang digunakan baik, ini karena sifat *informed*-nya membuat A* cenderung mengunjungi lebih sedikit simpul. Selain itu, A* lebih stabil dan handal daripada GBFS karena tidak hanya mengejar estimasi tetapi juga memperhatikan jalur yang telah ditempuh.

Heuristik dikatakan *admissible* jika nilainya tidak pernah melebihi biaya sebenarnya dari simpul saat ini ke simpul tujuan. Heuristik *manhattan distance* yang digunakan akan mengukur jarak dari *primary piece* ke jalan keluar tanpa mempertimbangkan penghalang. Heuristik ini **tergolong *admissible*** karena hasil pengukuran jarak tidak akan lebih besar dari jarak sebenarnya.

Heuristik *blocking cars*, dimana nilai heuristik dihitung dari jumlah *piece* yang menghalangi jalur keluar, juga tergolong admissible karena hanya menghitung jumlah penghalang, bukan biaya pergeserannya, sehingga dipastikan tidak akan melebihi nilai sebenarnya. Namun, untuk heuristik kombinasi, karena ada pembobotan, nilai heuristik yang didapat bisa melebihi nilai sebenarnya pada kasus tertentu, sehingga membuatnya tidak admissible

2.5 IDDFS

Iterative Deepening Depth First Search (atau *Iterative Deepening Search*) adalah algoritma pencarian yang menggunakan kelebihan dari *Depth First Search* (DFS) dan *Breadth First Search* (BFS). IDDFS melakukan pencarian DFS dengan kedalaman terbatas yang meningkat secara iteratif, sehingga meskipun menggunakan pendekatan DFS, algoritma ini dapat menjamin solusi optimal untuk *unweighted graph* (layaknya BFS).

Fungsi evaluasi dari IDDFS adalah sebagai berikut:

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

Langkah-langkah algoritma IDDFS:

- Mulai dari kedalaman *limit* = 0, dan terus menaikkan limit secara iteratif jika solusi belum ditemukan.
- Untuk setiap *limit*, lakukan eksplorasi kemungkinan menggunakan DFS dengan batas kedalaman tersebut. Jika dalam batas kedalaman ditemukan solusi, pencarian dihentikan. Jika tidak, lanjut ke iterasi berikutnya dengan limit + 1.
- Dalam setiap iterasi DFS, semua kemungkinan gerakan kendaraan dieksplorasi secara rekursif hingga kedalaman limit. Setiap konfigurasi papan (*state*) disimpan dalam visited set untuk mencegah siklus dan eksplorasi ulang.
- Berhenti ketika solusi ditemukan atau tidak ada lagi state yang bisa dieksplorasi pada kedalaman tertentu.

2.6 Hill Climbing Search

Hill Climbing Search adalah algoritma pencarian lokal dan bersifat *greedy*, yang mencoba mencari solusi dengan cara bergerak ke state tetangga yang memiliki nilai heuristik terbaik. Algoritma ini tidak dapat menjamin penemuan solusi (tidak *complete*), baik optimal maupun nonoptimal, karena bisa terjebak pada lokal optimum.

Fungsi evaluasi dari *Hill Climb Search* adalah sebagai berikut:

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

Langkah-langkah algoritma Hill Climb Search:

- Mulai dengan konfigurasi papan awal dan hitung nilai heuristiknya.
- Tandai konfigurasi papan saat ini agar tidak dikunjungi kembali dalam upaya pendakian yang sedang berlangsung.
- Hasilkan semua kemungkinan konfigurasi papan berikutnya (tetangga) yang dapat dicapai dari konfigurasi saat ini melalui satu gerakan mobil.
- Jika tidak ada konfigurasi tetangga yang dapat dihasilkan, hentikan pencarian.
- Jika tetangga dapat dihasilkan, **acak urutan dari daftar konfigurasi tetangga tersebut**. Pengacakan dilakukan agar mengurangi kemungkinan terjebak. Kemudian evaluasi setiap konfigurasi tetangga yang belum pernah dikunjungi dalam pendakian ini.
- Evaluasi setiap konfigurasi tetangga yang belum pernah dikunjungi dengan menghitung nilai estimasi kedekatan ke solusi untuk konfigurasi tetangga yang sedang dievaluasi.
- Setelah evaluasi, jika ada kandidat tetangga terbaik, perbarui konfigurasi papan saat ini menjadi kandidat terbaik tersebut, dan perbarui nilai heuristik saat ini
- Jika tidak ada kandidat tetangga terbaik, tetapi ada kandidat untuk gerakan menyamping (nilai heuristik sama), perbarui konfigurasi papan saat ini menjadi kandidat gerakan menyamping tersebut dan tingkatkan catatan jumlah gerakan menyamping berturut-turut.
- Jika tidak ada kandidat tetangga terbaik maupun kandidat gerakan menyamping yang valid untuk dipilih, hentikan pencarian karena terjebak (solusi tidak ditemukan).
- Catat bahwa satu langkah pencarian telah dilakukan dan ulangi proses dari pengecekan solusi dengan konfigurasi papan saat ini yang baru..

III. Implementasi

3.1 Struktur Program

Berikut struktur program yang diimplementasikan :

```
Tucil3_13523035_13523117/
.
├── bin
├── doc
│   └── LaporanTucil3_035_117.pdf
├── LICENSE
├── makefile
├── pom.xml
├── README.md
├── src
│   ├── algorithm
│   │   ├── heuristic
│   │   │   ├── BlockingCarsHeuristic.java
│   │   │   ├── CombinedHeuristic.java
│   │   │   ├── Heuristic.java
│   │   │   └── ManhattanToExitHeuristic.java
│   │   └── pathfinding
│   │       ├── AStar.java
│   │       ├── GreedyBestFirst.java
│   │       ├── HillClimbing.java
│   │       ├── IDDFS.java
│   │       ├── Pathfinder.java
│   │       └── UCS.java
│   ├── Main.java
│   ├── model
│   │   ├── Board.java
│   │   ├── Piece.java
│   │   ├── Position.java
│   │   └── State.java
│   ├── ui
│   │   ├── GUIHelper.java
│   │   └── GUI.java
│   └── utils
│       ├── InputParser.java
│       ├── Direction.java
│       └── OutputWriter.java
└── test
```

3.2 Source Code Program

3.2.1 Package src (root)

Main.java

```
public class Main {
    public static void main(String[] args) {
        GUI.main(args);
    }
};
```

3.2.2 Package model

Board.java

```
public class Board {
    public int rows, cols;
    public Map<Character, Piece> pieces;
    public char[][] grid;
    public Position exit;

    public Board(int rows, int cols, char[][] grid, Map<Character, Piece> pieces, Position exit) {
        this.rows = rows;
        this.cols = cols;
        this.grid = grid;
        this.pieces = pieces;
        Position adjustedExit = new Position(exit.row, exit.col);
        if (adjustedExit.row == 0) {adjustedExit.row = 1;}
        else if (adjustedExit.row == rows - 1) {adjustedExit.row = rows - 2;}
        if (adjustedExit.col == 0) {adjustedExit.col = 1;}
        else if (adjustedExit.col == cols - 1) {adjustedExit.col = cols - 2;}

        this.exit = adjustedExit;
    }

    public Board copy() {
        Map<Character, Piece> copiedPieces = new HashMap<>();
        for (Map.Entry<Character, Piece> entry : pieces.entrySet()) {
            copiedPieces.put(entry.getKey(), entry.getValue().copy());
        }

        char[][] copiedGrid = new char[rows][cols];
        for (int i = 0; i < rows; i++) {
            copiedGrid[i] = Arrays.copyOf(grid[i], cols);
        }

        return new Board(rows, cols, copiedGrid, copiedPieces, exit.copy());
    }

   public boolean isSolved() {
        Piece primary = pieces.get('P');
        if (primary.isHorizontal) {
            if (primary.start.row != this.exit.row) {return false;}
```

```
            for (int i = 0; i < primary.length; i++) {
                if (primary.start.col + i == this.exit.col) return true;
            }
        } else {
            if (primary.start.col != this.exit.col) {return false;}

            for (int i = 0; i < primary.length; i++) {
                if (primary.start.row + i == this.exit.row) return true;
            }
        }
        return false;
    }

    public Set<Position> getOccupiedPositions(Piece piece) {
        Set<Position> result = new HashSet<>();
        int row = piece.start.row;
        int col = piece.start.col;
        for (int i = 0; i < piece.length; i++) {
            result.add(new Position(row, col));
            if (piece.isHorizontal) col++;
            else row++;
        }
        return result;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Board)) return false;
        Board b = (Board) o;
        return Arrays.deepEquals(this.grid, b.grid);
    }

    @Override
    public int hashCode() {
        return Arrays.deepHashCode(this.grid);
    }

    public boolean isInEdge(Position pos){
        if (pos.col == 0 || pos.row == 0) return true;
        if (pos.col == this.cols-1 || pos.row == this.rows-1) return true;
        return false;
    }

    public void print() {
        for (int i = 0; i < this.rows; i++) {
            for (int j = 0; j < this.cols; j++) {
                char c = this.grid[i][j];
                if (c == 'P') {
                    System.out.print("\u001B[31m" + c + "\u001B[0m"); // Merah
                } else if (c == 'K') {
                    System.out.print("\u001B[34m" + c + "\u001B[0m"); // Biru
                } else {
                    System.out.print(c);
```

```
                }
            }
            System.out.println();
        }
        System.out.println();
    }

    // Used for GUI animation
    public static Board createIntermediateBoard(Board startBoard, Board endBoard, char pieceId, String direction, int
currentStep, int totalSteps) {
        Board intermediateBoard = startBoard.copy();
        Piece piece = intermediateBoard.pieces.get(pieceId);
        if (piece == null) {
            return intermediateBoard;
        }

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

        int deltaRow = 0;
        int deltaCol = 0;

        switch (direction.toLowerCase()) {
            case "up":
                deltaRow = -currentStep;
                break;
            case "down":
                deltaRow = currentStep;
                break;
            case "left":
                deltaCol = -currentStep;
                break;
            case "right":
                deltaCol = currentStep;
                break;
        }

        piece.start = new Position(startBoard.pieces.get(pieceId).start.row + deltaRow,
startBoard.pieces.get(pieceId).start.col + deltaCol);

        int r = piece.start.row;
        int c = piece.start.col;
        for (int i = 0; i < piece.length; i++) {
            intermediateBoard.grid[r][c] = pieceId;
            if (piece.isHorizontal) c++;
            else r++;
        }
```

```
        return intermediateBoard;
    }
}
```

Piece.java

```
public class Piece {
    public char id;
    public Position start;
    public int length;
    public boolean isHorizontal;

    public Piece(char id, Position start, int length, boolean isHorizontal) {
        this.id = id;
        this.start = start;
        this.length = length;
        this.isHorizontal = isHorizontal;
    }

    public Piece copy() {
        return new Piece(id, start.copy(), length, isHorizontal);
    }
}
```

Position.java

```
public class Position {
    public int row;
    public int col;

    public Position(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public Position copy() {
        return new Position(row, col);
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Position)) return false;
        Position p = (Position) o;
        return this.row == p.row && this.col == p.col;
    }

    @Override
    public int hashCode() {
        return Objects.hash(row, col);
    }
}
```

State.java

```
public class State {
    public Board board;
    public String move;  // contoh: "A-kanan"
    public State prev;

    public State(Board board, String move, State prev) {
        this.board = board;
        this.move = move;
        this.prev = prev;
    }

    public int getDepth() {
        int count = 0;
        State curr = this;
        while (curr.prev != null) {
            count++;
            curr = curr.prev;
        }
        return count;
    }

    public void printPath() {
        if (prev != null) {
            prev.printPath();
        }
        if (!move.equals("Start")) {
            System.out.println("Gerakan: " + move);
        }
    }
}
```

3.2.3 Package algorithm.heuristic

Heuristic.java

```
public interface Heuristic {
    int estimate(Board board);
}
```

BlockingCarsHeuristic.java

```
public class BlockingCarsHeuristic implements Heuristic {
    @Override
    public int estimate(Board board) {
        Piece primary = board.pieces.get('P');
        if (primary == null || !primary.isHorizontal) return Integer.MAX_VALUE;

        int row = primary.start.row;
        int col = primary.start.col + primary.length;
```

```
        int count = 0;

        while (col < board.cols) {
            char cell = board.grid[row][col];
            if (cell != '.' && cell != 'K') {
                count += (board.cols - col);
            }
            col++;
        }

        return count;
    }

    @Override
    public String toString() {
        return "BlockingCarsHeuristic";
    }
}
```

ManhattanToExitHeuristic.java

```
public class ManhattanToExitHeuristic implements Heuristic {
    @Override
    public int estimate(Board board) {
        Piece primary = board.pieces.get('P');
        if (primary == null || !primary.isHorizontal) return Integer.MAX_VALUE;

        int row = primary.start.row;
        int endCol = primary.start.col + primary.length - 1;

        int distanceToExit = Math.abs(endCol - (board.cols - 1));

        int blocking = 0;
        for (int col = endCol + 1; col < board.cols; col++) {
            if (board.grid[row][col] != '.' && board.grid[row][col] != 'K') {
                blocking++;
            }
        }

        return distanceToExit + (blocking * 2);
    }

    @Override
    public String toString() {
        return "ManhattanToExitHeuristic";
    }
}
```

CombinedHeuristic.java

```
public class CombinedHeuristic implements Heuristic {
    @Override
    public int estimate(Board board) {
```

```
        Piece primary = board.pieces.get('P');
        if (primary == null || !primary.isHorizontal) return Integer.MAX_VALUE;

        int row = primary.start.row;
        int endCol = primary.start.col + primary.length - 1;

        // Manhattan Distance
        int distanceToExit = Math.abs(endCol - (board.cols - 1));

        // Blocking Cars
        int blockingPenalty = 0;
        int weightedBlocking = 0;
        for (int col = endCol + 1; col < board.cols; col++) {
            char cell = board.grid[row][col];
            if (cell != '.' && cell != 'K') {
                blockingPenalty++;
                weightedBlocking += (board.cols - col);
            }
        }

        int result = distanceToExit + (2 * blockingPenalty) + weightedBlocking;

        return result;
    }

    @Override
    public String toString() {
        return "CombinedHeuristic";
    }
}
```

3.2.4 Package algorithm.pathfinding

Pathfinding.java

```
public class Pathfinder {
    protected long runtimeNano = -1;
    protected int nodes;
    Protected Heuristic heuristic;
    public List<State> solve(Board startBoard) {
        throw new UnsupportedOperationException("Method not implemented");
    }

    public String getName() {
        return "Abstract Pathfinder";
    }

    public long getRuntimeNano() {
        return this.runtimeNano;
    }
```

```java
public int getNodes(){
    return this.nodes;
}

protected List<State> generateNeighbors(State current) {
    List<State> result = new ArrayList<>();
    Board board = current.board;

    for (Piece piece : board.pieces.values()) {
        if (piece.isHorizontal) {
            // Gerak ke kiri
            for (int step = 1; step <= board.cols; step++) {
                if (canMovePieceHorizontal(board, piece, -step)) {
                    Board newBoard = movePieceAndCreateNewBoard(board, piece, 0, -step);
                    String move = piece.id + "-left" + (step > 1 ? "-" + step : "");
                    result.add(new State(newBoard, move, current));
                } else {
                    break;
                }
            }

            // Gerak ke kanan
            for (int step = 1; step <= board.cols; step++) {
                if (canMovePieceHorizontal(board, piece, step)) {
                    Board newBoard = movePieceAndCreateNewBoard(board, piece, 0, step);
                    String move = piece.id + "-right" + (step > 1 ? "-" + step : "");
                    result.add(new State(newBoard, move, current));
                } else {
                    break;
                }
            }
        } else {
            // Gerak ke atas
            for (int step = 1; step <= board.rows; step++) {
                if (canMovePieceVertical(board, piece, -step)) {
                    Board newBoard = movePieceAndCreateNewBoard(board, piece, -step, 0);
                    String move = piece.id + "-up" + (step > 1 ? "-" + step : "");
                    result.add(new State(newBoard, move, current));
                } else {
                    break;
                }
            }

            // Gerak ke bawah
            for (int step = 1; step <= board.rows; step++) {
                if (canMovePieceVertical(board, piece, step)) {
                    Board newBoard = movePieceAndCreateNewBoard(board, piece, step, 0);
                    String move = piece.id + "-down" + (step > 1 ? "-" + step : "");
                    result.add(new State(newBoard, move, current));
                } else {
                    break;
                }
            }
```

```
            }
        }
        return result;
    }

    private boolean canMovePieceHorizontal(Board board, Piece piece, int deltaCol) {
        if (!piece.isHorizontal) return false;

        int row = piece.start.row;
        int targetCol = (deltaCol > 0) ?
        piece.start.col + piece.length + deltaCol - 1 : piece.start.col + deltaCol;

        if (targetCol < 1 || targetCol >= board.cols-1) {
            return false;
        }

        if (deltaCol > 0) { // kanan
            for (int c = piece.start.col + piece.length; c <= piece.start.col + piece.length + deltaCol - 1; c++) {
                if (c >= 0 && c < board.cols) {
                    char cell = board.grid[row][c];
                    if (cell != '.' && cell != 'K') {
                        return false;
                    }
                } else {
                    return false;
                }
            }
        } else { // kiri
            for (int c = piece.start.col - 1; c >= piece.start.col + deltaCol; c--) {
                if (c >= 0 && c < board.cols) {
                    char cell = board.grid[row][c];
                    if (cell != '.' && cell != 'K') {
                        return false;
                    }
                } else {
                    return false;
                }
            }
        }

        return true;
    }

    private boolean canMovePieceVertical(Board board, Piece piece, int deltaRow) {
        if (piece.isHorizontal) return false;

        int col = piece.start.col;
        int targetRow = (deltaRow > 0) ?
                        piece.start.row + piece.length + deltaRow - 1 : // bawah
                        piece.start.row + deltaRow; // atas
        if (targetRow < 1 || targetRow >= board.rows-1) {
            return false;
        }
```

```
    if (deltaRow > 0) { // bawah
        for (int r = piece.start.row + piece.length; r <= piece.start.row + piece.length + deltaRow - 1; r++) {
            if (r >= 0 && r < board.rows) {
                char cell = board.grid[r][col];
                if (cell != '.' && cell != 'K') {
                    return false;
                }
            } else {
                return false;
            }
        }
    } else { // atas
        for (int r = piece.start.row - 1; r >= piece.start.row + deltaRow; r--) {
            if (r >= 0 && r < board.rows) {
                char cell = board.grid[r][col];
                if (cell != '.' && cell != 'K') {
                    return false;
                }
            } else {
                return false;
            }
        }
    }

    return true;
}

private Board movePieceAndCreateNewBoard(Board board, Piece piece, int deltaRow, int deltaCol) {
    Board newBoard = board.copy();
    Piece newPiece = newBoard.pieces.get(piece.id);

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

    newPiece.start = new Position(piece.start.row + deltaRow, piece.start.col + deltaCol);

    int r = newPiece.start.row;
    int c = newPiece.start.col;
    for (int i = 0; i < newPiece.length; i++) {
        newBoard.grid[r][c] = newPiece.id;
        if (newPiece.isHorizontal) c++;
        else r++;
    }

    return newBoard;
}
```

```
    protected List<State> reconstructStatePath(State goal) {
        List<State> path = new ArrayList<>();
        State current = goal;
        while (current != null) {
            path.add(current);
            current = current.prev;
        }
        Collections.reverse(path);
        return path;
    }

    protected static String boardToString(Board board) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < board.rows; i++) {
            for (int j = 0; j < board.cols; j++) {
                sb.append(board.grid[i][j]);
            }
        }
        return sb.toString();
    }
}
```

AStar.java

```
public class AStar extends Pathfinder {
    public AStar(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public String getName() {
        return "A* Search with " + heuristic.toString();}
    @Override
    public List<State> solve(Board startBoard) {
        PriorityQueue<Node> openSet = new PriorityQueue<>();
        Map<String, Integer> gScore = new HashMap<>();
        Set<String> visited = new HashSet<>();

        State start = new State(startBoard, "Start", null);
        long startTime = System.nanoTime();
        int startH = heuristic.estimate(start.board);
        openSet.add(new Node(start, startH, 0));
        String startBoardKey = Pathfinder.boardToString(startBoard);
        gScore.put(startBoardKey, 0);

        System.out.println("Starting A* search with " + heuristic.toString() + "...");
        System.out.println("Initial heuristic value: " + startH);
        int expandedNodes = 0;

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();
            State currState = current.state;
            Board board = currState.board;
```

```
            String boardKey = Pathfinder.boardToString(board);

            expandedNodes++;
            if (expandedNodes % 1000 == 0) {
                System.out.println("Expanded " + expandedNodes + " nodes, current queue size: " + openSet.size());
            }

            if (board.isSolved()) {
                this.runtimeNano = System.nanoTime() - startTime;
                this.nodes = expandedNodes;
                System.out.println("Solution found after expanding " + expandedNodes + " nodes!");
                return reconstructStatePath(currState);
            }

            if (visited.contains(boardKey)) continue;
            visited.add(boardKey);

            for (State neighbor : generateNeighbors(currState)) {
                String neighborKey = Pathfinder.boardToString(neighbor.board);
                if (visited.contains(neighborKey)) continue;

                int tentativeG = current.gScore + 1;
                if (!gScore.containsKey(neighborKey) || tentativeG < gScore.get(neighborKey)) {
                    gScore.put(neighborKey, tentativeG);
                    int h = heuristic.estimate(neighbor.board);
                    int f = tentativeG + h;
                    openSet.add(new Node(neighbor, f, tentativeG));
                }
            }
        }
        this.runtimeNano = System.nanoTime() - startTime;
        this.nodes = expandedNodes;
        System.out.println("No solution found after expanding " + expandedNodes + " nodes.");
        return new ArrayList<>();
    }

    private static class Node implements Comparable<Node> {
        State state;
        int fScore;
        int gScore;

        public Node(State state, int fScore, int gScore) {
            this.state = state;
            this.fScore = fScore;
            this.gScore = gScore;
        }

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.fScore, other.fScore);
        }
    }
}
```

GreedyBestFirst.java

```java
public class GreedyBestFirst extends Pathfinder {
    public GreedyBestFirst(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public String getName() {
        return "Greedy Best First Search with " + heuristic.toString();
    }

    @Override
    public List<State> solve(Board startBoard) {
        PriorityQueue<Node> openSet = new PriorityQueue<>();
        Set<String> visited = new HashSet<>();

        State start = new State(startBoard, "Start", null);
        long startTime = System.nanoTime();
        int startH = heuristic.estimate(startBoard);
        openSet.add(new Node(start, startH));

        System.out.println("Starting Greedy Best First search with " + heuristic.toString() + "...");
        System.out.println("Initial heuristic value: " + startH);
        int expandedNodes = 0;

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();
            State currState = current.state;
            Board board = currState.board;
            String boardKey = Pathfinder.boardToString(board);

            expandedNodes++;
            if (expandedNodes % 1000 == 0) {
                System.out.println("Expanded " + expandedNodes + " nodes, current queue size: " + openSet.size());
            }

            if (board.isSolved()) {
                this.runtimeNano = System.nanoTime() - startTime;
                this.nodes = expandedNodes;
                System.out.println("Solution found after expanding " + expandedNodes + " nodes!");
                return reconstructStatePath(currState);
            }

            if (visited.contains(boardKey)) continue;
            visited.add(boardKey);

            for (State neighbor : generateNeighbors(currState)) {
                String neighborKey = Pathfinder.boardToString(neighbor.board);
                if (!visited.contains(neighborKey)) {
                    int h = heuristic.estimate(neighbor.board);
                    openSet.add(new Node(neighbor, h));
                }
```

```
            }
        }
        this.runtimeNano = System.nanoTime() - startTime;
        this.nodes = expandedNodes;
        System.out.println("No solution found after expanding " + expandedNodes + " nodes.");
        return new ArrayList<>();
    }

    private static class Node implements Comparable<Node> {
        State state;
        int priority;

        public Node(State state, int priority) {
            this.state = state;
            this.priority = priority;
        }

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.priority, other.priority);
        }
    }
}
```

UCS.java

```
public class UCS extends Pathfinder {
    @Override
    public String getName() {
        return "Uniform Cost Search";
    }

    @Override
    public List<State> solve(Board startBoard) {
        PriorityQueue<Node> openSet = new PriorityQueue<>();
        Set<String> visited = new HashSet<>();
        Map<String, Integer> gScore = new HashMap<>();

        State start = new State(startBoard, "Start", null);
        long startTime = System.nanoTime();
        openSet.add(new Node(start, 0));
        String startBoardKey = Pathfinder.boardToString(startBoard);
        gScore.put(startBoardKey, 0);

        System.out.println("Starting UCS search...");
        int expandedNodes = 0;

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();
            State currState = current.state;
            Board board = currState.board;
            String boardKey = Pathfinder.boardToString(board);
```

```
            expandedNodes++;
            if (expandedNodes % 1000 == 0) {
                System.out.println("Expanded " + expandedNodes + " nodes, current queue size: " + openSet.size());
            }

            if (board.isSolved()) {
                this.runtimeNano = System.nanoTime() - startTime;
                this.nodes = expandedNodes;
                System.out.println("Solution found after expanding " + expandedNodes + " nodes!");
                return reconstructStatePath(currState);
            }

            if (visited.contains(boardKey)) continue;
            visited.add(boardKey);

            for (State neighbor : generateNeighbors(currState)) {
                String neighborKey = Pathfinder.boardToString(neighbor.board);
                if (visited.contains(neighborKey)) continue;

                int tentativeG = current.cost + 1;
                if (!gScore.containsKey(neighborKey) || tentativeG < gScore.get(neighborKey)) {
                    gScore.put(neighborKey, tentativeG);
                    openSet.add(new Node(neighbor, tentativeG));
                }
            }
        }
        this.runtimeNano = System.nanoTime() - startTime;
        this.nodes = expandedNodes;
        System.out.println("No solution found after expanding " + expandedNodes + " nodes.");
        return new ArrayList<>();
    }

    private static class Node implements Comparable<Node> {
        State state;
        int cost;

        public Node(State state, int cost) {
            this.state = state;
            this.cost = cost;
        }

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.cost, other.cost);
        }
    }
}
```

IDDFS.java

```
public class IDDFS extends Pathfinder {
    @Override
```

```java
    public String getName() {
        return "Iterative Deepening Search (IDS)";
    }

    @Override
    public List<State> solve(Board startBoard) {
        System.out.println("Starting Iterative Deepening Search...");
        long startTime = System.nanoTime();
        int totalNodesExpandedEstimate = 0;

        for (int depthLimit = 0; ; depthLimit++) {
            State startState = new State(startBoard, "Start", null);
            Set<String> visitedInCurrentPath = new HashSet<>();

            int[] nodesExpandedThisDLS = {0};
            List<State> resultPath = depthLimitedSearch(startState, 0, depthLimit, visitedInCurrentPath,
nodesExpandedThisDLS);
            totalNodesExpandedEstimate += nodesExpandedThisDLS[0];

            if (resultPath != null) {
                this.runtimeNano = System.nanoTime() - startTime;
                this.nodes = totalNodesExpandedEstimate;
                System.out.println("Solution found at depth " + (resultPath.size() - 1) + " after exploring approximately "
+ totalNodesExpandedEstimate + " states.");
                return resultPath;
            }

            if (depthLimit > 30 && totalNodesExpandedEstimate > 1000000) {
                 System.out.println("Search stopped: Exceeded reasonable depth or node expansion limit. Total states
explored: " + totalNodesExpandedEstimate);
                 break;
            }
            if (depthLimit > 50) {
                System.out.println("Search stopped: Exceeded absolute depth limit of 50. Total states explored: " +
totalNodesExpandedEstimate);
                break;
            }
        }
        this.runtimeNano = System.nanoTime() - startTime;
        this.nodes = totalNodesExpandedEstimate;
        System.out.println("No solution found. Total states explored: " + totalNodesExpandedEstimate);
        return new ArrayList<>();
    }

    private List<State> depthLimitedSearch(State currentState, int currentDepth, int maxDepth, Set<String>
visitedInCurrentPath, int[] nodesExpandedCounter) {
        nodesExpandedCounter[0]++;
        String currentBoardKey = Pathfinder.boardToString(currentState.board);

        if (currentState.board.isSolved()) {
            return reconstructStatePath(currentState);
        }
```

```
        if (currentDepth >= maxDepth) {
            return null;
        }

        visitedInCurrentPath.add(currentBoardKey);

        for (State neighbor : generateNeighbors(currentState)) {
            String neighborBoardKey = Pathfinder.boardToString(neighbor.board);
            if (!visitedInCurrentPath.contains(neighborBoardKey)) {
                List<State> path = depthLimitedSearch(neighbor, currentDepth + 1, maxDepth, visitedInCurrentPath,
nodesExpandedCounter);
                if (path != null) {
                    visitedInCurrentPath.remove(currentBoardKey);
                    return path;
                }
            }
        }
        visitedInCurrentPath.remove(currentBoardKey);
        return null;
    }
}
```

HillClimbing.java

```
public class HillClimbing extends Pathfinder {
    public HillClimbing(Heuristic heuristic) {
        this.heuristic = heuristic;
    }

    @Override
    public String getName() {
        return "Hill Climbing Search using " + heuristic.toString();
    }

    @Override
    public List<State> solve(Board startBoard) {
        System.out.println("Starting Hill Climbing search using " + heuristic.toString() + "...");

        this.nodes = 0;
        long startTime = System.nanoTime();

        State currentState = new State(startBoard, "Start", null);
        int currentHeuristicValue = heuristic.estimate(currentState.board);
        int stepsTaken = 0;
        int consecutiveSidewaysMoves = 0;
        this.nodes++;

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

        while (true) {
            if (currentState.board.isSolved()) {
                this.runtimeNano = System.nanoTime() - startTime;
                System.out.println("Solution found after " + stepsTaken + " steps (" + consecutiveSidewaysMoves + " final
```

```
sideways).");
                return reconstructStatePath(currentState);
            }

            String currentBoardKey = boardToString(currentState.board);
            visitedStatesThisAttempt.add(currentBoardKey);

            List<State> neighbors = generateNeighbors(currentState);
            Collections.shuffle(neighbors);

            State bestNeighbourValue = null;
            int bestHeuristicValue = currentHeuristicValue;

            State firstSidewaysNeighbor = null;

            if (neighbors.isEmpty()) {
                break;
            }

            for (State neighbor : neighbors) {
                this.nodes++;
                String neighborBoardKey = boardToString(neighbor.board);
                if (visitedStatesThisAttempt.contains(neighborBoardKey)) {continue;}

                int neighborHeuristic = heuristic.estimate(neighbor.board);

                if (neighborHeuristic < bestHeuristicValue) {
                    bestHeuristicValue = neighborHeuristic;
                    bestNeighbourValue = neighbor;
                } else if (neighborHeuristic == currentHeuristicValue && firstSidewaysNeighbor == null) {
                    firstSidewaysNeighbor = neighbor;
                }
            }

            if (bestNeighbourValue != null) {
                currentState = bestNeighbourValue;
                currentHeuristicValue = bestHeuristicValue;
                consecutiveSidewaysMoves = 0;
            } else if (firstSidewaysNeighbor != null) {
                currentState = firstSidewaysNeighbor;
                consecutiveSidewaysMoves++;
            } else {
                break;
            }
            stepsTaken++;
        }

        this.runtimeNano = System.nanoTime() - startTime;

        if (currentState.board.isSolved()) {
             System.out.println("Solution found at MAX_STEPS (" + stepsTaken + ").");
             return reconstructStatePath(currentState);
        }
```

```
        System.out.println("No solution found or stuck at local optimum/plateau after " + stepsTaken + " steps.");
        return new ArrayList<>();
    }
}
```

3.2.5 Package ui

GUI.java

```
public class GUI extends Application {
    public String algorithm;
    public String heuristic;
    public String filePath;
    public Board board;
    public Pathfinder solver;
    public List<State> solution;
    public double animationDelay;

    public Stage primaryStage;
    public StackPane boardDisplayArea;

    static int FILE_PATH_FIELD_WIDTH = 300;


    @Override
    public void start(Stage primaryStage) {
        this.primaryStage = primaryStage;
        this.primaryStage.setTitle("Rush Hour Solver");
        initGui();
        this.primaryStage.show();
        this.primaryStage.centerOnScreen();
    }


    public void initGui(){
        Timeline timeline = new Timeline(new KeyFrame(
            Duration.millis(70),
            ae -> titleScreen()));
        timeline.play();titleScreen();
    }


    public void titleScreen(){
        Label titleLabel = new Label("Rush Hour Solver");
        titleLabel.setFont(Font.font("Arial", FontWeight.BOLD, 30));

        Label authorLabel = new Label("By: 13523035, 13523117");
        authorLabel.setFont(Font.font("Arial", 18));

        VBox titleBox = new VBox(10);
```

```
        titleBox.getChildren().addAll(titleLabel, authorLabel);
        titleBox.setAlignment(Pos.CENTER);

        Button startButton = GUIHelper.createButton("Start", e -> {userInputs();});
        startButton.setPrefWidth(170);
        startButton.setPrefHeight(40);

        Button exitButton = GUIHelper.createButton("Exit", e -> {this.primaryStage.close();});
        exitButton.setPrefWidth(170);
        exitButton.setPrefHeight(40);

        VBox buttonBox = new VBox(30);
        buttonBox.getChildren().addAll(startButton, exitButton);
        buttonBox.setAlignment(Pos.CENTER);

        BorderPane mainLayout = new BorderPane();
        mainLayout.setPadding(new Insets(150, 20, 20, 20));

        mainLayout.setTop(titleBox);
        BorderPane.setAlignment(titleBox, Pos.CENTER);

        mainLayout.setCenter(buttonBox);
        BorderPane.setAlignment(buttonBox, Pos.CENTER);

        Scene scene = new Scene(mainLayout, 800, 600);
        this.primaryStage.setScene(scene);
        this.primaryStage.centerOnScreen();
    }


    public void userInputs(){
        // Algorithms Elements
        Label algoLabel = new Label("Choose your algorithm:");
        algoLabel.setFont(Font.font("Arial", FontWeight.BOLD, 20));

        ToggleGroup algoToggleGroup = new ToggleGroup();

        RadioButton aStarRadio = new RadioButton("A* Search");
        aStarRadio.setToggleGroup(algoToggleGroup);
        aStarRadio.setUserData("A* Search");
        aStarRadio.setFont(Font.font("Arial", 16));

        RadioButton greedyRadio = new RadioButton("Greedy Best First Search");
        greedyRadio.setToggleGroup(algoToggleGroup);
        greedyRadio.setUserData("Greedy Best First Search");
        greedyRadio.setFont(Font.font("Arial", 16));

        RadioButton ucsRadio = new RadioButton("Uniform Cost Search");
        ucsRadio.setToggleGroup(algoToggleGroup);
        ucsRadio.setUserData("Uniform Cost Search");
        ucsRadio.setFont(Font.font("Arial", 16));

        RadioButton iddfsRadio = new RadioButton("Iterative Deepening Search");
```

```
        iddfsRadio.setToggleGroup(algoToggleGroup);
        iddfsRadio.setUserData("Iterative Deepening Search");
        iddfsRadio.setFont(Font.font("Arial", 16));

        RadioButton hillClimbRadio = new RadioButton("Hill Climbing Search");
        hillClimbRadio.setToggleGroup(algoToggleGroup);
        hillClimbRadio.setUserData("Hill Climbing Search");
        hillClimbRadio.setFont(Font.font("Arial", 16));

        aStarRadio.setSelected(true);
        this.algorithm = "A* Search";

        algoToggleGroup.selectedToggleProperty().addListener((observable, oldVal, newVal) -> {
            if (newVal != null) {
                this.algorithm = (String) newVal.getUserData();
                System.out.println("Algorithm selected: " + this.algorithm);
            }
        });

        VBox radioButtonBox = new VBox(10, aStarRadio, greedyRadio, ucsRadio, iddfsRadio,hillClimbRadio);
        radioButtonBox.setAlignment(Pos.CENTER_LEFT);

        HBox radioContainer = new HBox(radioButtonBox);
        radioContainer.setAlignment(Pos.CENTER);
        radioContainer.setMinWidth(300);

        HBox labelContainer = new HBox(algoLabel);
        labelContainer.setAlignment(Pos.CENTER);

        VBox algoBox = new VBox(10, labelContainer, radioContainer);
        algoBox.setAlignment(Pos.CENTER);
        // End of Algorithm Elements

        // File Input Elements
        Label fileLabel = new Label("Browse your board configuration file:");
        fileLabel.setFont(Font.font("Arial", FontWeight.BOLD, 20));
        TextField filePathField = new TextField();
        filePathField.setPromptText("e.g., /path/to/your/puzzle.txt");
        filePathField.setPrefWidth(FILE_PATH_FIELD_WIDTH);
        filePathField.setPrefHeight(GUIHelper.INPUT_BTN_HEIGHT);
        filePathField.setEditable(false);


        Button browseButton = GUIHelper.createButton("Browse...", e -> {
            FileChooser fileChooser = new FileChooser();
            fileChooser.setTitle("Open Configuration File");
            fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("Text Files", "*.txt"));
            File selectedFile = fileChooser.showOpenDialog(this.primaryStage);
            if (selectedFile != null) {
                filePathField.setText(selectedFile.getAbsolutePath());
                this.filePath = selectedFile.getAbsolutePath();
                System.out.println("File selected: " + this.filePath);
            }
```

```
        });

        HBox fileInputBox = new HBox(10, filePathField, browseButton);
        fileInputBox.setAlignment(Pos.CENTER);
        // End of File Input Elements


        Button nextButton = GUIHelper.createButton("Next", e -> {
             this.filePath = filePathField.getText();

            if (this.algorithm == null || this.algorithm.isEmpty()) {
                System.out.println("Please select an algorithm.");
                return;
            }
            if (this.filePath == null || this.filePath.isEmpty()) {
                System.out.println("Please enter or browse for a configuration file.");
                return;
            }
            if (this.algorithm.equals("A* Search") || this.algorithm.equals("Greedy Best First Search") ||
this.algorithm.equals("Hill Climbing Search")){
                heuristicInput();
            } else {
                processInput();
            }
        });


        Button backButton = GUIHelper.createButton("Back", e-> {titleScreen();});

        HBox navigationButtonBox = new HBox(20, backButton, nextButton);
        navigationButtonBox.setAlignment(Pos.CENTER);

        VBox userInputLayout = new VBox(25);
        userInputLayout.setPadding(new Insets(30));
        userInputLayout.setAlignment(Pos.CENTER);
        userInputLayout.getChildren().addAll(algoBox, fileLabel, fileInputBox, navigationButtonBox);

        Scene userInputScene = new Scene(userInputLayout, 800, 600);
        this.primaryStage.setScene(userInputScene);
        this.primaryStage.centerOnScreen();
    }


    public void heuristicInput(){
        Label heuristicLabel = new Label("Choose your Heuristic:");
        heuristicLabel.setFont(Font.font("Arial", FontWeight.BOLD, 20));

        ToggleGroup heuristicToggleGroup = new ToggleGroup();

        RadioButton blockingCarsRadio = new RadioButton("Blocking Cars");
        blockingCarsRadio.setToggleGroup(heuristicToggleGroup);
        blockingCarsRadio.setUserData("Blocking Cars");
        blockingCarsRadio.setFont(Font.font("Arial", 16));
```

```
RadioButton manhattanRadio = new RadioButton("Manhattan To Exit");
manhattanRadio.setToggleGroup(heuristicToggleGroup);
manhattanRadio.setUserData("Manhattan To Exit");
manhattanRadio.setFont(Font.font("Arial", 16));

RadioButton combinedRadio = new RadioButton("Combined Heuristic");
combinedRadio.setToggleGroup(heuristicToggleGroup);
combinedRadio.setUserData("Combined Heuristic");
combinedRadio.setFont(Font.font("Arial", 16));

blockingCarsRadio.setSelected(true);
this.heuristic = "Blocking Cars";

heuristicToggleGroup.selectedToggleProperty().addListener((observable, oldVal, newVal) -> {
    if (newVal != null) {
        this.heuristic = (String) newVal.getUserData();
        System.out.println("Heuristic Selected: " + this.heuristic);
    }
});

VBox radioButtonBox = new VBox(10, blockingCarsRadio, manhattanRadio, combinedRadio);
radioButtonBox.setAlignment(Pos.CENTER_LEFT);

HBox radioContainer = new HBox(radioButtonBox);
radioContainer.setAlignment(Pos.CENTER);
radioContainer.setMinWidth(300);

HBox labelContainer = new HBox(heuristicLabel);
labelContainer.setAlignment(Pos.CENTER);

VBox heuristicBox = new VBox(10, labelContainer, radioContainer);
heuristicBox.setAlignment(Pos.CENTER);
// End of Heuristic Elements

Button nextButton = GUIHelper.createButton("Next", e->{
    if (this.heuristic == null || this.heuristic.isEmpty()) {
        System.out.println("Please select a heuristic.");
        return;
    }
    processInput();
});

Button backButton = GUIHelper.createButton("Back", e->{userInputs();});

HBox navigationButtonBox = new HBox(20, backButton, nextButton);
navigationButtonBox.setAlignment(Pos.CENTER);

VBox userInputLayout = new VBox(25);
userInputLayout.setPadding(new Insets(30));
userInputLayout.setAlignment(Pos.CENTER);
userInputLayout.getChildren().addAll(heuristicBox, navigationButtonBox);
```

```
        Scene userInputScene = new Scene(userInputLayout, 800, 600);
        this.primaryStage.setScene(userInputScene);
        this.primaryStage.centerOnScreen();
    }


    public void processInput(){
        try {
            this.board = InputParser.parseFromFile(this.filePath);
            System.out.println("Berhasil membaca file.");
            this.board.print();

            initSearch();
        } catch (IOException e) {
            inputError(e.getMessage(), true);
        }
    }


    public void initSearch() {
        this.boardDisplayArea = new StackPane();

        VBox boardViewFromDrawBoard = GUIHelper.drawBoard(this.board);
        if (boardViewFromDrawBoard != null) {
            this.boardDisplayArea.getChildren().add(boardViewFromDrawBoard);
        }

        Label algoDisplayLabel = new Label("Algorithm: " + this.algorithm);
        algoDisplayLabel.setFont(Font.font("Arial", FontWeight.NORMAL, 18));

        VBox infoBox = new VBox(5);
        infoBox.setAlignment(Pos.CENTER);
        infoBox.getChildren().add(algoDisplayLabel);

        if (("A* Search".equals(this.algorithm) || "Greedy Best First Search".equals(this.algorithm) || "Hill Climbing
Search".equals(this.algorithm))
            && this.heuristic != null && !this.heuristic.isEmpty()) {
            Label heuristicDisplayLabel = new Label("Heuristic: " + this.heuristic);
            heuristicDisplayLabel.setFont(Font.font("Arial", FontWeight.NORMAL, 18));
            infoBox.getChildren().add(heuristicDisplayLabel);
        }

        Label initialBoardLabel = new Label("Initial Board");
        initialBoardLabel.setFont(Font.font("Arial", FontWeight.BOLD, 20));

        Label delayInputLabel = new Label("Animation Delay (ms):");
        delayInputLabel.setFont(Font.font("Arial", FontWeight.NORMAL, 16));
        this.animationDelay = 100;
        TextField delayTextField = new TextField();
        delayTextField.setPrefWidth(80);
        delayTextField.setPromptText("100");
        UnaryOperator<TextFormatter.Change> filter = change -> {
            String newText = change.getControlNewText();
```

```
            if (newText.matches("\\d*\\.?\\d*")) { // digits and only one decimal point
                return change;
            }
            return null;
        };
        TextFormatter<Double> delayFormatter = new TextFormatter<>(new StringConverter<Double>() {
                @Override
                public String toString(Double object) {
                    return object == null ? "" : object.toString();
                }
                @Override
                public Double fromString(String string) {
                    try {
                        return Double.parseDouble(string);
                    } catch (NumberFormatException e) {
                        return null;
                    }
                }
            }, this.animationDelay, filter);
        delayTextField.setTextFormatter(delayFormatter);
        delayTextField.textProperty().addListener((obs, oldVal, newVal) -> {
            try {
                double delay = Double.parseDouble(newVal);
                if (delay > 0) {this.animationDelay = delay;}
            } catch (NumberFormatException e) {
                System.out.println("Invalid delay input: " + newVal);
            }
        });

        HBox delayInputBox = new HBox(10, delayInputLabel, delayTextField);
        delayInputBox.setAlignment(Pos.CENTER);


        Button startSearchButton = GUIHelper.createButton("Start", e -> {});
        Button backButton = GUIHelper.createButton("Back", e -> {userInputs();});
        startSearchButton.setOnAction(e -> {
            startSearchButton.setDisable(true);
            backButton.setDisable(true);
            startSearch();
        });


        HBox buttonControlBox = new HBox(20, backButton, startSearchButton);
        buttonControlBox.setPadding(new Insets(0,0,40,0));
        buttonControlBox.setAlignment(Pos.CENTER);

        VBox searchScreenLayout = new VBox(20);
        searchScreenLayout.setAlignment(Pos.CENTER);

        searchScreenLayout.getChildren().add(initialBoardLabel);
        searchScreenLayout.getChildren().add(delayInputBox);
        if (this.boardDisplayArea != null) {
            searchScreenLayout.getChildren().add(this.boardDisplayArea);
```

```
        }
        searchScreenLayout.getChildren().add(infoBox);
        searchScreenLayout.getChildren().add(buttonControlBox);

        double sceneWidth = 800;
        double sceneHeight = 800;
        if (this.primaryStage.getScene() != null) {
            sceneWidth = Math.max(sceneWidth, this.primaryStage.getScene().getWidth());
            sceneHeight = Math.max(sceneHeight, this.primaryStage.getScene().getHeight());
        }

        Scene searchScene = new Scene(searchScreenLayout, sceneWidth, sceneHeight);
        this.primaryStage.setScene(searchScene);
        this.primaryStage.centerOnScreen();
    }


    public void startSearch() {
        this.solver = null;
        if ("Uniform Cost Search".equals(this.algorithm)) {
            solver = new UCS();
        } else if ("Greedy Best First Search".equals(this.algorithm)) {
            if ("Blocking Cars".equals(this.heuristic)) {
                solver = new GreedyBestFirst(new BlockingCarsHeuristic());
            } else if ("Manhattan To Exit".equals(this.heuristic)) {
                solver = new GreedyBestFirst(new ManhattanToExitHeuristic());
            } else if("Combined Heuristic".equals(this.heuristic)){
                solver = new GreedyBestFirst(new CombinedHeuristic());
            }
        } else if ("A* Search".equals(this.algorithm)) {
            if ("Blocking Cars".equals(this.heuristic)) {
                solver = new AStar(new BlockingCarsHeuristic());
            } else if ("Manhattan To Exit".equals(this.heuristic)) {
                solver = new AStar(new ManhattanToExitHeuristic());
            } else if("Combined Heuristic".equals(this.heuristic)){
                solver = new AStar(new CombinedHeuristic());
            }
        } else if ("Iterative Deepening Search".equals(this.algorithm)){
            solver = new IDDFS();
        } else if ("Hill Climbing Search".equals(this.algorithm)){
             if ("Blocking Cars".equals(this.heuristic)) {
                solver = new HillClimbing(new BlockingCarsHeuristic());
            } else if ("Manhattan To Exit".equals(this.heuristic)) {
                solver = new HillClimbing(new ManhattanToExitHeuristic());
            } else if("Combined Heuristic".equals(this.heuristic)){
                solver = new HillClimbing(new CombinedHeuristic());
            }
        }
        this.solution = this.solver.solve(this.board);
        if (this.solution.isEmpty()) {
            inputError("No solution found.", false);
            return;
        }
```

```
        animateSolution();
    }


    public void animateSolution() {
        Timeline timeline = new Timeline();
        double totalDuration = 0;

        for (int i = 0; i < this.solution.size(); i++) {
            final State currentState = this.solution.get(i);
            final State prevState = (i > 0) ? this.solution.get(i-1) : null;

            if (prevState != null) {
                String move = currentState.move;
                if (move != null && !move.equals("Start")) {
                    String[] parts = move.split("-");
                    if (parts.length >= 2) {
                        char pieceId = parts[0].charAt(0);
                        String direction = parts[1];
                        int steps = (parts.length > 2) ? Integer.parseInt(parts[2]) : 1;

                        for (int step = 1; step <= steps; step++) {
                            final int currentStep = step;
                            KeyFrame intermediateFrame = new KeyFrame(
                                Duration.millis(totalDuration + step * this.animationDelay),
                                event -> {
                                    Board intermediateBoard = Board.createIntermediateBoard(prevState.board,
currentState.board, pieceId, direction, currentStep, steps);

                                    VBox newBoardView = GUIHelper.drawBoard(intermediateBoard);
                                    if (newBoardView != null) {
                                        this.boardDisplayArea.getChildren().clear();
                                        newBoardView.setAlignment(Pos.CENTER);
                                        this.boardDisplayArea.getChildren().add(newBoardView);
                                    }
                                }
                            );
                            timeline.getKeyFrames().add(intermediateFrame);
                        }
                        totalDuration += steps * this.animationDelay;
                    }
                }
            } else {
                KeyFrame startFrame = new KeyFrame(Duration.ZERO, event -> {
                    VBox newBoardView = GUIHelper.drawBoard(currentState.board);
                    if (newBoardView != null) {
                        this.boardDisplayArea.getChildren().clear();
                        newBoardView.setAlignment(Pos.CENTER);
                        this.boardDisplayArea.getChildren().add(newBoardView);
                    }
                });
                timeline.getKeyFrames().add(startFrame);
            }
```

```
            if (i == this.solution.size() - 1) {
                this.board = currentState.board;
            }
        }

        timeline.setOnFinished(event -> {
            System.out.println("Animation finished.");
            solutionsFound();
        });

        System.out.println("Playing animation with " + timeline.getKeyFrames().size() + " frames.");
        timeline.play();
    }


    public void solutionsFound(){
        this.boardDisplayArea.getChildren().clear();

        VBox finalBoardView = GUIHelper.drawBoard(this.board);
        if (finalBoardView != null) {
            finalBoardView.setAlignment(Pos.CENTER);
            this.boardDisplayArea.getChildren().add(finalBoardView);
        }

        Label timeLabel = new Label("Execution time: " + this.solver.getRuntimeNano()/1e6 + "ms");
        timeLabel.setFont(Font.font("Arial", FontWeight.NORMAL, 18));

        Label nodesLabel = new Label("Nodes visited: " + this.solver.getNodes());
        nodesLabel.setFont(Font.font("Arial", FontWeight.NORMAL, 18));

        VBox infoBox = new VBox(5);
        infoBox.getChildren().addAll(timeLabel, nodesLabel);
        infoBox.setAlignment(Pos.CENTER);


        Label finalStateTitleLabel = new Label("Solution Found");
        finalStateTitleLabel.setFont(Font.font("Arial", FontWeight.BOLD, 20));
        finalStateTitleLabel.setPadding(new Insets(10, 0, 0, 0));

        Button backToTitleButton = GUIHelper.createButton("Back", e -> userInputs());
        Button replayButton = GUIHelper.createButton("Replay", e -> {
            if (this.solution != null && !this.solution.isEmpty()) {
                System.out.println("Replay button clicked!");
                ((Button)e.getSource()).setDisable(true);
                backToTitleButton.setDisable(true);

                animateSolution();
            } else {
                inputError("No solution path available to replay.", false);
            }
        });
        Button saveButton = GUIHelper.createButton("Save", e->{});
```

```
        saveButton.setOnAction(e -> {saveToFile(this.solution, this.algorithm, this.solver.getRuntimeNano()/1e6 ,
saveButton);});

        HBox buttonControlBox = new HBox(20, backToTitleButton, replayButton, saveButton);
        buttonControlBox.setAlignment(Pos.CENTER);
        buttonControlBox.setPadding(new Insets(0, 0, 50, 0));

        VBox solutionScreenLayout = new VBox(20);
        solutionScreenLayout.setAlignment(Pos.CENTER);
        solutionScreenLayout.setPadding(new Insets(20));
        solutionScreenLayout.getChildren().addAll(finalStateTitleLabel, this.boardDisplayArea, infoBox, buttonControlBox);

        double sceneWidth = 800;
        double sceneHeight = 700;
        if (this.primaryStage.getScene() != null) {
            sceneWidth = Math.max(sceneWidth, this.primaryStage.getScene().getWidth());
            sceneHeight = Math.max(sceneHeight, this.primaryStage.getScene().getHeight());
        }

        Scene solutionScene = new Scene(solutionScreenLayout, sceneWidth, sceneHeight);
        this.primaryStage.setScene(solutionScene);
        this.primaryStage.centerOnScreen();
    }


    public void saveToFile(List<State> solution, String algorithmName, double executionTime, Button callerButton){
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Save Solution As");

        // Default file name
        String initialFileName = "solution.txt";

        fileChooser.setInitialFileName(initialFileName);
        fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("Text Files (*.txt)", "*.txt"));

        Stage stage = (Stage) callerButton.getScene().getWindow();
        File fileToSave = fileChooser.showSaveDialog(stage);
        if (!fileToSave.getAbsolutePath().toLowerCase().endsWith(".txt")) fileToSave = new
File(fileToSave.getAbsolutePath() + ".txt");
        if (fileToSave != null) {
            try {
                String outputPath = OutputWriter.writeSolution( solution, algorithmName, executionTime,
fileToSave.getAbsolutePath());

                Alert successAlert = new Alert(AlertType.INFORMATION);
                successAlert.setTitle("Success");
                successAlert.setHeaderText(null);
                successAlert.setContentText("Solution saved successfully to:\n" + outputPath);
                successAlert.showAndWait();
            } catch (Exception ex) {
                Alert errorAlert = new Alert(AlertType.ERROR);
                errorAlert.setTitle("Error");
                errorAlert.setHeaderText("An unexpected error occurred while saving");
```

```
                errorAlert.setContentText(ex.getMessage());
                errorAlert.showAndWait();
            }
        } else {
            System.out.println("Save operation cancelled by user.");
        }
    }


    // initially just for error, but used for no solutions too
    public void inputError(String errorMessage, boolean isError) {
        String label;
        if (isError) {
            label = "Error";
        } else {
            label = "Result:";
        }
        Label errorTitleLabel = new Label(label);
        errorTitleLabel.setFont(Font.font("Arial", FontWeight.BOLD, 24));
        errorTitleLabel.setTextFill(Color.RED);

        Label errorMessageLabel = new Label(errorMessage);
        errorMessageLabel.setFont(Font.font("Arial", 16));
        errorMessageLabel.setWrapText(true);
        errorMessageLabel.setTextAlignment(TextAlignment.CENTER);
        errorMessageLabel.setMaxWidth(400);

        Button backButton = GUIHelper.createButton("Back", e-> {userInputs();});

        VBox errorLayout = new VBox(30);
        errorLayout.getChildren().addAll(errorTitleLabel, errorMessageLabel, backButton);
        errorLayout.setAlignment(Pos.CENTER);
        errorLayout.setPadding(new Insets(40));

        Scene errorScene = new Scene(errorLayout, 500, 300);
        this.primaryStage.setScene(errorScene);
        this.primaryStage.centerOnScreen();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

GUIHelper.java

```
public class GUIHelper {
    static int INPUT_BTN_WIDTH = 120;
    static int INPUT_BTN_HEIGHT = 30;
    final static private Map<Character, Color> pieceColors = new HashMap<>();
    static {
        pieceColors.put('A', Color.CYAN); pieceColors.put('B', Color.MAGENTA); pieceColors.put('C', Color.YELLOW);
        pieceColors.put('D', Color.ORANGE); pieceColors.put('E', Color.SPRINGGREEN); pieceColors.put('F', Color.DODGERBLUE);
        pieceColors.put('G', Color.VIOLET); pieceColors.put('H', Color.GOLD); pieceColors.put('I', Color.LIGHTPINK);
        pieceColors.put('J', Color.TURQUOISE); pieceColors.put('L', Color.SALMON); pieceColors.put('M', Color.KHAKI);
```

```
        pieceColors.put('N', Color.SKYBLUE); pieceColors.put('O', Color.PLUM); pieceColors.put('Q', Color.LIGHTGREEN);
        pieceColors.put('R', Color.CORAL); pieceColors.put('S', Color.HOTPINK); pieceColors.put('T', Color.DARKVIOLET);
        pieceColors.put('U', Color.BEIGE); pieceColors.put('V', Color.ORANGERED); pieceColors.put('W', Color.BROWN);
        pieceColors.put('X', Color.BLUE); pieceColors.put('Y', Color.AZURE); pieceColors.put('Z', Color.OLIVEDRAB);
    }

    public static Button createButton(String title, EventHandler<ActionEvent> action) {
        Button button = new Button(title);
        button.setPrefWidth(INPUT_BTN_WIDTH);
        button.setPrefHeight(INPUT_BTN_HEIGHT);
        button.setOnAction(action);

        String modernStyle = "-fx-background-color:rgb(0, 128, 255); " +
                             "-fx-text-fill: white; " +
                             "-fx-font-size: 14px; " +
                             "-fx-font-weight: bold; " +
                             "-fx-background-radius: 5px; " +
                             "-fx-border-radius: 5px; " +
                             "-fx-padding: 8px 18px; " +
                             "-fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.1), 5, 0, 0, 1);";
        button.setStyle(modernStyle);

        String hoverStyle = "-fx-background-color:rgb(0, 81, 194); " +
                            "-fx-text-fill: white; " +
                            "-fx-font-size: 14px; " +
                            "-fx-font-weight: bold; " +
                            "-fx-background-radius: 5px; " +
                            "-fx-border-radius: 5px; " +
                            "-fx-padding: 8px 18px; " +
                            "-fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.1), 5, 0, 0, 1);";

        String pressedStyle = "-fx-background-color:rgb(0, 98, 255); " +
                              "-fx-text-fill: white; " +
                              "-fx-font-size: 14px; " +
                              "-fx-font-weight: bold; " +
                              "-fx-background-radius: 5px; " +
                              "-fx-border-radius: 5px; " +
                              "-fx-padding: 8px 18px; " +
                              "-fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.2), 3, 0, 0, 1);";

        button.setOnMouseEntered(e -> button.setStyle(hoverStyle));
        button.setOnMouseExited(e -> button.setStyle(modernStyle));
        button.setOnMousePressed(e -> button.setStyle(pressedStyle));
        button.setOnMouseReleased(e -> button.setStyle(hoverStyle));

        return button;
    }

    public static VBox drawBoard(Board board) {
        GridPane gridPane = new GridPane();
        gridPane.setAlignment(Pos.CENTER);
        gridPane.setPadding(new Insets(10));
        gridPane.setHgap(1);
        gridPane.setVgap(1);

        int displayRows = board.rows;
        int displayCols = board.cols;
```

```
        double cellSize = 50;
        gridPane.setMinSize(displayCols * cellSize, displayRows * cellSize);

        for (int i = 0; i < displayRows; i++) {
            for (int j = 0; j < displayCols; j++) {
                StackPane cellPane = new StackPane();
                cellPane.setPrefSize(cellSize, cellSize);

                Rectangle cellRect = new Rectangle(cellSize, cellSize);
                Label pieceLabel = new Label("");
                pieceLabel.setFont(Font.font("Arial", FontWeight.BOLD, cellSize * 0.6));

                char pieceChar = (board.grid != null && i < board.grid.length && j < board.grid[i].length)? board.grid[i][j] : '.';
                pieceLabel.setText(String.valueOf(pieceChar));

                if (pieceChar == '.') {
                    pieceLabel.setText("");
                    cellRect.setFill(Color.LIGHTGRAY);
                    cellRect.setStroke(Color.DARKGRAY);
                } else if (pieceChar == 'P') {
                    cellRect.setFill(Color.RED);
                    cellRect.setStroke(Color.DARKRED);
                    pieceLabel.setTextFill(Color.WHITE);
                } else if (pieceChar == 'K') {
                    cellRect.setFill(Color.LIME);
                    cellRect.setStroke(Color.GREEN);
                    pieceLabel.setTextFill(Color.BLACK);
                }else if (pieceChar == ' '){
                    continue;
                }else {
                    Color pieceColor = pieceColors.get(pieceChar);
                    cellRect.setFill(pieceColor);
                    cellRect.setStroke(pieceColor.darker());

                    // contrasting text color
                    if (pieceColor.getBrightness() * pieceColor.getSaturation() < 0.15) {
                        pieceLabel.setTextFill(Color.WHITE);
                    } else {
                        pieceLabel.setTextFill(Color.BLACK);
                    }
                }
                cellPane.getChildren().addAll(cellRect, pieceLabel);
                gridPane.add(cellPane, j, i);
            }
        }

        VBox boardLayout = new VBox(20, gridPane);
        boardLayout.setAlignment(Pos.CENTER);
        boardLayout.setPadding(new Insets(20));

        return boardLayout;
    }
}
```

3.2.6 Package utils

Direction.java

```java
public enum Direction {
    UP(-1, 0),
    DOWN(1, 0),
    LEFT(0, -1),
    RIGHT(0, 1);

    public final int dx;
    public final int dy;

    Direction(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
    }

    public static Direction fromString(String s) {
        return switch (s.toLowerCase()) {
            case "up" -> UP;
            case "down" -> DOWN;
            case "left" -> LEFT;
            case "right" -> RIGHT;
            default -> null;
        };
    }
}
```

InputParser.java

```java
public class InputParser {
    private static int getKColIdx(String line) {
        if (line == null) return -1;
        int kPos = -1, kCount = 0;
        for (int i = 0; i < line.length(); i++) {
            char c = line.charAt(i);
            if (c == 'K') { kPos = i; kCount++; }
            else if (c != ' ') return -1;
        }
        return (kCount == 1) ? kPos : -1;
    }


    public static Board parseFromFile(String path) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(path));
        Position exit = null;
        char[][] grid;
        int usableRows, usableCols;
        Map<Character, Piece> pieces = new HashMap<>();
        Set<Position> allCellsForPieces = new HashSet<>();
        Set<Character> validSymbols = new HashSet<>();
        boolean hasPrimaryPiece = false;

        try {
            String dimsLine = br.readLine();
```

```
            if (dimsLine == null) throw new IOException("Missing dimensions line.");
            String[] sizeParts = dimsLine.split(" ");
            if (sizeParts.length != 2) throw new IOException("Invalid dimensions format: " + dimsLine);

            try {
                usableRows = Integer.parseInt(sizeParts[0]);
                usableCols = Integer.parseInt(sizeParts[1]);

                if (usableRows <= 0 || usableCols <= 0) {
                    throw new IOException("Board dimention can't be negative: " + usableRows + "x" + usableCols);
                }
            } catch (NumberFormatException e) {
                throw new IOException("Board dimention must be integer: " + dimsLine);
            }

            grid = new char[usableRows + 2][usableCols + 2];
            for (char[] row : grid) Arrays.fill(row, ' ');

            String pieceCountLine = br.readLine();
            if (pieceCountLine == null) throw new IOException("Missing piece count or first grid line.");

            int pieceCount = -1;
            try {
                pieceCount = Integer.parseInt(pieceCountLine.trim());
                if (pieceCount < 0) {
                    throw new IOException("The number of character (piece) has to be positive: " + pieceCount);
                }
            } catch (NumberFormatException e) {
                // Bukan angka, anggap sebagai baris pertama grid
                pieceCountLine = null;
            }

            String lineForKAboveOrFirstGrid = pieceCountLine != null ? br.readLine() : pieceCountLine;
            if (lineForKAboveOrFirstGrid == null) throw new IOException("Missing K_above or first grid line.");
            int KColInLine = getKColIdx(lineForKAboveOrFirstGrid);

            int totalKCount = 0;

            if (KColInLine != -1) {
                if (KColInLine >= usableCols) throw new IOException("K at column " + KColInLine + " is outside playable width " +
usableCols);
                exit = new Position(0, KColInLine + 1);
                grid[0][KColInLine + 1] = 'K';
                lineForKAboveOrFirstGrid = null;
                totalKCount++;
            }

            int actualRows = 0;

            for (int i = 0; i < usableRows; i++) {
                String currentLine = (i == 0 && lineForKAboveOrFirstGrid != null) ? lineForKAboveOrFirstGrid : br.readLine();
                while (currentLine != null && currentLine.trim().isEmpty()) {
                    currentLine = br.readLine();
                }
                if (currentLine == null) throw new IOException("EOF: Expected " + usableRows + " grid rows, found " + i);
                actualRows++;
```

```
            char[] lineChars;
            String originalLine = currentLine;
            currentLine = currentLine.replaceAll(" ", "");

            // K di kiri
            boolean hasLeftK = false;
            if (currentLine.length() > 0 && currentLine.charAt(0) == 'K') {
                if (totalKCount > 0) throw new IOException("Multiple K definitions found (total so far: " + totalKCount + ")");
                hasLeftK = true;
                exit = new Position(i + 1, 0);
                grid[i+1][0] = 'K';
                currentLine = currentLine.substring(1);
                totalKCount++;
            }

            // K di kanan
            boolean hasRightK = false;
            if (currentLine.length() > 0 && currentLine.charAt(currentLine.length() - 1) == 'K') {
                if (totalKCount > 0) throw new IOException("Multiple K definitions found (total so far: " + totalKCount + ")");
                hasRightK = true;
                exit = new Position(i + 1, usableCols + 1);
                grid[i+1][usableCols+1] = 'K';
                currentLine = currentLine.substring(0, currentLine.length() - 1);
                totalKCount++;
            }

            if (currentLine.length() == usableCols) {
                lineChars = currentLine.toCharArray();
                for (int j = 0; j < usableCols; j++) {
                    char c = lineChars[j];
                    // Validasi karakter
                    if (!isValidChar(c)) {
                        throw new IOException("Invalid character '" + c + "' at (" + (i+1) + "," + (j+1) + " position)");
                    }
                    if (c == 'P') hasPrimaryPiece = true;
                    if (c != ' ' && c != '.' && c != 'K') validSymbols.add(c);
                    grid[i+1][j+1] = c;
                }
            } else {
                String errorMsg = "Row " + (i+1) + " has incorrect length. ";
                if (hasLeftK) errorMsg += "Found K at left, ";
                if (hasRightK) errorMsg += "Found K at right, ";
                errorMsg += "Current content length: " + currentLine.length() + ", expected: " + usableCols;
                throw new IOException(errorMsg);
            }
        }

        if (actualRows != usableRows) {
            throw new IOException("The number of row doesn't match: expected " + usableRows + ", but only found " + actualRows);
        }

        if (exit == null) {
            String kBelowLine = br.readLine();
            if (kBelowLine != null) {
                KColInLine = getKColIdx(kBelowLine);
                if (KColInLine != -1) {
                    if (KColInLine >= usableCols) throw new IOException("K at column " + KColInLine + " is outside playable
```

```
width " + usableCols);
                        exit = new Position(usableRows + 1, KColInLine + 1);
                        grid[usableRows+1][KColInLine+1] = 'K';
                        totalKCount++;
                    } else throw new IOException("Extra data after grid, not valid line for K");
                }
            }

            if (totalKCount == 0) {
                throw new IOException("There is no 'K' character (exit) on the board");
            } else if (totalKCount > 1) {
                throw new IOException("There is more than one 'K' character (exit), found: " + totalKCount);
            }

            if (!hasPrimaryPiece) {
                throw new IOException("There is no 'P' (primary piece) on the board");
            }

            int pCount = 0;
            Position pStart = null;

            for (int r = 0; r < usableRows; r++) {
                for (int c = 0; c < usableCols; c++) {
                    char val = grid[r+1][c+1];
                    Position currentPos = new Position(r + 1, c + 1);

                    if (val == ' ' || val == '.') continue;
                    if (allCellsForPieces.contains(currentPos)) continue;

                    if (val == 'K') {
                        throw new IOException("Character 'K' found inside playable area");
                    }

                    if (pieces.containsKey(val)) {
                        throw new IOException("Wrong piece configuration for '" + val + "' found!");
                    }

                    List<Position> currPieceOccupied = new ArrayList<>();
                    currPieceOccupied.add(currentPos);
                    int len = 1;
                    boolean horizontal = false;

                    if (c + 1 < usableCols && grid[r+1][(c+1)+1] == val) {
                        horizontal = true;
                        for (int cc = c + 1; cc < usableCols; cc++) {
                            if (grid[r+1][cc+1] == val) {
                                Position nextPos = new Position(r+1, cc+1);
                                if(allCellsForPieces.contains(nextPos)) throw new IOException("Piece '"+val+"' at "+currentPos+"
overlaps with another piece at "+nextPos);
                                currPieceOccupied.add(nextPos);
                                len++;
                            } else break;
                        }
                    } else if (r + 1 < usableRows && grid[(r+1)+1][c+1] == val) {
                        for (int rr = r + 1; rr < usableRows; rr++) {
                            if (grid[rr+1][c+1] == val) {
                                Position nextPos = new Position(rr+1, c+1);
```

```
                            if(allCellsForPieces.contains(nextPos)) throw new IOException("Piece '"+val+"' at "+currentPos+"
overlaps with another piece at "+nextPos);
                            currPieceOccupied.add(nextPos);
                            len++;
                        } else break;
                    }
                }

                for(Position p : currPieceOccupied) {
                    if(!allCellsForPieces.add(p)) {
                         throw new IOException("Cell " + p + " for piece '" + val + "' was already processed, indicating overlap
or malformed piece.");
                    }
                }

                if (val == 'P') {
                    pCount++;
                    pStart = currentPos;
                }

                pieces.put(val, new Piece(val, currentPos, len, horizontal));
            }
        }

        Piece primaryPiece = pieces.get('P');
        if (primaryPiece != null) {
            boolean canReachExit = false;

            if (primaryPiece.isHorizontal) {
                if ((exit.col == 0 || exit.col == usableCols + 1) &&
                    exit.row == primaryPiece.start.row) {
                    canReachExit = true;
                }
            }
            else {
                if ((exit.row == 0 || exit.row == usableRows + 1) &&
                    exit.col == primaryPiece.start.col) {
                    canReachExit = true;
                }
            }

            if (!canReachExit) {
                throw new IOException("Primary piece 'P' can't reach the exit 'K' (Make sure no extra whitespace on the
board!)");
            }
        }
        for (char i = 'A'; i <= 'Z'; i++) {
            if ((char) i != 'K' && pieces.containsKey((char)i)){
                if (pieces.get((char)i).length < 2 || pieces.get(i).length > 3){
                    throw new IOException("Invalid piece length " + (char) i);
                }
            }
        }
        return new Board(usableRows + 2, usableCols + 2, grid, pieces, exit);
    } finally {
        br.close();
    }
```

```
    }


    private static boolean isValidChar(char c) {
        return Character.isLetterOrDigit(c) || c == ' ' || c == '.' || c == 'K';
    }
}
```

OutputWriter.java

```
public class OutputWriter {
    public static String writeSolution(List<State> solution, String algorithmName, double executionTime, String outputPath) throws
IOException {
        try (PrintWriter writer = new PrintWriter(new FileWriter(outputPath))) {
            writer.println("============================================================");
            writer.println("  RUSH HOUR GAME SOLUTION USING " + algorithmName.toUpperCase());
            writer.println("============================================================");
            writer.println();
            writer.println("Number of moves: " + (solution.size() - 1));
            writer.println("Execution time: " + String.format("%.3f ms", executionTime));
            writer.println();
            writer.println("------------------------------------------------------------");
            writer.println("SOLUTION STEPS:");
            writer.println("------------------------------------------------------------");
            State state;
            for (int i = 1; i < solution.size(); i++) {
                state = solution.get(i);
                writer.println("Move " + i + ": " + state.move);
            }
            writer.println("\n");
            for (int i = 0; i < solution.size(); i++){
                state = solution.get(i);
                writer.print("\nMove " + i + ": " + state.move);
                for (int r = 0; r < state.board.rows; r++) {
                    for (int c = 0; c < state.board.cols; c++) {
                        writer.print(state.board.grid[r][c]);
                    }
                    writer.println();
                }
            }
            writer.println("------------------------------------------------------------");
            writer.println("\nSolution finished. Game solved successfully!");
        }

        return outputPath;
    }

}
```

3.3 Implementasi Bonus

1. **Implementasi Algoritma Alternatif**

   Untuk spesifikasi bonus algoritma alternatif selain UCS, GBFS, dan A*. Kami mengimplementasikan dua algoritma alternatif, yaitu *Iterative Deepening Depth First Search*

dan *Hill Climbing Search*. Untuk penjelasan lebih lanjut mengenai dua algoritma tersebut, telah dijabarkan pada Analisis Algoritma.

2. **Implementasi Heuristik Alternatif**

   Heuristik yang diimplementasikan pada program ada 3, yaitu heuristik *Blocking Cars, Manhattan Distance* dan *Combined Heuristic,* yaitu gabungan dari kedua heuristik sebelumnya.

   Heuristik *Blocking Cars* menghitung jumlah mobil yang secara langsung menghalangi jalur *primary piece* untuk mencapai pintu keluar. Semakin sedikit mobil yang menghalangi, semakin baik nilai heuristiknya, karena ini mengindikasikan *primary piece* lebih bebas untuk bergerak ke tujuan.

   Heuristik *Manhattan Distance* mengukur jarak mobil target ke pintu keluar dengan menghitung jumlah langkah horizontal dan vertikal yang diperlukan jika tidak ada mobil lain di papan. Heuristik ini memberikan perkiraan tentang seberapa jauh *primary piece* dari tujuannya, namun mengabaikan kompleksitas pergerakan mobil-mobil lain yang mungkin perlu digeser terlebih dahulu.

   Adapun *Combined Heuristic* menggabungkan kedua heuristik dengan perhitungan sebagai berikut:

   $$h(n) = \text{blocingCars}(n) + \text{manhattan}(n)$$

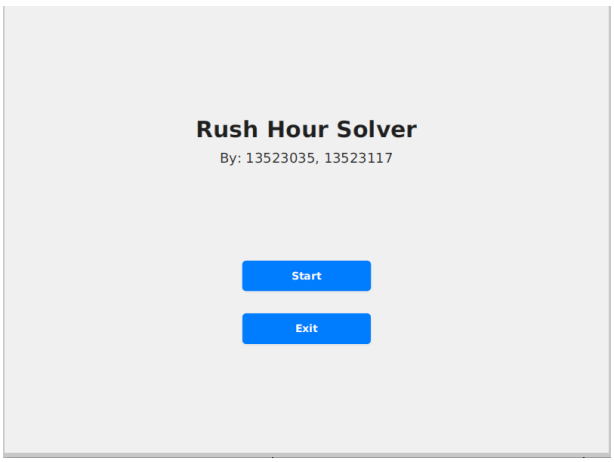
   Heuristik ini memberikan estimasi yang lebih informatif dengan mempertimbangkan jarak pintu keluar serta hambatan di jalurnya.

3. **Implementasi GUI**

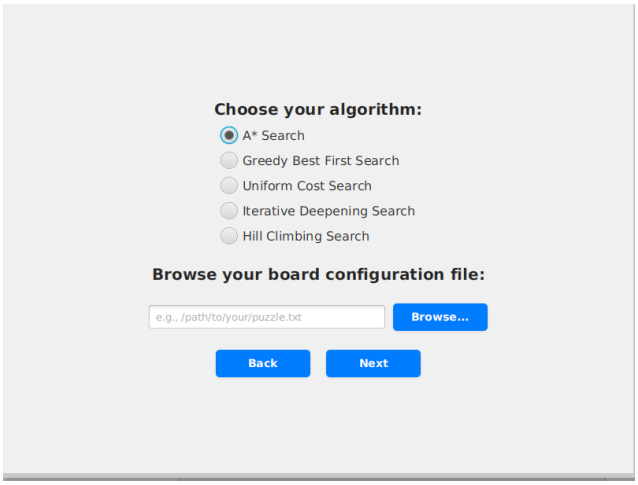
   Pengguna menggunakan GUI sebagai cara berinteraksi dengan program. Adapun implementasi GUI menggunakan *framework* JavaFX dalam bahasa Java. GUI memiliki fitur-fitur seperti pemilihan algoritma dan heuristic melalui *radio buttons* dan pemilihan file *input* dan *output* melalui *FileChooser* yang disediakan JavaFX. Selain itu, juga tersedia animasi hasil *pathfinding* yang menunjukkan *state* dari board pada setiap langkah permainan.

   Layar-layar pada GUI:

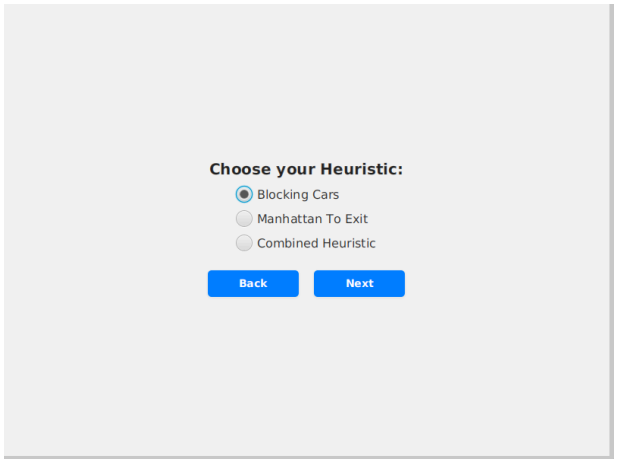
Start Screen:



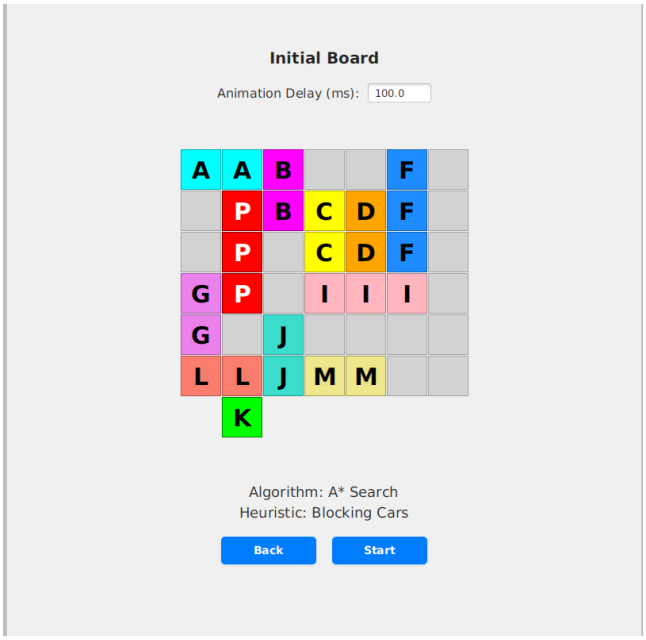
Input Algoritma dan File:



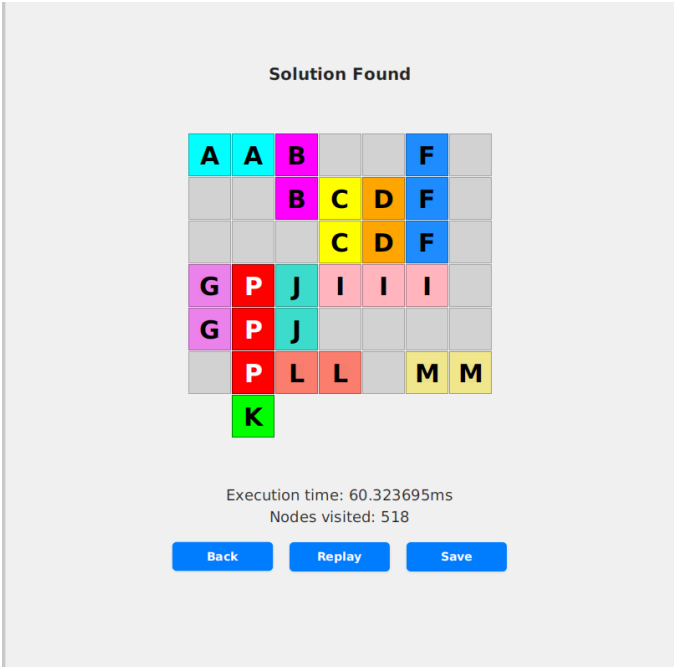
Pemilihan Heuristik:



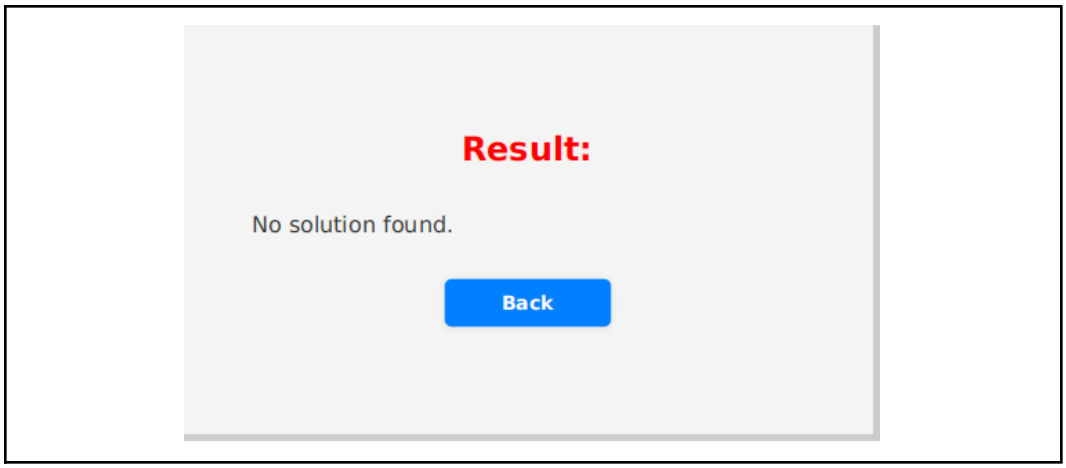
Memulai Pencarian:



Jika ditemukan solusi:



Jika tidak ditemukan solusi:



IV. Eksperimen dan Analisis

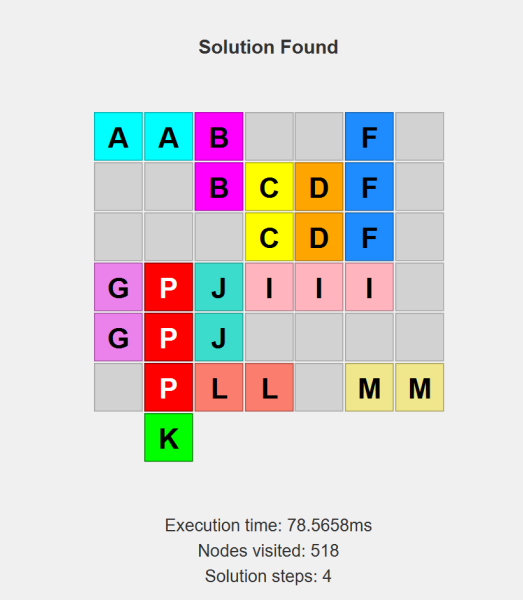
Pada bagian ini, akan dijabarkan berbagai hasil eksperimen berdasarkan konfigurasi papan serta algoritma dan heuristik yang digunakan. Untuk hasil output yang akan ditampilkan hanyalah tangkapan layar *state* terakhir dari papan, serta *runtime* dan *nodes* yang dikunjungi. Jika ingin melihat hasil secara keseluruhan, dapat melihat hasil output pada repository. Selain itu, konfigurasi yang digunakan hanyalah **konfigurasi papan yang benar**.

4.1 Uji 1: Perbedaan Algoritma Ber-Heuristik

Konfigurasi 1

```
6 7
11
AAB..F.
.PBCDF.
.P.CDF.
GP.III.
G.J....
LLJMM..
 K
```

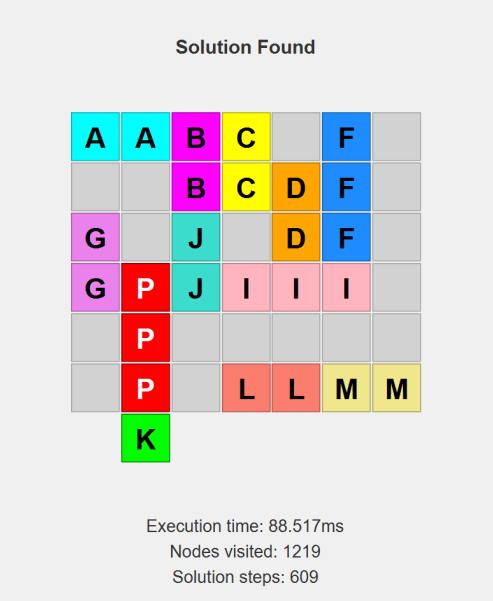
Algoritma : A* search, Heuristik : Blocking Cars
Hasil:



File : result_test1_AStar.txt

Algoritma : GBFS, Heuristik : Blocking Cars

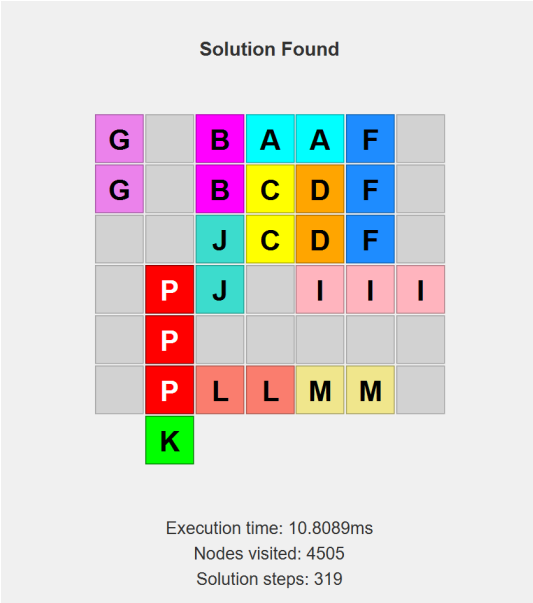
Hasil:



File : result_test1_GBFS.txt

Algoritma : Hill Climbing Search, Heuristik : Blocking Cars

Hasil:



File : result_test1_HillClimbing.txt

Analisis Hasil

Berdasarkan hasil pengujian yang dilakukan pada tiga algoritma yang berbeda, yakni *A* Search*, *Greedy Best First Search*, dan *Hill Climbing Search* dengan satu heuristik yang sama, diperoleh bahwa *Hill Climbing Search* adalah algoritma yang paling cepat dalam menemukan solusi permasalahan permainan *Rush Hour* ini. Hal ini karena sifat dasar *Hill Climbing Search* yang sangat lokal dan deterministik. *Hill Climbing Search* hanya fokus pada satu tetangga terbaik berdasarkan nilai heuristik *h(n)*. Saat menemukan langkah terbaik, algoritma langsung berpindah ke sana tanpa menyimpan riwayat jalur ataupun mempertimbangkan alternatif lain. Sedangkan *A* Search* menggunakan *priority queue* berdasarkan *total cost* dan GBFS menggunakan *priority queue* berdasarkan heuristiknya. Meskipun GBFS lebih fokus daripada A*, algoritma ini tetap mengkonsumsi waktu dan memori lebih besar daripada Hill Climbing yang "*single-minded*".

Algoritma *A* Search* berhasil menemukan solusi dengan panjang hanya 4 langkah, tetapi algoritma ini harus mengunjungi 518 node terlebih dahulu. Hal ini karena A* secara sistematis mengevaluasi banyak kemungkinan jalur dengan meminimalkan fungsi total biaya $f(n) = g(n) + h(n)$. Meskipun *g(n)* membuatnya mengeksplorasi banyak cabang, algoritma A* Search menjamin solusi yang ditemukan adalah solusi terpendek dan optimal. Jadi, meskipun prosesnya lebih mahal secara waktu dan eksplorasi, algoritma *A* Search* menghasilkan solusi yang paling efisien dari sisi langkah.

Sebaliknya, algoritma GBFS lebih fokus pada heuristik murni (*h(n)*), tanpa mempertimbangkan jarak

dari node awal. Hal ini menyebabkan GBFS mengejar goal secara agresif, sering kali memilih jalur yang tampak menjanjikan berdasarkan estimasi heuristik saja, tetapi jalan yang ditelusuri adalah jalan yang panjang. Akibatnya, solusi yang ditemukan memiliki panjang 609 langkah, jauh lebih panjang dibanding A* dan membutuhkan eksplorasi 1219 node karena banyaknya cabang yang harus dipertimbangkan ulang setelah jalur terdekat ternyata tidak benar-benar efisien. Hal ini memperlihatkan bahwa GBFS tidak menjamin optimalitas dan sangat tergantung pada bentuk fungsi heuristic.

Sementara itu, algoritma *Hill Climbing Search* menunjukkan hasil yang sangat cepat, meski mengunjungi 4505 node dan menemukan solusi dalam 319 langkah. Ini terjadi karena algoritma *Hill Climbing Search* hanya melihat satu tetangga terbaik setiap saat, tanpa menyimpan state lain. Meskipun solusi yang ditemukan tidak sependek milik A*, tapi tidak seburuk GBFS dan jumlah langkahnya relatif lebih panjang karena algoritma ini tidak menjamin menemukan solusi terpendek, hanya solusi tercepat berdasarkan local improvement. Algoritma *Hill Climbing Search* juga bisa terjebak pada local optimum, tetapi pada kasus ini, algoritma ini berhasil mencapai goal tanpa harus melakukan *backtracking* atau *restart*.

4.2 Uji 2: Perbedaan Heuristik

Konfigurasi 2

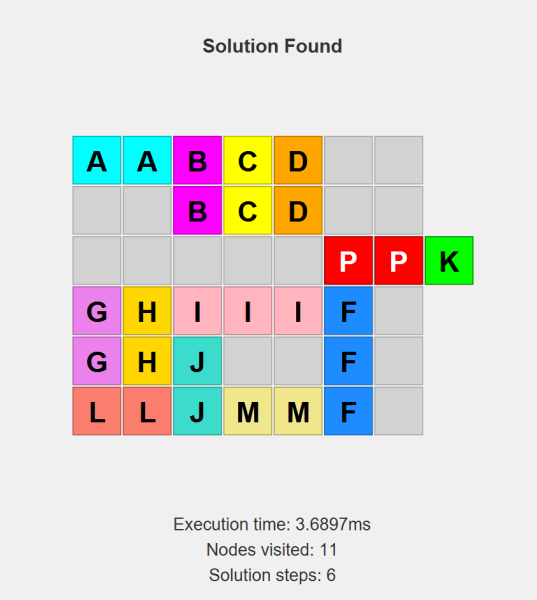
```
6 7
12
AAB..F.
..BCDF.
.PPCDF.K
GH.III.
GHJ....
LLJMM..
```

Algoritma : A*, Heuristik : Blocking Cars
Hasil:

File : result_test2_BlockingCars.txt

Algoritma : A* , Heuristik : Manhattan to Exit

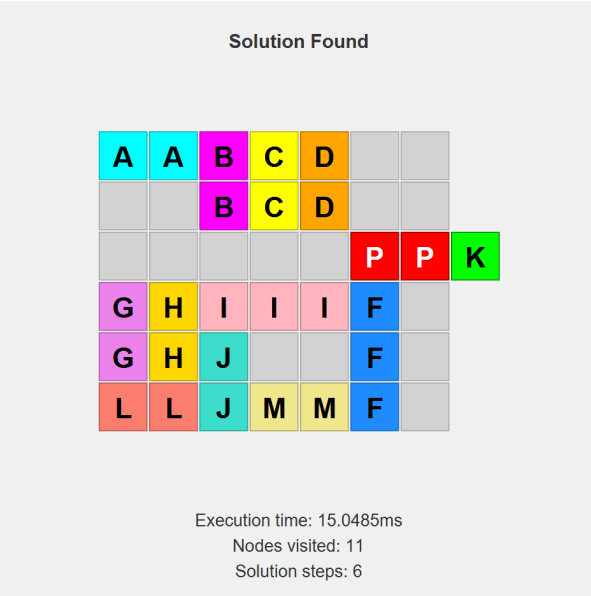
Hasil:



File : result_test2_ManhattanToExit.txt

Algoritma : A*, Heuristik : Combined Heuristic

Hasil :



File : result_test2_Combined.txt

Analisis Hasil

Heuristik *Blocking Cars* menghitung jumlah kendaraan yang menghalangi jalan keluar primary piece karena heuristik ini hanya mempertimbangkan "jumlah penghalang" dan bukan posisi akhir, maka algoritma ini akan mengarahkan pencarian pada langkah-langkah yang segera membebaskan jalur keluar tanpa memperhatikan apakah kendaraan tersebut harus digeser jauh atau tidak. Akibatnya, algoritma A* dengan *Blocking Cars* memerlukan lebih banyak node untuk dievaluasi (15 node), tetapi mampu menemukan solusi optimal dengan hanya 5 langkah. Meskipun lebih lambat dari sisi waktu (31 ms), ini terjadi karena banyak node perlu dipertimbangkan untuk memastikan semua penghalang dibersihkan secara efisien. A* mengeksplorasi lebih luas untuk memastikan kombinasi $g(n) + h(n)$ benar-benar minimum.

Heuristik *Manhattan to Exit* hanya menghitung jarak horizontal antara ujung *primary piece* ke posisi pintu keluar, tanpa mempertimbangkan apakah jalur tersebut diblokir atau tidak karena heuristik ini sangat sederhana dan menghitung jarak lurus, hasilnya cenderung lebih cepat (3 ms) karena A* dipandu ke arah goal secara langsung. Namun, karena tidak mempertimbangkan penghalang di jalur, pencarian justru menghasilkan solusi sub-optimal sepanjang 6 langkah. Meski demikian, pencariannya lebih fokus dan sempit, hanya 11 node yang perlu dikunjungi.

Heuristik *Combined* ini merupakan gabungan dari *Blocking Cars* dan *Manhattan to Exit*. Algoritma ini bekerja dengan menjumlahkan keduanya, heuristik ini mendapatkan informasi yang lebih seimbang, yaitu sejauh mana goal dan berapa banyak rintangan di jalur. Namun karena A* memprioritaskan $f(n) =$

g(n) + h(n), nilai *h(n)* yang terlalu besar kadang bisa menyebabkan eksplorasi sedikit menyimpang dari jalur optimal. Dalam pengujian ini, *Combined Heuristic* menghasilkan jumlah *node visited* yang sama seperti *Manhattan to Exit* (11 node) dan menghasilkan solusi dengan panjang 6 langkah, sama seperti *Manhattan to Exit*, tetapi dengan waktu eksekusi lebih cepat dari *Blocking Cars* dan lebih stabil dari *Manhattan to Exit* (15 ms).

4.3 Uji 3: Informed vs Uninformed

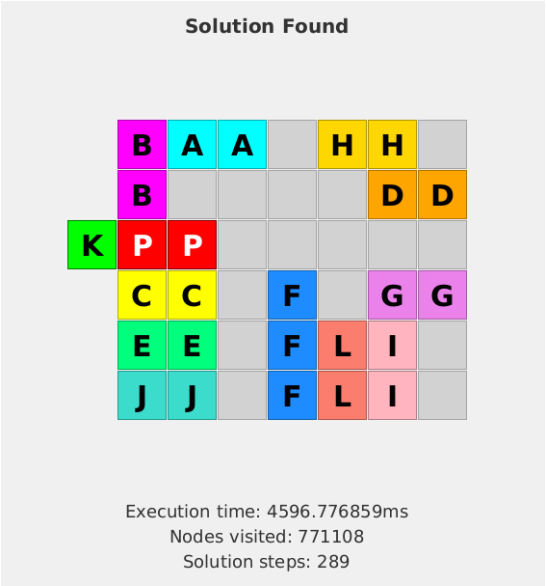
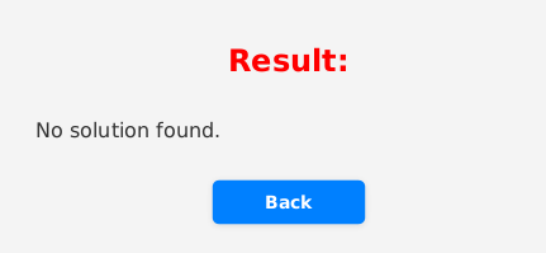
Konfigurasi 3

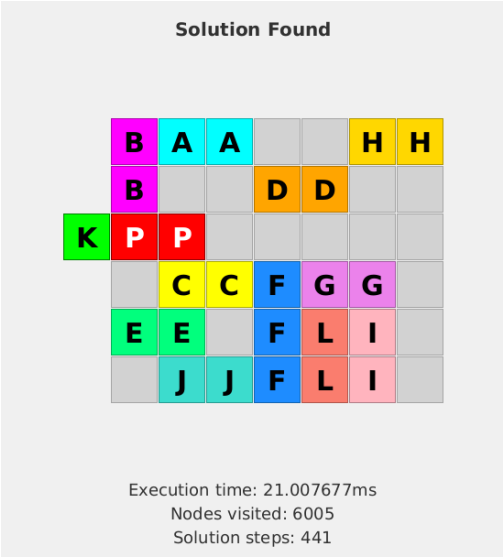
```
6 7
11
 .AAHH..
 BDD.L..
KB..FLPP
 CC.FGG.
 EE.F.I.
 .JJ..I.
```

Algoritma : A*, Heuristik : Combined Heuristic

Hasil:

File: result_test3_A*.txt

Algoritma : GBFS, Heuristik : Combined Heuristic
Hasil:    File: result_test3_GBFS.txt
Algoritma : Hill Climbing Search, Heuristik : Combined Heuristic

Algoritma : Hill Climbing Search, Heuristik : Blocking Cars
Hasil :



File: result_test3_HillClimbing.txt

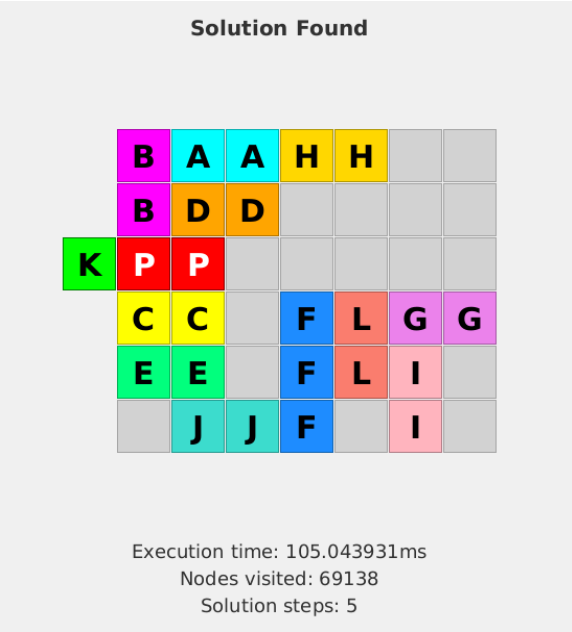
Algoritma : UCS

Hasil:

File: result_test3_UCS.txt

Algoritma : IDDFS

Hasil :



File: result_test3_IDDFS.txt

Analisis Hasil :

Berdasarkan pengujian, algoritma tercepat, dengan *node* yang dikunjungi kedua tersedikit adalah *Hill Climbing* jika menggunakan heuristik *Blocking Cars*, sedangkan dengan heuristik *Combined Search* (juga dengan *Manhattan*), *Hill Climbing* tidak berhasil menemukan solusi (karena terjebak pada suatu kasus). Selain itu, solusi yang ditemukan juga tidak optimal. Selain *Hill Climbing,* algoritma *informed search* lainnya (GBFS dan A*) cenderung tidak efisien baik waktu atau jumlah *node* yang dikunjungi, dengan GBFS mengunjungi jauh lebih banyak *node* dibanding semua algoritma lain, ini mungkin terjadi karena GBFS mendapat nilai heuristik yang *misleading* karena GBFS tidak menghitung biaya untuk mencapai suatu simpul.Untuk solusi, A* berhasil mendapatkan solusi yang optimal sedangkan GBFS tidak berhasil. Untuk *uninformed search*, baik UCS maupun IDDFS efisien secara waktu, dimana kedua algoritma mendapatkan solusi dalam waktu yang relatif dekat. Selain itu, kedua algoritma juga mendapat solusi yang optimal. Perbedaan dari kedua algoritma tersebut adalah pada jumlah *node* yang dikunjungi, dimana IDDFS mengunjungi jauh lebih banyak *node* karena sifatnya yang iteratif, sehingga banyak simpul yang dikunjungi berkali-kali.

4.4 Uji 4 : Papan Kompleks

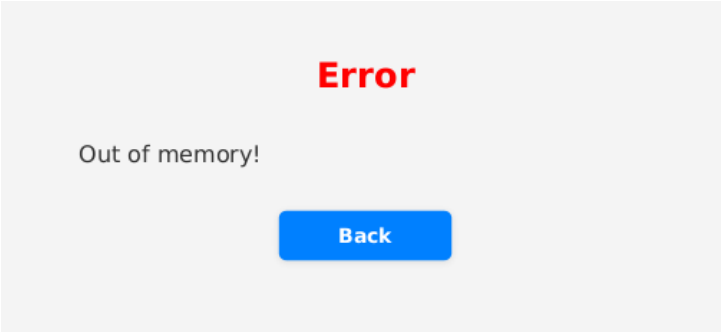
Konfigurasi 4

```
12 12
24
    K
...AAA..CC..
```

```
.....B......
DDD..B...RR.
...EEE..S...
...G....S...
.O.G.FF.S...
.O.GHH.JJJ..
NN...IIYM.TT
...QQ..YM...
....PLL.M...
..VVP..UUU..
.WW.P.XX.ZZ.
```

Algoritma : A* Search, Heuristik : Manhattan To Distance
Hasil: 
Algoritma : UCS
Hasil: 
Algoritma : GBFS, Heuristik : Combined Heuristic

Hasil:



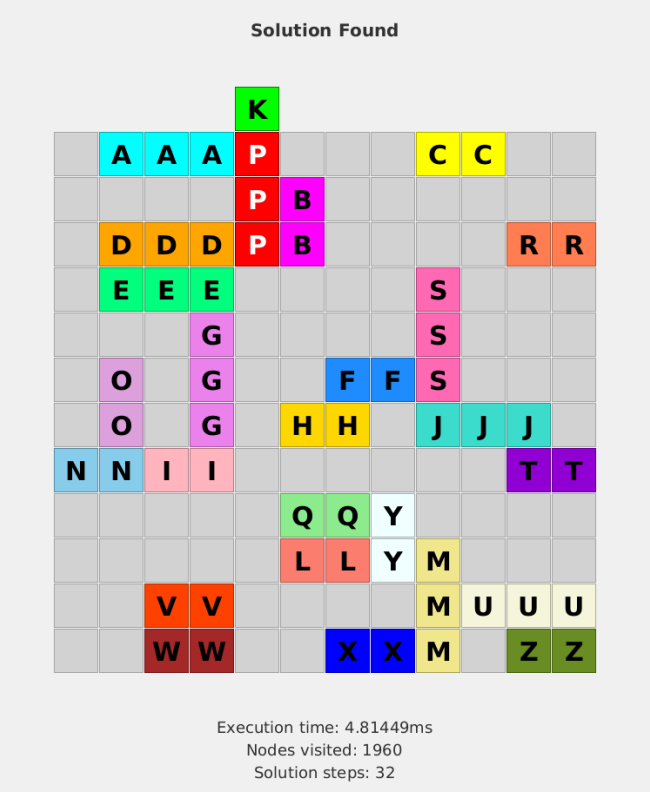
Algoritma : IDDFS

Hasil:

File: result_test5_IDDFS.txt

Algoritma : Hill Climbing Search, Heuristik : Combined Heuristic

Hasil:



File: result_test5_HillClimbing.txt

Analisis Hasil :

Untuk papan yang kompleks ini (mengandung jumlah maksimal *piece*), algoritma utama (A*, GBFS, dan UCS) tidak berhasil menemukan solusi karena memori yang digunakan terlalu besar. Ini terjadi karena algoritma tersebut menggunakan struktur data *queue* yang disimpan pada setiap iterasinya sehingga menyebabkan algoritma tersebut menggunakan banyak memori. Selain itu juga, cara kami menyimpan solusi, yaitu menggunakan *List of State* yang berisikan struktur data objek-objek lagi (*Board, Piece*), menyebabkan program kami kurang efisien dalam penggunaan memorinya. Cara lebih baik untuk menyimpan solusi adalah dengan menyimpan string langkah-langkahnya saja. Terlepas dari itu, IDDFS dan *Hill Climb* berhasil menemukan solusi, dimana IDDFS menemukan solusi yang optimal, namun dalam waktu yang relatif lama, dan jumlah *node* yang banyak, sedangkan *hill climbing* tidak menemukan solusi optimal, tetapi secara waktu dan *node* dikunjungi sangat efisien.

V. Kesimpulan

Setelah mengimplementasikan dan menguji berbagai algoritma pencarian pada permainan *Rush Hour*, kami menemukan bahwa setiap algoritma memiliki karakteristik dan keunggulan masing-masing. A* secara konsisten memberikan solusi dengan langkah paling optimal, namun membutuhkan eksplorasi

node yang cukup banyak. *Greedy Best First Search* lebih cepat dalam mengeksekusi, namun cenderung terjebak pada jalur yang tampak menjanjikan tapi memutar. Sementara itu, *Hill Climbing* justru menjadi yang paling cepat secara waktu karena hanya melihat langkah terbaik secara lokal, walaupun solusinya tidak selalu optimal.

Di sisi lain, variasi heuristik yang digunakan juga sangat mempengaruhi arah dan efisiensi pencarian. Heuristik *Blocking Cars* lebih akurat dalam menilai kesulitan, tetapi agak berat dari sisi waktu. Heuristik *Manhattan* lebih ringan dan cepat. Gabungan keduanya memberikan hasil yang cukup seimbang. Melalui eksperimen ini, kami belajar bahwa tidak ada satu kombinasi algoritma dan heuristik yang selalu paling unggul, semuanya bergantung pada konteks, kebutuhan, dan karakteristik konfigurasi papan yang dihadapi.

Lampiran

A. Pranala Repository

Link: https://github.com/grwna/Tucil3_13523035_13523117

B. Tabel Checklist

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

Daftar Pustaka

[1] N. U. Maulidevi dan R. Munir, "Penentuan Rute (Route/Path Planning) - Bagian 1: Algoritma A*," Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika ITB, 2025. [Daring]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

[2] N. U. Maulidevi dan R. Munir, "Penentuan Rute (Route/Path Planning) - Bagian 2: Algoritma A*," Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika ITB, 2025. [Daring]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf