

Laporan Tugas Kecil 3 IF2211 Strategi Algoritma: Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Nathaniel Jonathan Rusli – 13523013¹, Kenneth Poenadi – 13523040²

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

¹13523013@std.stei.itb.ac.id, ²13523040@std.stei.itb.ac.id

Daftar Isi

Bab 1 Latar Belakang	2
Bab 2 Deskripsi Tugas	3
Bab 3 Source Code Program	5
3.1 Board.java	5
3.2 Car.java	9
3.3 Move.java	11
3.4 State.java	13
3.5 Algorithm.java	18
3.6 UCS.java	19
3.7 Greedy.java	21
3.8 AStar.java	22
3.9 Fringe.java	24
Bab 4 Penjelasan dan Analisis Kompleksitas Algoritma	27
4.1 Uniform Cost Search (UCS)	27
4.2 Greedy Best First Search (GBFS)	30
4.3 A* Search	34
4.4 Fringe Search (Bonus Algorithm)	38
Bab 5 Implementasi Bonus	42
5.1 Pathfinding Algorithm Tambahan	42
5.2 Heuristik Tambahan	42
5.3 GUI dan Animasi (JavaFX)	43
Bab 6 Testing dan Analisis Hasil Testing	48
6.1 Testing	48
6.2 Analisis Hasil Testing	71
Bab 7 Kesimpulan dan Saran	73
7.1 Kesimpulan	73
7.2 Saran	73
Lampiran	74

Bab 1

Latar Belakang

Dalam era komputasi modern, algoritma pathfinding memegang peranan penting dalam berbagai aplikasi seperti navigasi GPS, game komputer, robotika, dan pemodelan jaringan. Algoritma-algoritma ini berperan dalam mencari jalur terpendek atau optimal antara dua titik dalam suatu ruang permasalahan. Di antara banyak algoritma pathfinding yang tersedia, Uniform Cost Search (UCS), Greedy Best First Search, dan A* merupakan algoritma yang banyak diimplementasikan karena keunggulannya dalam berbagai kasus.

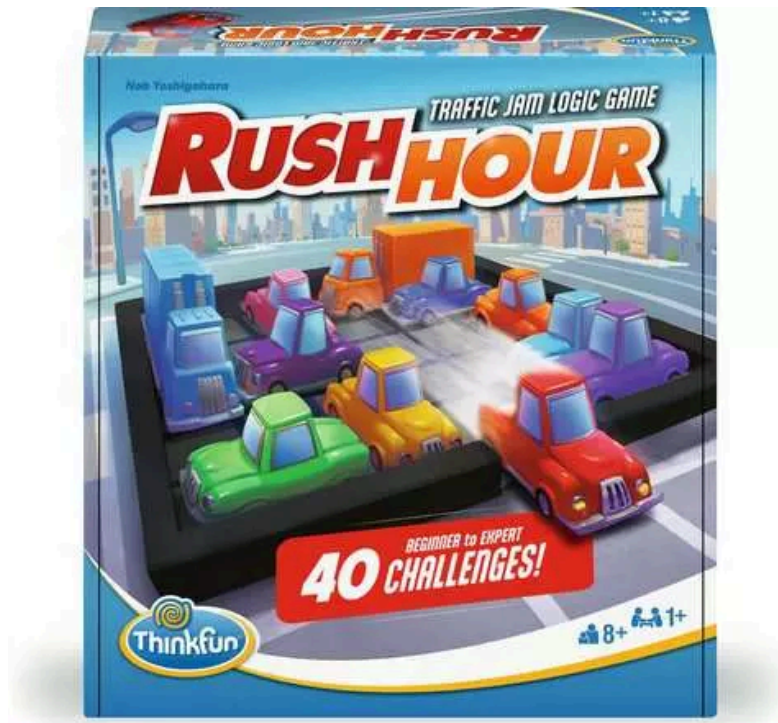
Rush Hour adalah permainan puzzle logika yang menantang pemain untuk menggeser kendaraan dalam grid 6×6 agar mobil utama dapat mencapai pintu keluar. Setiap kendaraan hanya dapat bergerak maju atau mundur sesuai orientasinya (horizontal atau vertikal) dan tidak dapat berputar. Tantangan dalam permainan ini terletak pada kompleksitas ruang pencarian yang dihadapi pemain untuk menentukan urutan pergerakan yang tepat dengan langkah minimum.

Dari segi komputer, Rush Hour dipandang sebagai masalah pencarian jalur dalam ruang keadaan (*state space*), di mana setiap keadaan merepresentasikan konfigurasi papan pada satu waktu tertentu. Transisi antar keadaan terjadi ketika satu kendaraan digeser, menghasilkan konfigurasi papan yang baru. Tujuan akhir pencarian adalah menemukan rangkaian pergeseran kendaraan yang memungkinkan mobil utama mencapai pintu keluar.

Penerapan algoritma pathfinding seperti UCS, Greedy Best First Search, dan A* pada Rush Hour menjadi menarik karena memungkinkan kita menganalisis efektivitas dan efisiensi masing-masing algoritma dalam konteks permasalahan yang berbeda dari pencarian jalur tradisional. UCS menjamin solusi optimal dengan memprioritaskan jalur dengan biaya terendah, Greedy Best First Search menggunakan heuristik untuk bergerak cepat menuju tujuan, sementara A* menggabungkan kedua pendekatan tersebut untuk mencapai keseimbangan antara optimalisasi dan efisiensi namun tetap mendapatkan hasil yang optimal.

Bab 2

Deskripsi Tugas



Gambar 1. Rush Hour Puzzle

(Sumber:

<https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

Rush Hour adalah sebuah permainan puzzle logika berbasis grid yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6×6) 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.

Bab 3

Source Code Program

3.1 Board.java

Representasi *Class* papan permainan Rush Hour pada program:

```
package rushhour.lib;
public class Board {
    private int width;
    private int height;
    private char[][] grid;
    private Car[] cars;
    private int doorX;
    private int doorY;

    public Board(int width, int height) {
        this.width = width;
        this.height = height;
        this.grid = new char[height + 2][width + 2];

        for (int i = 0; i < height + 2; i++) {
            for (int j = 0; j < width + 2; j++) {
                if (i == 0 || i == height + 1 || j == 0 || j == width + 1) {
                    grid[i][j] = 'X';
                } else {
                    grid[i][j] = '.';
                }
            }
        }
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    public char[][] getGrid() {
        return grid;
    }

    public Car[] getCars() {
        return cars;
    }

    public int getDoorX() {
        return doorX;
    }
}
```

```

    }

    public int getDoorY() {
        return doorY;
    }

    public boolean isEmpty(int x, int y) {
        int gridX = x + 1;
        int gridY = y + 1;

        if (x < 0 || x >= width || y < 0 || y >= height) {
            return false;
        }
        return grid[gridY][gridX] == '.';
    }

    public boolean isDoorAt(int x, int y) {
        return x == doorX && y == doorY;
    }

    public void updateGrid() {
        for (int i = 0; i < height + 2; i++) {
            for (int j = 0; j < width + 2; j++) {
                if (i == 0 || i == height + 1 || j == 0 || j == width + 1) {
                    grid[i][j] = 'X';
                } else {
                    grid[i][j] = '.';
                }
            }
        }

        if (doorX >= 0 && doorX < width && doorY >= 0 && doorY < height) {
            grid[doorY + 1][doorX + 1] = 'K';
        } else if (doorX == width && doorY >= 0 && doorY < height) {
            grid[doorY + 1][doorX + 1] = 'K';
        } else if (doorX >= 0 && doorX < width && doorY == height) {
            grid[doorY + 1][doorX + 1] = 'K';
        }

        if (cars != null) {
            for (Car car : cars) {
                int[][] cells = car.posisiCar();
                for (int[] cell : cells) {
                    int x = cell[0];
                    int y = cell[1];
                    if (x >= 0 && x < width && y >= 0 && y < height) {
                        grid[y + 1][x + 1] = car.getId();
                    }
                }
            }
        }
    }

    public Car getCarsAt(int x, int y) {

```

```

        for (Car car : cars) {
            int[][] positions = car.posisiCar();
            for (int[] pos : positions) {
                if (pos[0] == x && pos[1] == y) {
                    return car;
                }
            }
        }
        return null;
    }

    public void setCars(Car[] cars) {
        this.cars = cars;
        updateGrid();
    }

    public void setDoor(int x, int y) {
        this.doorX = x;
        this.doorY = y;
        updateGrid();
    }

    public boolean isSolved() {
        for (Car car : cars) {
            if (car.isPrimary()) {
                if (car.isHorizontal()) {
                    //right door (doorX == width)
                    if (doorX == width && car.getX() + car.getLength() == width &&
car.getY() == doorY) {
                        return true;
                    }
                    //left door (doorX == -1)
                    else if (doorX == -1 && car.getX() == 0 && car.getY() == doorY) {
                        return true;
                    }
                    else if (doorX >= 0 && doorX < width &&
car.getX() + car.getLength() == doorX && car.getY() ==
doorY) {
                        return true;
                    }
                } else { //vertical car
                    if (doorY == height && car.getY() + car.getLength() == height &&
car.getX() == doorX) {
                        return true;
                    }
                    else if (doorY == -1 && car.getY() == 0 && car.getX() == doorX) {
                        return true;
                    }
                    else if (doorY >= 0 && doorY < height &&
car.getY() + car.getLength() == doorY && car.getX() ==
doorX) {
                        return true;
                    }
                }
            }
        }
    }

```

```

        }
        return false;
    }
}
return false;
}

public void printBoard() {
    for (int i = 0; i < height + 2; i++) {
        for (int j = 0; j < width + 2; j++) {
            System.out.print(grid[i][j]);
        }
        System.out.println();
    }
}

public void printPlayableArea() {
    for (int i = 1; i <= height; i++) {
        for (int j = 1; j <= width; j++) {
            System.out.print(grid[i][j]);
        }
        System.out.println();
    }
}

public Car getCarById(char id) {
    for (Car car : cars) {
        if (car.getId() == id) {
            return car;
        }
    }
    return null;
}

public Board copy() {
    Board newBoard = new Board(width, height);
    newBoard.setDoor(doorX, doorY);

    if (cars != null) {
        Car[] newCars = new Car[cars.length];
        for (int i = 0; i < cars.length; i++) {
            newCars[i] = cars[i].copy();
        }
        newBoard.setCars(newCars);
    }

    return newBoard;
}
}

```


3.2 Car.java

Representasi *Class* kendaraan pada permainan Rush Hour pada program:

```
package rushhour.lib;
public class Car {
    private char id;
    private int x;
    private int y;
    private int length;
    private boolean isHorizontal;
    private boolean isPrimary; // mobil yg dikeluarkan bukan

    public Car() {
        this.id = 'K';
        this.x = 0;
        this.y = 0;
        this.length = 0;
        this.isHorizontal = false;
        this.isPrimary = false;
    }

    public Car(char id, int x, int y, int length, boolean isHorizontal, boolean
isPrimary) {
        this.id = id;
        this.x = x;
        this.y = y;
        this.length = length;
        this.isHorizontal = isHorizontal;
        this.isPrimary = isPrimary;
    }

    public char getId() {
        return id;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }
}
```

```

public int getLength() {
    return length;
}

public boolean isHorizontal() {
    return isHorizontal;
}

public boolean isPrimary() {
    return isPrimary;
}

public boolean canMoveUp(Board board) {
    if (isHorizontal) return false; //mobil horizontal ga bisa naik
    return y > 0 && board.isCellEmpty(x, y - 1);
}

public boolean canMoveDown(Board board) {
    if (isHorizontal) return false; //mobil horizontal ga bisa turun
    return y + length < board.getHeight() && board.isCellEmpty(x, y + length);
}

public boolean canMoveLeft(Board board) {
    if (!isHorizontal) return false; //mobil vertical ga bisa ke kiri
    return x > 0 && board.isCellEmpty(x - 1, y);
}

public boolean canMoveRight(Board board) {
    if (!isHorizontal) return false; //mobil vertical ga bisa ke kanan
    int rightEdge = x + length;
    if (isPrimary && board.isDoorAt(rightEdge, y)) {
        return true;
    }
    return rightEdge < board.getWidth() && board.isCellEmpty(rightEdge, y);
}

public void moveUp() {
    y--;
}

public void moveDown() {
    y++;
}

public void moveLeft() {
    x--;
}

public void moveRight() {
    x++;
}

public int[][] posisiCar() {
    int[][] cells = new int[length][2];
}

```

```

        for (int i = 0; i < length; i++) {
            if (isHorizontal) {
                cells[i][0] = x + i;
                cells[i][1] = y;
            } else {
                cells[i][0] = x;
                cells[i][1] = y + i;
            }
        }
        return cells;
    }

    public Car copy() {
        return new Car(id, x, y, length, isHorizontal, isPrimary);
    }
}

```

3.3 Move.java

Representasi pergerakan mobil, mewakili ID mobil dan arahnya:

```

package rushhour.lib;
import java.util.ArrayList;
import java.util.List;

public class Move {
    private char carId;
    private String direction;

    public Move(char carId, String direction) {
        this.carId = carId;
        this.direction = direction;
    }

    public char getCarId() {
        return carId;
    }

    public String getDirection() {
        return direction;
    }

    @Override
    public String toString() {
        return carId + "-" + direction;
    }

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

```

```

        Move other = (Move) obj;
        return carId == other.carId && direction.equals(other.direction);
    }

    public void applyMove(Board board, Move move) {
        Car car = board.getCarById(move.carId);

        if (move.direction.equals("up")) {
            car.moveUp();
        } else if (move.direction.equals("down")) {
            car.moveDown();
        } else if (move.direction.equals("left")) {
            car.moveLeft();
        } else if (move.direction.equals("right")) {
            car.moveRight();
        }

        board.updateGrid();
    }

    public static List<Move> getPossibleMoves(Board board) {
        List<Move> possibleMoves = new ArrayList<>();

        for (Car car : board.getCars()) {
            if (car.canMoveUp(board)) {
                possibleMoves.add(new Move(car.getId(), "up"));
            }
            if (car.canMoveDown(board)) {
                possibleMoves.add(new Move(car.getId(), "down"));
            }
            if (car.canMoveLeft(board)) {
                possibleMoves.add(new Move(car.getId(), "left"));
            }
            if (car.canMoveRight(board)) {
                possibleMoves.add(new Move(car.getId(), "right"));
            }
        }

        return possibleMoves;
    }
}

```

3.4 State.java

Representasi papan permainan Rush Hour pada program:

```
package rushhour.lib;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Objects;
import java.util.Set;

public class State implements Comparable<State> { //kelas buat nampung state board
    skrg
    public Board board;
    public List<Move> moves; // History dari moves yang sudah dilakukan
    public int gValue; // Cost so far: untuk UCS dan A*
    public int hValue; // Heuristic value: untuk A* dan Greedy
    public int fValue; // Untuk A*:  $f(n) = g(n) + h(n)$ 

    // Constructor untuk UCS (tidak butuh hValue)
    public State(Board board, List<Move> moves) {
        this.board = board;
        this.moves = new ArrayList<>(moves);
        calculateGValue();
    }

    /* Constructor untuk Greedy dan A*
     * Perhitungan hValue dijadikan parameter karena terdapat
     * beberapa metode perhitungan hValue */
    public State(Board board, List<Move> moves, int hValue) {
        this.board = board;
        this.moves = new ArrayList<>(moves);
        calculateGValue();
        this.hValue = hValue;
        calculateFValue();
    }

    // Getter
    public Board getBoard() {
        return board;
    }

    public List<Move> getMoves() {
        return moves;
    }

    public int getGValue() {
        return gValue;
    }

    public int getHValue() {
```

```

        return hValue;
    }

    public int getFValue() {
        return fValue;
    }

    /*Convert state jadi String untuk visitedStates
    * untuk menghindari duplicates dan cycles */
    public static String getBoardStateString(Board board) {
        StringBuilder sb = new StringBuilder();
        for (Car car : board.getCars()) {
            sb.append(car.getId()).append(car.getX()).append(car.getY());
        }
        return sb.toString();
    }

    // Calculation functions
    public void calculateGValue() {
        this.gValue = this.moves.size();
    }

    // f(n) = g(n) + h(n)
    public void calculateFValue() {
        this.fValue = getGValue() + getHValue();
    }

    public static int calculateHValue(String heuristic, Board board) {
        if (heuristic.equals("blockingCars")) {
            return State.calculateBlockingCarsHeuristic(board);
        } else if (heuristic.equals("exitDistance")) {
            return State.calculateExitDistanceHeuristic(board);
        } else {
            return State.calculateBlockingCarsAndExitDistanceHeuristic(board);
        }
    }

    public static int calculateBlockingCarsHeuristic(Board board) {
        Car primaryCar = null;

        // Cari mobil utama (primary car)
        for (Car car : board.getCars()) {
            if (car.isPrimary()) {
                primaryCar = car;
                break;
            }
        }

        if (primaryCar == null) return Integer.MAX_VALUE;

        int doorX = board.getDoorX();
        int doorY = board.getDoorY();
    }

```

```

        // Horizontal case
        if (primaryCar.isHorizontal()) {
            Set<Car> blockingCars = new HashSet<>();

            // Periksa semua sel antara ujung mobil utama sampai ke pintu keluar
            for (int x = primaryCar.getX() + primaryCar.getLength(); x < doorX; x++)
            {
                if (!board.isCellEmpty(x, primaryCar.getY())) {
                    Car carAtPosition = board.getCarsAt(x, primaryCar.getY());
                    if (carAtPosition != null && carAtPosition != primaryCar) {
                        blockingCars.add(carAtPosition);
                    }
                }
            }

            return blockingCars.size();
        }

        // Vertical case
        else {
            Set<Car> blockingCars = new HashSet<>();

            if (doorY > primaryCar.getY()) {
                for (int y = primaryCar.getY() + primaryCar.getLength(); y < doorY;
y++) {
                    if (!board.isCellEmpty(primaryCar.getX(), y)) {
                        Car carAtPosition = board.getCarsAt(primaryCar.getX(), y);
                        if (carAtPosition != null && carAtPosition != primaryCar) {
                            blockingCars.add(carAtPosition);
                        }
                    }
                }
            } else {
                for (int y = 0; y < primaryCar.getY(); y++) {
                    if (!board.isCellEmpty(primaryCar.getX(), y)) {
                        Car carAtPosition = board.getCarsAt(primaryCar.getX(), y);
                        if (carAtPosition != null && carAtPosition != primaryCar) {
                            blockingCars.add(carAtPosition);
                        }
                    }
                }
            }

            return blockingCars.size();
        }
    }

    public static int calculateExitDistanceHeuristic(Board board) {
        Car primaryCar = null;
        for (Car car : board.getCars()) {
            if (car.isPrimary()) {
                primaryCar = car;
            }
        }
    }

```

```

        break;
    }
}

if (primaryCar == null) return 9999999;

int doorX = board.getDoorX();
int doorY = board.getDoorY();

int distance = 0;

if (primaryCar.isHorizontal()) {
    distance = doorX - (primaryCar.getX() + primaryCar.getLength());
    return distance;
} else {
    distance = doorY - (primaryCar.getY() + primaryCar.getLength());
    if (doorY == 0) {
        distance = primaryCar.getY();
    }
    return distance;
}
}

public static int calculateBlockingCarsAndExitDistanceHeuristic(Board board) {
    Car primaryCar = null;
    for (Car car : board.getCars()) {
        if (car.isPrimary()) {
            primaryCar = car;
            break;
        }
    }

    if (primaryCar == null) return 9999999;

    int doorX = board.getDoorX();
    int doorY = board.getDoorY();

    int distance = 0;

    if (primaryCar.isHorizontal()) {
        distance = doorX - (primaryCar.getX() + primaryCar.getLength());

        Set<Car> blockingCars = new HashSet<>();
        for (int x = primaryCar.getX() + primaryCar.getLength(); x < doorX; x++)
        {
            if (!board.isCellEmpty(x, primaryCar.getY())) {
                Car carAtPosition = board.getCarsAt(x, primaryCar.getY());
                if (carAtPosition != null) {
                    blockingCars.add(carAtPosition);
                }
            }
        }
    }
}

```



```

        return distance + blockingCars.size();
    } else {
        distance = doorY - (primaryCar.getY() + primaryCar.getLength());
        if (doorY == 0) {
            distance = primaryCar.getY();
        }

        int blockingCars = 0;
        if (doorY > primaryCar.getY()) {
            for (int y = primaryCar.getY() + primaryCar.getLength(); y <
doorY; y++) {
                if (!board.isCellEmpty(primaryCar.getX(), y)) {
                    blockingCars++;
                }
            }
        } else {
            for (int y = 0; y < primaryCar.getY(); y++) {
                if (!board.isCellEmpty(primaryCar.getX(), y)) {
                    blockingCars++;
                }
            }
        }

        return distance + blockingCars;
    }
}

@Override
public int compareTo(State other) {
    return Integer.compare(this.fValue, other.fValue);
}

@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }
    State state = (State) o;
    return Objects.equals(board, state.board);
}

@Override
public int hashCode() {
    return Objects.hash(board);
}

public State addMove(Move move, Board newBoard, int newHValue) {
    List<Move> newMoves = new ArrayList<>(moves);
    newMoves.add(move);
    return new State(newBoard, newMoves, newHValue);
}

```

```
}  
}
```

3.5 Algorithm.java

Abstract class yang menjadi fondasi *Class* seluruh algoritma (UCS, GBFS, A*, dan Fringe Search):

```
package rushhour.lib;  
import java.util.*;  
  
/* Abstract Class Algorithm  
 * untuk implementasi seluruh algoritma lainnya:  
 * UCS, Greedy Best First, A*, dan Fringe*/  
public abstract class Algorithm {  
    protected Board initialBoard;  
    protected Set<String> visitedStates;  
    protected int nodesExplored;  
    protected long timeElapsed;  
  
    // Constructor  
    public Algorithm(Board board) {  
        this.initialBoard = board;  
        this.visitedStates = new HashSet<>();  
        this.nodesExplored = 0;  
        this.timeElapsed = 0;  
    }  
  
    // Method solve default untuk uninformed algorithm  
    public List<Move> solve() {  
        return solve(null);  
    }  
  
    // Method solve untuk informed algorithm  
    public abstract List<Move> solve(String heuristic);  
  
    // Digunakan seluruh algoritma  
    public void displaySolution(List<Move> solution) {  
        if (solution == null) {  
            System.out.println("No solution found.");  
            return;  
        }  
  
        Board board = initialBoard.copy();  
        System.out.println("Papan Awal");  
        board.printPlayableArea();  
  
        for (int i = 0; i < solution.size(); i++) {  
            Move move = solution.get(i);  
            move.applyMove(board, move);  
        }  
    }  
}
```

```

        System.out.println("Gerakan " + (i + 1) + ": " + move);
        board.printPlayableArea();
    }
}

// Getter: nodes explored
public int getNodesExplored() {
    return nodesExplored;
}

// Getter: time elapsed
public long getTimeElapsed() {
    return timeElapsed;
}
}

```

3.6 UCS.java

Class implementasi algoritma Uniform Cost Search (UCS) pada program:

```

package rushhour.lib;
import java.util.*;

// Lihat Class Algorithm untuk melihat class member
public class UCS extends Algorithm {
    /* Cukup memasukkan initialBoard state yang ingin di-solve
     * Gunakan solve() untuk memberikan solusi berdasarkan algo UCS */
    public UCS(Board board) {
        super(board);
    }

    @Override
    public List<Move> solve(String heuristic) {
        // Inisialisasi data structure
        PriorityQueue<State> queue = new PriorityQueue<>(Comparator.comparingInt(s
-> s.getGValue()));
        List<Move> initialMoves = new ArrayList<>();
        queue.add(new State(initialBoard.copy(), initialMoves));

        // Waktu mulai
        long start = System.currentTimeMillis();

        // Proses UCS
        while (!queue.isEmpty()) {
            State stateNow = queue.poll();
            Board currentBoard = stateNow.getBoard();
            List<Move> currentMoves = stateNow.getMoves();

            nodesExplored++;
            // Goal state

```

```

        if (currentBoard.isSolved()) {
            // Waktu selesai
            long end = System.currentTimeMillis();
            timeElapsed = end - start;

            return currentMoves;
        }

        /* Validasi duplikat
         * Untuk menghindari cycle atau repeated states */
        String stateString = State.getBoardStateString(currentBoard);
        if (visitedStates.contains(stateString)) {
            continue;
        } else {
            visitedStates.add(stateString);
        }

        // Iterasi seluruh kemungkinan move pada state terkini
        for (Move move : Move.getPossibleMoves(currentBoard)) {
            Board nextBoard = currentBoard.copy();
            move.applyMove(nextBoard, move);

            String nextStateString = State.getBoardStateString(nextBoard);
            if (!visitedStates.contains(nextStateString)) {
                List<Move> nextMoves = new ArrayList<>(currentMoves);
                nextMoves.add(move);

                queue.add(new State(nextBoard, nextMoves));
            }
        }
    }
    // Waktu selesai
    long end = System.currentTimeMillis();
    timeElapsed = end - start;

    return null;
}
}

```

3.7 Greedy.java

Class implementasi algoritma Greedy Best First Search pada program:

```
package rushhour.lib;
import java.util.*;

// Lihat Class Algorithm untuk melihat class member
public class Greedy extends Algorithm {
    /* Cukup memasukkan initialBoard state yang ingin di-solve
     * Gunakan solve() untuk memberikan solusi berdasarkan algo GBFS */
    public Greedy(Board board) {
        super(board);
    }

    /* String heuristic: metode heuristic yang dipilih */
    @Override
    public List<Move> solve(String heuristic) {
        if (!heuristic.equals("blockingCars") &&
            !heuristic.equals("exitDistance") &&
            !heuristic.equals("blockingCarsAndExitDistance")) {
            System.out.println("Invalid heuristic.");
            return new ArrayList<Move>(); // Empty list
        }

        // Waktu mulai
        long start = System.currentTimeMillis();

        // Inisialisasi data structure
        PriorityQueue<State> queue = new PriorityQueue<>(Comparator.comparingInt(s
-> s.getHValue()));
        List<Move> initialMoves = new ArrayList<>();
        int initialHValue = State.calculateHValue(heuristic, initialBoard);
        queue.add(new State(initialBoard.copy(), initialMoves, initialHValue));

        // Proses GBFS search
        while (!queue.isEmpty()) {
            State statenow = queue.poll();
            Board currentBoard = statenow.getBoard();
            List<Move> currentMoves = statenow.getMoves();

            nodesExplored++;
            // Goal state
            if (currentBoard.isSolved()) {
                // Waktu selesai
                long end = System.currentTimeMillis();
                timeElapsed = end - start;

                return currentMoves;
            }
        }

        /* Validasi duplikat
```

```

        * Untuk menghindari cycle atau repeated states */
String stateString = State.getBoardStateString(currentBoard);
if (visitedStates.contains(stateString)) {
    continue;
} else {
    visitedStates.add(stateString);
}

// Iterasi seluruh kemungkinan move pada state terkini
for (Move move : Move.getPossibleMoves(currentBoard)) {
    Board nextBoard = currentBoard.copy();
    move.applyMove(nextBoard, move);

    String nextStateString = State.getBoardStateString(nextBoard);
    if (!visitedStates.contains(nextStateString)) {
        List<Move> nextMoves = new ArrayList<>(currentMoves);
        nextMoves.add(move);
        int hValue = State.calculateHValue(heuristic, nextBoard);

        queue.add(new State(nextBoard, nextMoves, hValue));
    }
}

// Waktu selesai
long end = System.currentTimeMillis();
timeElapsed = end - start;

return null;
}
}

```

3.8 AStar.java

Class implementasi algoritma A* Search pada program:

```

package rushhour.lib;
import java.util.*;

// Lihat Class Algorithm untuk melihat class member
public class AStar extends Algorithm {
    /* Cukup memasukkan initialBoard state yang ingin di-solve
    * Gunakan solve() untuk memberikan solusi berdasarkan algo A* */
    public AStar(Board board) {
        super(board);
    }

    /* String heuristic: metode heuristic yang dipilih */
    @Override
    public List<Move> solve(String heuristic) {
        // Validasi input heuristic
        if (!heuristic.equals("blockingCars") &&

```

```

!heuristic.equals("exitDistance") &&
!heuristic.equals("blockingCarsAndExitDistance")) {
    System.out.println("Invalid heuristic.");
    return new ArrayList<Move>(); // Empty list
}

// Waktu mulai
long start = System.currentTimeMillis();

// Inisialisasi data structure
PriorityQueue<State> queue = new PriorityQueue<>(Comparator.comparingInt(s
-> s.getFValue()));
List<Move> initialMoves = new ArrayList<>();
int initialHValue = State.calculateHValue(heuristic, initialBoard);
queue.add(new State(initialBoard.copy(), initialMoves, initialHValue));

// Proses A* search
while (!queue.isEmpty()) {
    State statenow = queue.poll();
    Board currentBoard = statenow.getBoard();
    List<Move> currentMoves = statenow.getMoves();

    nodesExplored++;
    // Goal state
    if (currentBoard.isSolved()) {
        // Waktu selesai
        long end = System.currentTimeMillis();
        timeElapsed = end - start;

        return currentMoves;
    }

    /* Validasi duplikat
     * Untuk menghindari cycle atau repeated states */
    String stateString = State.getBoardStateString(currentBoard);
    if (visitedStates.contains(stateString)) {
        continue;
    } else {
        visitedStates.add(stateString);
    }

    // Iterasi seluruh kemungkinan move pada state terkini
    for (Move move : Move.getPossibleMoves(currentBoard)) {
        Board nextBoard = currentBoard.copy();
        move.applyMove(nextBoard, move);

        String nextStateString = State.getBoardStateString(nextBoard);
        if (!visitedStates.contains(nextStateString)) {
            List<Move> nextMoves = new ArrayList<>(currentMoves);
            nextMoves.add(move);
            int hValue = State.calculateHValue(heuristic, nextBoard);

            queue.add(new State(nextBoard, nextMoves, hValue));
        }
    }
}

```

```

        }
    }
}
// Waktu selesai
long end = System.currentTimeMillis();
timeElapsed = end - start;

return null;
}
}

```

3.9 Fringe.java

Class implementasi algoritma Fringe Search pada program:

```

package rushhour.lib;
import java.util.*;

// Lihat Class Algorithm untuk melihat class member
public class Fringe extends Algorithm{
    /* Cukup memasukkan initialBoard state yang ingin di-solve
     * Gunakan solve() untuk memberikan solusi berdasarkan algo Fringe */
    public Fringe(Board board) {
        super(board);
    }

    /* String heuristic: metode heuristic yang dipilih */
    @Override
    public List<Move> solve(String heuristic) {
        // Pengecekan metode heuristic yang digunakan
        if (!heuristic.equals("blockingCars") &&
            !heuristic.equals("exitDistance") &&
            !heuristic.equals("blockingCarsAndExitDistance")) {
            System.out.println("Invalid heuristic.");
            return new ArrayList<Move>(); // Empty list
        }

        // Waktu mulai
        long start = System.currentTimeMillis();

        /* Inisialisasi data structure
         * Salah satu karakteristik utama dari Fringe search
         * yaitu terdapat list now dan later*/
        Deque<State> now = new LinkedList<State>();
        Deque<State> later = new LinkedList<State>();

        /* Nilai initial threshold adalah nilai f(n) dari state awal
         * dengan g(n) = 0 dan h(n) bergantung pada jenis heuristic yang dipilih*/
        int initialHValue = State.calculateHValue(heuristic, initialBoard);
        State initialState = new State(initialBoard, new ArrayList<>(),
            initialHValue);
    }
}

```



```

int threshold = initialState.getFValue();
int increment = 5;

// Inisialisasi list now
now.add(initialState);

// Proses Fringe search
while (!now.isEmpty()) {
    while (!now.isEmpty()) {
        State currentState = now.poll();
        Board currentBoard = currentState.getBoard();
        List<Move> currentMoves = currentState.getMoves();

        nodesExplored++;
        // Goal state
        if (currentBoard.isSolved()) {
            // Waktu selesai
            long end = System.currentTimeMillis();
            timeElapsed = end - start;

            return currentMoves;
        }

        /* Validasi duplikat
         * Untuk menghindari cycle atau repeated states */
        String stateString = State.getBoardStateString(currentBoard);
        if (visitedStates.contains(stateString)) {
            continue;
        } else {
            visitedStates.add(stateString);
        }

        // Iterasi seluruh kemungkinan move pada state terkini
        for (Move move : Move.getPossibleMoves(currentBoard)) {
            Board nextBoard = currentBoard.copy();
            move.applyMove(nextBoard, move);

            String nextStateString = State.getBoardStateString(nextBoard);
            if (!visitedStates.contains(nextStateString)) {
                List<Move> nextMoves = new ArrayList<>(currentMoves);
                nextMoves.add(move);
                int hValue = State.calculateHValue(heuristic, nextBoard);

                State newState = new State(nextBoard, nextMoves, hValue);
                if (newState.getFValue() <= threshold) {
                    now.add(newState);
                } else {
                    later.add(newState);
                }
            }
        }
    }
}

```

```

        // Tidak ditemukan solusi
        if (later.isEmpty()) break;
        // Increment nilai threshold baru
        threshold += increment;
        // Update now dan later
        now.addAll(later); // now = later
        later.clear();     // later = []
    }

    // Waktu selesai
    long end = System.currentTimeMillis();
    timeElapsed = end - start;
    return new ArrayList<>();
}
}

```

Bab 4

Penjelasan dan Analisis Kompleksitas Algoritma

4.1 Uniform Cost Search (UCS)

Uniform Cost Search adalah algoritma pencarian berbobot yang selalu memilih node dengan **biaya kumulatif** terkecil dari awal hingga saat itu. Tidak ada heuristik yang digunakan semua keputusan tergantung kepada biaya perjalanan sejauh ini, $g(n)$.

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

$g(n)$ = total biaya (cost) untuk mencapai node n dari start.

UCS menyimpan frontier dalam priority queue yang diurutkan naik berdasarkan $g(n)$.

```
Pseudocode UCS:
function UCS(start, goal):
    frontier ← priority queue ordered by  $g(n)$ 
    frontier.insert(start,  $g(start)=0$ )
    explored ← empty set

    while frontier is not empty:
        node ← frontier.pop()           // node dengan  $g(n)$  minimal
        if node.state == goal:
            return reconstruct_path(node)

        explored.add(node.state)

        for each successor in expand(node):
            newCost ← node.g + cost(node, successor)
            if successor.state ∉ explored AND
              (successor ∉ frontier OR newCost < successor.g):
                successor.parent ← node
                successor.g ← newCost
                frontier.insert_or_update(successor, newCost)

    return failure
```

Uniform Cost Search (UCS) memulai dari node awal dengan biaya kumulatif $g(start) = 0$

dan menyimpan semua node frontier dalam priority queue yang diurutkan berdasarkan $g(n)$, yaitu total biaya untuk mencapai node tersebut. Pada tiap

iterasi, UCS mengeluarkan node dengan nilai $g(n)$ terkecil, menandainya sebagai *explored*, lalu menghasilkan semua successor-nya, menghitung biaya baru $g_{new} = g(current) + cost(current, succ)$, dan menambahkan atau memperbarui successor di frontier jika belum dieksplorasi atau ditemukan jalur lebih murah. Dengan selalu memilih jalur dengan biaya terendah terlebih dahulu, UCS menjamin bahwa goal yang di-pop dari queue, tidak ada jalur lain dengan biaya lebih kecil yang tertinggal itulah sebabnya hasilnya **admissible** (optimal) **asalkan semua biaya edge non-negatif**.

Di Rush Hour, biasanya mengasumsikan setiap gerakan mobil punya biaya yang sama (misalnya 1 per langkah), maka:

1. **Uniform Cost Search (UCS)** akan selalu memilih node dengan biaya kumulatif terkecil, yaitu jumlah langkah paling sedikit dari root.
2. **Breadth-First Search (BFS)** memilih node berdasarkan kedalaman (*depth*) dalam tree, yang sama dengan jumlah langkah dari root jika setiap edge bernilai 1.

Dengan pendekatan lapisan per lapisan (layer by layer), UCS dan BFS mengekskansi semua konfigurasi dengan 0 langkah, lalu 1 langkah, lalu 2 langkah, dan seterusnya. Hasil *path* optimal (jalur dengan jumlah langkah minimal) yang ditemukan pun identik. Hal ini juga disebabkan oleh kesamaan struktur data antara UCS dan BFS yang menggunakan queue untuk iterasi setiap node yang ada.

Pada fungsi `solve()` yang berisi logika utama dari Uniform Cost Search (UCS), digunakan Priority Queue sebagai struktur data utama. UCS memprioritaskan eksplorasi state berdasarkan biaya kumulatif $g(n)$ dari titik awal menuju state saat ini, tanpa mempertimbangkan estimasi heuristik ke goal seperti pada A*.

Seluruh operasi `poll()` dan `add()` pada Priority Queue membutuhkan waktu $O(\log n)$ karena berbasis struktur data heap.

```
while (!queue.isEmpty()) {  
    // Pemrosesan utama UCS  
}
```

Di setiap iterasi, UCS mengambil node dengan biaya terendah $g(n)$ dari queue, dan menghasilkan semua gerakan sah dari state tersebut:

```

for (Move move : Move.getPossibleMoves(currentBoard)) {
    // Pemrosesan setiap gerakan sah
    ...
}

```

Bila kita asumsikan:

b = branching factor (jumlah maksimum gerakan sah per state),

d = depth solusi (panjang jalur terpendek dari start ke goal),

mn = ukuran grid papan permainan (m baris \times n kolom).

Dengan demikian, UCS menjelajahi semua node dengan biaya $g(n) <$ biaya solusi optimal, dan mungkin beberapa dengan $g(n) =$ biaya solusi. Dalam kasus terburuk, UCS bisa menjelajahi semua node hingga kedalaman d , yaitu:

$$O(b^d) \text{ node}$$

(sama dengan BFS karena UCS = BFS dengan bobot)

Untuk setiap node yang dieksplorasi, dilakukan:

```

copy() papan  $\rightarrow O(mn)$  (karena representasi grid)
Evaluasi biaya  $g(n) \rightarrow O(1)$ 
Penambahan ke Priority Queue  $\rightarrow O(\log N)$ , dengan  $N$  = jumlah elemen saat ini
dalam queue, maksimum  $O(b^d)$ 

```

Maka, total waktu untuk seluruh node adalah:

$$\text{Jumlah node} \times \text{biaya per node} \approx O(b^d) \times O(\log b^d) = O(b^d \log b^d) = O(b^d \cdot d \log b)$$

Namun karena faktor logaritmiknya lebih kecil dibandingkan eksponensial b^d , maka untuk notasi asimtotik disederhanakan menjadi:

$$O(b^d)$$

UCS menyimpan:

1. Priority Queue untuk state yang sedang diproses \rightarrow hingga $O(b^d)$
2. Visited Set untuk mendeteksi duplikasi dan mencegah siklus \rightarrow hingga $O(b^d)$

sehingga Space complexity UCS adalah:

$$O(b^d)$$

4.2 Greedy Best First Search (GBFS)

Seperti yang kita ketahui, Greedy Best-First Search adalah algoritma pencarian heuristik yang bekerja dengan prinsip serakah dalam memilih node yang akan dieksplorasi berikutnya. Algoritma ini menggunakan fungsi evaluasi $f(n)$ yang didasarkan sepenuhnya pada fungsi heuristik $h(n)$, di mana:

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

Fungsi heuristik $h(n)$ merupakan estimasi biaya dari node saat ini ke node tujuan (goal). Estimasi ini bisa berupa perhitungan jarak, jumlah langkah, atau metrik lain yang relevan dengan permasalahan yang dihadapi. Dalam kasus ini, kami menggunakan 2 heuristik yang berbeda,

1. Distance to Door

Heuristik Exit Distance atau yang biasa dikenal dengan istilah Manhattan Distance mengukur seberapa jauh mobil utama (primary car) dari pintu keluar. Implementasi heuristik ini mengukur jarak Manhattan dari posisi saat ini mobil utama ke pintu keluar. Semakin kecil nilai heuristik ini, semakin dekat mobil utama dengan pintu keluar, dan semakin menjanjikan konfigurasi tersebut.

```
if (primaryCar.isHorizontal()) {  
    distance = doorX - (primaryCar.getX() + primaryCar.getLength());  
    return distance;  
} else {  
    distance = doorY - (primaryCar.getY() + primaryCar.getLength());  
    if (doorY == 0) {  
        distance = primaryCar.getY();  
    }  
    return distance;  
}
```

Estimasi ini hanya memperhitungkan jarak minimum yang harus ditempuh mobil utama jika tidak ada penghalang sama sekali. Karena biaya nyata $h^*(n) \leq h(n) \leq h^*(n)$ selalu \geq jarak lurus ke kanan atau ke bawah, heuristik ini:

- **Optimistik** (optimistic): Tidak pernah melebihi-lebihkan biaya sebenarnya.
- **Admissible**: Menjamin tidak memandu pencarian melewati solusi optimal.
- **Sederhana namun lemah**: Kadang kurang efektif jika ada banyak mobil penghalang.

2. Blocking Cars

Heuristik *Blocking Cars* menghitung jumlah kendaraan yang menghalangi jalur langsung antara mobil utama dan pintu keluar + jarak Manhattan ke posisi pintu. Semakin sedikit kendaraan yang menghalangi, semakin kecil nilai heuristik, dan semakin menjanjikan konfigurasi tersebut untuk dieksplorasi. Nilai kendaraan yang menghalangi akan membuat nilai $h(n)$ nya semakin besar

```
return blockingCars.size();
```

Heuristik ini hanya menghitung jumlah minimum mobil penghalang yang pasti harus dipindahkan agar jalur keluar menjadi bebas hambatan. Ia **tidak** mengestimasi berapa banyak langkah atau perpindahan yang dibutuhkan untuk memindahkan setiap mobil penghalang, melainkan hanya menganggap bahwa setiap mobil yang menghalangi harus disingkirkan setidaknya satu kali.

- Karena setiap mobil penghalang memang **wajib** dipindahkan minimal satu kali dalam skenario nyata, nilai heuristik ini tidak pernah melebihi-lebihkan biaya sebenarnya.
- Dengan demikian, heuristik **blocking cars** bersifat **admissible**, meski seringkali terlalu lemah untuk memberikan arah pencarian yang efektif terutama ketika jumlah hambatan besar atau distribusinya kompleks.

3. *Distance to Door dan Blocking Cars (blockingCarsAndExitDistance)*

```
return distance + blockingCars;
```

- **Admissible:** Karena masing-masing komponen tidak pernah melebihi biaya sebenarnya ($h_{exit}(n) \leq h * (n)$ dan $h_{block}(n) \leq h * (n)$).
- **Lebih Informatif:** Berbeda dari *Exit Distance* saja atau *Blocking Cars* saja, heuristik gabungan ini mempertimbangkan seberapa jauh mobil utama dari pintu keluar **dan** berapa banyak hambatan yang harus diatasi.

Dengan alur ini, setiap konfigurasi unik hanya dieksplorasi satu kali, dan GBFS tetap serakah memilih jalur yang menurut heuristik paling menjanjikan. Namun hasil dari GBFS ini bukanlah hasil yang **optimum global** karena Greedy Best-First Search hanya memprioritaskan nilai heuristik $h(n)$ dan sama sekali mengabaikan biaya yang sudah ditempuh ($g(n)$) sehingga:

- Tidak Mempertimbangkan Biaya Cost Secara Nyata

GBFS bisa terpicat rute yang tampak paling dekat ke goal berdasarkan estimasi saja, padahal untuk mencapai titik itu mungkin diperlukan banyak langkah ($g(n)$ besar). Akibatnya, total biaya ($g(n)+h(n)$) dari solusi yang ditemukan bisa jauh lebih tinggi dibanding solusi lain yang lebih jauh secara heuristik tapi lebih murah dalam jumlah langkah.

- Rentan Terjebak di *Dead-End*

Karena tidak melihat ke belakang, GBFS mudah mengikuti jalur yang terlihat menjanjikan hingga menemui jalan buntu, lalu terpaksa *backtracking* ke titik sebelumnya yang bisa memakan waktu dan eksplorasi ekstra.

Oleh karena itu, **admissible** merujuk pada heuristik yang tidak pernah **over-estimate** biaya sebenarnya, yakni $h(n) \leq h^*(n)$. Baik heuristik **Exit Distance**, **Blocking Cars** yang kita bahas tadi memang admissible.

Namun **algoritma** Greedy Best-First Search menggunakan fungsi evaluasi

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

tanpa memperhitungkan biaya sejauh ini $g(n)$. Akibatnya:

1. GBFS bisa tertipu oleh jalur yang tampak mendekat ke goal secara heuristik, tapi sebenarnya lebih panjang atau memerlukan banyak manuver penghalang.
2. GBFS **tidak menjamin** menemukan solusi optimal (**optimum global**), karena bisa melewati solusi terbaik demi mengejar nilai $h(n)$ paling kecil.

Pada fungsi `solve()` dari kelas Greedy, digunakan struktur Priority Queue yang memprioritaskan eksplorasi state berdasarkan heuristik murni $h(n)$, tanpa mempertimbangkan biaya perjalanan sejauh ini ($g(n)$).

Semua operasi `poll()` dan `add()` pada Priority Queue memerlukan waktu $O(\log n)$ karena menggunakan struktur heap.

```
while (!queue.isEmpty()) {  
    // Ambil state dengan h(n) terkecil  
}
```

Dalam setiap iterasi, algoritma:

1. Mengambil state dengan nilai heuristik $h(n)$ terkecil,
2. Mengecek apakah sudah mencapai goal (`isSolved()`),
3. Menambahkan seluruh gerakan sah ke dalam antrian eksplorasi, jika belum pernah dikunjungi:

```
for (Move move : Move.getPossibleMoves(currentBoard)) {
    // Pemrosesan setiap gerakan sah
    ...
}
```

Untuk perhitungan time complexity juga sama seperti GBFS, yaitu:

Bila kita asumsikan:

b = branching factor (jumlah maksimum gerakan sah per state),

d = depth solusi (panjang jalur terpendek dari start ke goal),

mn = ukuran grid papan permainan (m baris \times n kolom).

Dengan demikian, GBFS menjelajahi semua node dengan biaya $g(n) <$ biaya solusi optimal, dan mungkin beberapa dengan $g(n) =$ biaya solusi. Dalam kasus terburuk, GBFS bisa menjelajahi semua node hingga kedalaman d , yaitu:

$$O(b^d) \text{ node}$$

Untuk setiap node yang dieksplorasi, dilakukan:

```
copy() papan  $\rightarrow O(mn)$  (karena representasi grid)
Evaluasi biaya  $g(n) \rightarrow O(1)$ 
Penambahan ke Priority Queue  $\rightarrow O(\log N)$ , dengan  $N$  = jumlah elemen saat ini
dalam queue, maksimum  $O(b^d)$ 
```

Maka, total waktu untuk seluruh node adalah:

$$\text{Jumlah node} \times \text{biaya per node} \approx O(b^d) \times O(\log b^d) = O(b^d \cdot d \log b)$$

Namun karena faktor logaritmiknya lebih kecil dibandingkan eksponensial b^d , maka untuk notasi asimtotik disederhanakan menjadi:

$$O(b^d)$$

GBFS menyimpan:

1. Priority Queue untuk state yang sedang diproses → hingga $O(b^d)$
2. Visited Set untuk mendeteksi duplikasi dan mencegah siklus → hingga $O(b^d)$

Sehingga Space complexity GBFS adalah:

$$O(b^d)$$

4.3 A* Search

A* Search merupakan *informed pathfinding algorithm* atau algoritma yang menggunakan heuristik, menggabungkan kekuatan dari Uniform Cost Search (UCS) yang memperhitungkan *cost* sejauh ini $g(n)$ dan Greedy Best First Search (GBFS) yang mempertimbangkan estimasi ke *goal* dengan heuristik $h(n)$. Dengan ini, A* memanfaatkan *evaluation function*:

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

Keterangan:

- $g(n)$ adalah total biaya yang telah ditempuh dari start ke node n , pada kasus ini dihitung dengan jumlah *move* untuk mencapai *state* terkini, sama seperti pada implementasi UCS.
- $h(n)$ adalah estimasi biaya dari node n ke goal (heuristik), dengan opsi heuristik yang sama seperti pada implementasi Greedy Best First Search, yaitu jumlah kendaraan yang menutupi pintu (*Blocking Cars*), jarak ke pintu exit (*Exit Distance* atau *Manhattan Distance*), dan gabungan keduanya (*Blocking Cars and Exit Distance*).
- $f(n)$ adalah estimasi total biaya jalur dari start ke goal melalui n (penjumlahan total biaya sejauh ini dari start ke n , dan estimasi biaya dari n ke goal).

Dengan menggabungkan informasi *backward* $g(n)$ dan *forward* $h(n)$, A* Search *algorithm* memiliki kemampuan untuk menyeimbangkan efisiensi eksplorasi dan optimalitas hasil.

A* Search dijamin optimal jika dan hanya jika heuristik $h(n)$ yang dipilih bersifat *admissible*, yaitu:

Untuk setiap node n , $h(n) \leq h^*(n)$

Di mana $h^*(n)$ adalah biaya minimum sebenarnya dari n ke goal

Seluruh heuristik yang digunakan pada implementasi bersifat *admissible*, berikut adalah penjelasannya:

1. Distance to Door (*exitDistance*)

```
...
// Horizontal case
distance = doorX - (primaryCar.getX() + primaryCar.getLength());
return distance;
...
// Vertical case
distance = doorY - (primaryCar.getY() + primaryCar.getLength());
if (doorY == 0) {
    distance = primaryCar.getY();
}
return distance;
...
```

Estimasi ini hanya memperhitungkan jarak minimum yang harus ditempuh mobil utama jika tidak ada penghalang sama sekali. Dalam praktiknya, biaya nyata $h^*(n)$ akan selalu \geq dari sekadar bergerak lurus ke kanan/kebawah. Ini adalah heuristik paling sederhana, dan sepenuhnya optimistik. Dengan ini, heuristik *exit distance* bersifat *admissible* walaupun kadang kali dapat terlalu lemah.

2. Blocking Cars (*blockingCars*)

```
return blockingCars.size();
```

Heuristik ini hanya menghitung jumlah minimum hambatan yang pasti harus ditangani. Ia tidak mengasumsikan berapa banyak langkah yang dibutuhkan untuk memindahkan hambatan, hanya bahwa hambatan itu ada. Dalam kasus nyata, untuk bisa keluar, setiap mobil penghalang harus disingkirkan setidaknya sekali, jadi ini adalah estimasi konservatif dan tidak pernah melebihi biaya sesungguhnya. Dengan ini, heuristik *blocking cars* bersifat *admissible* walaupun lemah.

3. Distance to Door dan Blocking Cars (*blockingCarsAndExitDistance*)

```
return distance + blockingCars;
```

Heuristik ini merupakan penjumlahan dari heuristik *exit distance* dan *blocking cars* yang tetap *admissible* karena penjumlahan kedua estimasi minimum tetap merupakan estimasi minimum sehingga tidak melebihi $h^*(n)$. Dibandingkan kedua heuristik sebelumnya, heuristik ini lebih informatif dan selektif serta lebih direkomendasikan untuk A* jika ingin kombinasi efisiensi dan jaminan optimalitas.

A* Search diimplementasikan pada program dengan *data structure* utama, yaitu PriorityQueue yang diurutkan berdasarkan $f(n)$:

```
PriorityQueue<State> queue = new PriorityQueue<>(Comparator.comparingInt(s ->
s.getFValue()));
```

dan kode utama A* Search berfokus pada:

```
for (Move move : Move.getPossibleMoves(currentBoard)) {
    Board nextBoard = currentBoard.copy();
    move.applyMove(nextBoard, move);

    String nextStateString = State.getBoardStateString(nextBoard);
    if (!visitedStates.contains(nextStateString)) {
        List<Move> nextMoves = new ArrayList<>(currentMoves);
        nextMoves.add(move);
        int hValue = State.calculateHValue(heuristic, nextBoard);

        queue.add(new State(nextBoard, nextMoves, hValue));
    }
}
```

dengan perhitungan $f(n)$ pada State:

```
public State(Board board, List<Move> moves, int hValue) {
    this.board = board;
    this.moves = new ArrayList<>(moves);
    calculateGValue();
    this.hValue = hValue;
    calculateFValue(); // FValue adalah GValue + hValue
}
```

Meninjau hasil implementasi dari kode A* Search *algorithm*, terutama pada *function* solve() yang berisi logika utama dari *algorithm*, didapatkan *algorithm complexity* sebagai berikut:

1. Pada *data structure* utama Priority Queue seperti yang terdapat di atas, seluruh operasi poll() dan add() membutuhkan waktu $O(\log n)$.

2. Iterasi utama implementasi A* Search menggunakan *while loop* yang berjalan sebanyak jumlah *state* unik yang dieksplorasi hingga mencapai *goal*:

```
while (!queue.isEmpty()) {  
    // Pemrosesan utama A* Search  
    ...  
}
```

Dan pada setiap iterasi pun, algoritma menghasilkan seluruh kemungkinan gerakan sah dari kendaraan saat ini menggunakan:

```
for (Move move : Move.getPossibleMoves(currentBoard)) {  
    // Pemrosesan setiap gerakan sah  
    ...  
}
```

Bila terdapat b kemungkinan gerakan pada setiap *state*, dan solusi ditemukan pada *depth* d , maka jumlah total node yang bisa dijelajahi secara teoritis adalah $O(b^d)$.

3. Operasi pada setiap *node* mencakup *copy()* papan dengan kompleksitas $O(mn)$ ($m \times n$ ukuran *grid*), mengevaluasi heuristik $h(n)$ dengan kompleksitas $O(1)$, serta menambahkan ke Priority Queue dengan kompleksitas $O(\log n)$, sehingga total waktu untuk semua operasi *enqueue* adalah:

$$O(b^d \cdot \log b^d) = O(b^d \cdot d \log b) \approx O(b^d)$$

- Dengan ini, **time complexity** dari algoritma A* Search adalah

$$O(b^d)$$

- Karena seluruh *data structure* seperti Priority Queue untuk pemrosesan utama dan Set untuk *visitedStates* seluruhnya dapat menampung hingga $O(b^d)$ pada *worst case scenario*, maka **space complexity** yang didapatkan juga adalah

$$O(b^d)$$

Sesuai dengan teori atau definisi *A* Search algorithm*, menggunakan Priority Queue di atas, eksplorasi *state* diprioritaskan pada nilai $f(n)$ yang paling menjanjikan. Sama seperti algoritma-algoritma sebelumnya, terdapat pengecekan terhadap duplikasi untuk menghindari *cycle* dan memastikan efektivitas serta efisiensi, menggunakan `State.getBoardStateString()` dan kombinasi `Set<String>` `visitedStates`.

Dengan ini, dapat disimpulkan A* memiliki berbagai keunggulan seperti:

1. Menjamin solusi optimal bila $h(n)$ *admissible*.
2. Menjelajahi lebih sedikit *node* dibandingkan UCS dalam banyak kasus.
3. Tidak mudah terjebak dalam *dead-end* seperti GBFS.
4. Menyeimbangkan eksplorasi dan juga eksploitasi.

Namun, A* juga memiliki berbagai kekurangan bila dibandingkan dengan algoritma lainnya seperti:

1. Lebih lambat dari Greedy Best First Search jika $h(n)$ buruk (*non-admissible* atau tidak informatif).
2. Menggunakan memori yang lebih besar karena perlu menyimpan banyak *state*.

4.4 Fringe Search (*Bonus Algorithm*)

Fringe Search *algorithm* merupakan *informed pathfinding algorithm* yang dikembangkan sebagai aproksimasi dari A* Search dengan efisiensi memori yang jauh lebih baik. Fringe Search mempertahankan prinsip dasar dari A*, yaitu evaluasi node berdasarkan fungsi:

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

Namun, tidak seperti A*, Fringe Search tidak menggunakan Priority Queue untuk memilih node dengan $f(n)$ terendah, melainkan menggunakan dua struktur data linear (*list*) *now* dan *later*.

Alih-alih memilih node dengan $f(n)$ terendah secara eksplisit, Fringe Search menggunakan nilai *threshold* untuk menentukan *node* mana yang boleh dieksplorasi. Proses algoritma dibagi menjadi dua fase:

1. *Now Queue*: menyimpan node yang memiliki nilai $f(n)$ lebih rendah dari nilai *threshold* saat ini.

2. Later Queue: menyimpan node yang memiliki nilai $f(n)$ lebih tinggi dari nilai *threshold* saat ini dan akan diproses di iterasi berikutnya ketika *threshold* ditingkatkan.

Dengan ini, Fringe Search tetap mendahulukan *node* dengan $f(n)$ lebih kecil namun tanpa menyortir secara ketat seperti A*. Penggunaan Priority Queue pada A* Search sangat menambahkan *overhead* sehingga proses pada Fringe Search mengurangi konsumsi memori dan waktu *sorting* secara signifikan. Untuk mendapatkan gambaran yang lebih jelas mengenai Fringe Search, berikut merupakan *pseudocode*-nya:

```
FringeSearch(start):
    now = [start]
    later = []
    threshold = f(start)
    while now not empty:
        for node in now:
            if node is goal:
                return path
            for each child:
                if f(child) ≤ threshold:
                    now.add(child)
                else:
                    later.add(child)
        now = later
        later = []
        threshold += increment
```

Pada program Java ini, Fringe Search diimplementasikan dengan menggunakan `Deque<State>` (*linked list*) untuk menyimpan *state* aktif:

```
Deque<State> now = new LinkedList<State>();
Deque<State> later = new LinkedList<State>();
```

Nilai *threshold* awal diinisialisasi dengan $f(n)$ dari *initial state*:

```
int threshold = initialState.getFValue();
```

Selama proses eksplorasi, *state* dengan $f(n) \leq \text{threshold}$ akan langsung diproses dan dikembangkan (dimasukkan ke *list now*), sedangkan *state* dengan $f(n) > \text{threshold}$ disimpan ke *list later* untuk iterasi berikutnya.

Jika *now* habis dan solusi belum ditemukan, *threshold* akan dinaikkan dan *now* akan diisi ulang dari *later*, lalu proses diulang. Berikut adalah kode Java implementasinya:

```

// Proses Fringe search
while (!now.isEmpty()) {
    while (!now.isEmpty()) {
        State currentState = now.poll();
        Board currentBoard = currentState.getBoard();
        List<Move> currentMoves = currentState.getMoves();

        // Pengecekan goal state
        ...

        // Validasi state duplikat
        ...

        // Iterasi seluruh kemungkinan move pada state terkini
        for (Move move : Move.getPossibleMoves(currentBoard)) {
            ...

        // Tidak ditemukan solusi
        if (later.isEmpty()) break;
        // Increment nilai threshold baru
        threshold += increment;
        // Update now dan later
        now.addAll(later); // now = later
        later.clear();     // later = []
    }
}

```

Dengan mengetahui cara kerja dan tujuan dari Fringe Search, terdapat berbagai karakteristik penting yang perlu diperhatikan seperti berikut:

1. Tidak menggunakan Priority Queue untuk menghindari *overhead* yang mahal.
2. Menggunakan dua *queue* linear (berupa *list*) *now* dan *later* untuk efisiensi.
3. Menggunakan pendekatan *threshold increment* untuk aproksimasi kontrol terhadap $f(n)$.
4. Tidak selalu optimal karena node dengan $f(n)$ dapat tertunda atau bahkan dilewati jika *threshold* meningkat terlalu cepat.
5. Maka, untuk meningkatkan peluang solusi optimal, kita dapat:
 - Mengatur *threshold increment* seminimal mungkin.
 - Menggunakan heuristik yang lebih tajam dan *admissible*.
 - Menyimpan lebih banyak *state* di *later* (melakukan *tweaking* logika pemindahan *state*).

Berdasarkan implementasi Fringe Search di atas, analisis kompleksitas Fringe Search *algorithm* adalah sebagai berikut:

1. Fokus utama analisis *time complexity* dari Fringe Search adalah pada *double while loop* seperti pada *code block* di atas. Seluruh iterasi memproses seluruh *fridge state* dan *threshold* akan dinaikkan serta pencarian berlanjut ketika tidak ditemukan solusi.
2. Berdasarkan hal tersebut, pada *worst case scenario*, seluruh *state* tetap dapat dijelajahi seperti pada A* Search karena tidak terdapat *sorting* yang menghalangi eksplorasi (seperti pada Priority Queue A* Search).
3. Pemrosesan *node* hanya dilakukan satu kali dan hanya berdasarkan *threshold* sehingga *time complexity*-nya adalah $O(1)$.
- Dengan ini, dapat disimpulkan bahwa ***time complexity*** dari Fringe Search juga adalah

$$O(b^d)$$

Namun secara praktikal akan lebih cepat dari A* Search absennya *sorting* yang membutuhkan *time complexity* $O(\log n)$.

- Namun, tidak sama seperti A* Search, Fringe Search memiliki ***space complexity*** yang lebih kecil (*unique selling point* dari Fringe Search). Penggunaan *list now* dan *later* hanya menyimpan sebagian dari total *node*, yang mana di level d , jumlah *node* pada *fringe* $\approx b.d$ (karena masing-masing level dapat bercabang b kali). Dengan ini, estimasi memori dari Fringe Search adalah

$$O(b.d)$$

yang jauh lebih efisien dibandingkan A* Search.

Sebagai kesimpulan atas karakteristik-karakteristik dari Fringe Search, berikut merupakan keunggulannya:

1. Memori efisien dibandingkan A* (lebih cocok untuk *large-scale puzzles*).
2. Menghindari overhead sorting dari PriorityQueue.
3. Skalabilitas lebih baik dalam sistem *real-time*.

dan juga memiliki kekurangan seperti berikut:

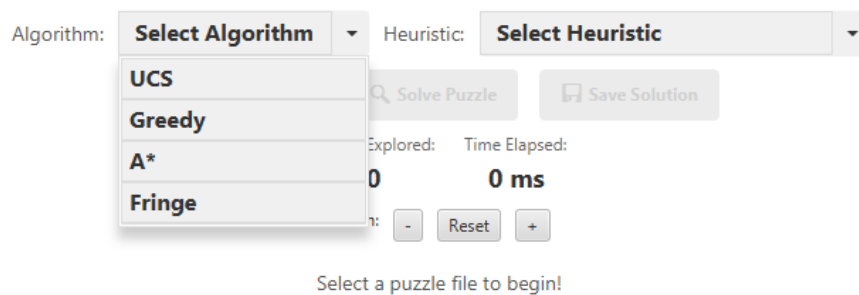
1. Tidak menjamin optimalitas (bukan algoritma optimal secara teoretis).
2. Kualitas solusi bergantung pada pemilihan heuristik dan *threshold tuning*.
3. *Threshold* yang terlalu agresif bisa melewati jalur lebih murah (terdapat *tradeoff* antara *threshold increment* dan optimalitas).

Bab 5

Implementasi Bonus

5.1 Pathfinding Algorithm Tambahan

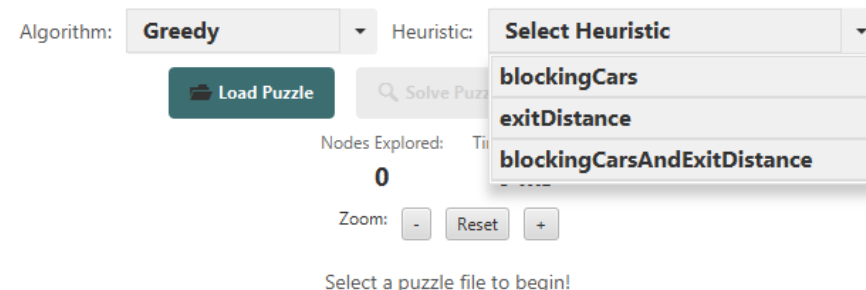
Selain *pathfinding algorithm* utama seperti Uniform Cost Search (UCS), Greedy Best First Search, dan A* Search, program ini juga memberikan opsi tambahan dan mengimplementasikan Fringe Search sebagai algoritma tambahan. Pemilihan dan implementasi algoritma Fringe Search sudah dijelaskan pada bagian sebelumnya (Penjelasan Algoritma).



Gambar 2. Pilihan Algoritma Program

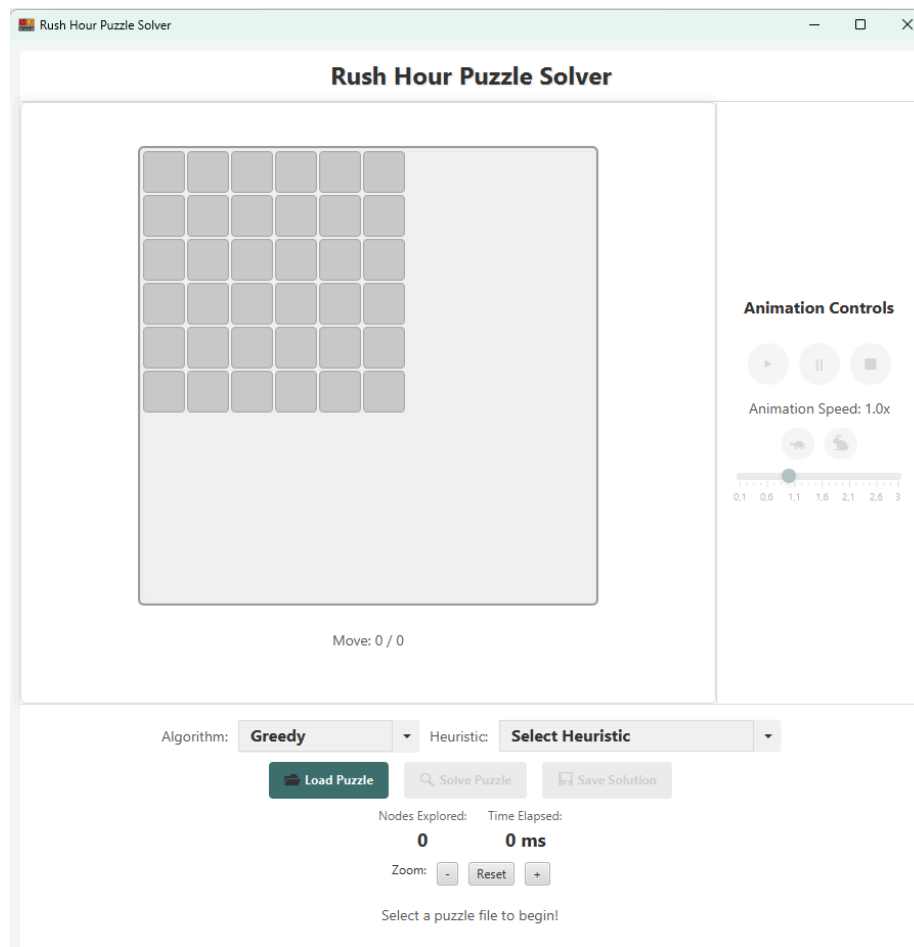
5.2 Heuristik Tambahan

Program ini mendukung lebih dari satu jenis heuristik yang dapat dipilih langsung oleh pengguna saat menjalankan algoritma pencarian. Seperti yang telah dijelaskan pada bagian sebelumnya (Penjelasan Algoritma), heuristik yang diimplementasikan antara lain *Distance to Door (Exit Distance)*, *Blocking Cars*, dan gabungan keduanya (*Exit Distance + Blocking Cars*). Ketiga heuristik ini dirancang agar dapat digunakan secara fleksibel pada algoritma-algoritma *informed* seperti GBFS, A*, dan Fringe Search yang ketiganya melibatkan perhitungan heuristik $h(n)$.



Gambar 3. Pilihan Heuristik *Informed Algorithms*

5.3 GUI dan Animasi (JavaFX)



Gambar 4. Tampilan Utama GUI JavaFX

Implementasi antarmuka grafis (GUI) dan animasi pada program ini bertujuan untuk meningkatkan pengalaman pengguna dalam menjalankan, memahami, dan memvisualisasikan proses penyelesaian puzzle Rush Hour secara interaktif dan intuitif. Dibandingkan dengan antarmuka berbasis terminal atau teks, penggunaan GUI memberikan representasi visual yang lebih jelas mengenai state papan dan gerakan kendaraan, serta memungkinkan pengguna untuk berinteraksi dengan fitur program secara langsung melalui tombol, dropdown, dan elemen visual lainnya.

Animasi ditambahkan untuk memperlihatkan bagaimana solusi yang dihasilkan oleh algoritma dijalankan langkah demi langkah. Setiap transisi gerakan kendaraan divisualisasikan secara bertahap agar pengguna dapat mengikuti jalannya solusi dengan mudah. Efek visual juga diterapkan pada kendaraan yang

sedang bergerak, seperti efek sorotan (*highlight*), transisi warna, dan perbesaran (*scale*), untuk menekankan perubahan state secara visual.

Program GUI dan animasi dibangun menggunakan JavaFX, sebuah framework modern untuk pembuatan aplikasi desktop berbasis Java dengan dukungan terhadap elemen-elemen visual yang kaya dan terintegrasi secara langsung dengan Java *object-oriented design*. JavaFX dipilih karena:

1. Mendukung komponen UI modular seperti Button, Label, GridPane, dan ComboBox.
2. Mendukung animasi bawaan seperti ScaleTransition, FadeTransition, dan Timeline.
3. Terintegrasi dengan baik dengan struktur proyek Maven dan class Java.

Untuk mendukung proses build dan *dependency management*, program menggunakan Apache Maven sebagai build tool. Maven berfungsi untuk:

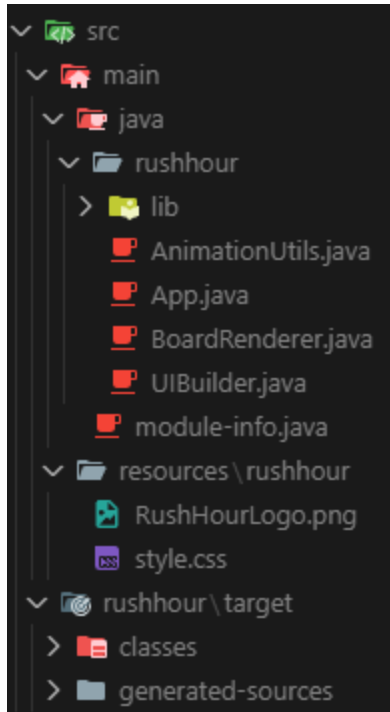
1. Menyederhanakan pengelolaan dependensi JavaFX melalui file pom.xml
2. Menyediakan struktur proyek yang konsisten dan mudah dikembangkan
3. Memudahkan kompilasi dan eksekusi program dengan perintah tunggal:

```
mvn clean javafx:run
```

Konfigurasi Maven mencakup:

- Deklarasi versi JDK (maven.compiler.source dan target) untuk memastikan kompatibilitas dengan Java 17+
- Dependency JavaFX modules (javafx-controls, javafx-fxml)
- Plugin javafx-maven-plugin untuk memudahkan proses run dan integrasi JavaFX ke dalam lifecycle Maven
- Penentuan *entry point* melalui konfigurasi *main Class*.

Penggunaan Maven juga memungkinkan proyek ini dijalankan secara portabel di berbagai sistem operasi (Windows, macOS, Linux) selama Java dan Maven tersedia, menjadikannya *platform-independent*.



Gambar 5. Struktur Folder Konfigurasi Maven dalam *Directory /src*

Fitur GUI dan animasi JavaFX diimplementasikan dan dibagi menjadi empat kode Class yang bertanggung jawab atas bagian spesifik (pada *directory src/main/java/rushhour/*):

1. App.java : Kontrol utama aplikasi dan *event handler*.
2. UIBuilder.java : Membangun struktur layout JavaFX.
3. BoardrRendered.java : Menggambar grid papan dan kendaraan.
4. AnimationUtils.java : Efek dan transisi visual pada animasi kendaraan.

JavaFX menyediakan berbagai Class atau komponen utama yang diimplementasikan pada program ini:

1. Stage dan Scene
 - Stage adalah jendela utama aplikasi.
 - Scene berisi semua elemen visual yang ditampilkan pada Stage.
2. BorderPane sebagai Layout Utama
 - Komponen BorderPane digunakan untuk menyusun layout aplikasi:
 - Top : Judul program (*createTitleArea*)

- Center : Tampilan papan permainan (createBoardArea).
- Bottom : Panel kontrol utama (createControlPanel).
- Right : Panel kontrol animasi (createAnimationControls).

3. GridPane untuk Papan Puzzle

- GridPane digunakan untuk menampilkan sel grid permainan Rush Hour.
- Setiap sel terdiri dari StackPane yang memuat:
 - Rectangle : Sebagai visual kendaraan atau ruang kosong.
 - Label : Huruf yang merepresentasikan kendaraan.

4. ComboBox untuk Pemilihan Algoritma dan Heuristik

- algoBox : berisi pilihan UCS, Greedy, A*, Fringe.
- heuristicBox : berisi pilihan blockingCars, exitDistance, dan kombinasi keduanya.
- heuristicBox dinonaktifkan jika algoritma UCS dipilih.

5. Button untuk Aksi Utama

- Load Puzzle : membuka file .txt
- Solve Puzzle : menjalankan algoritma pilihan
- Save Solution : menyimpan hasil solusi ke file teks
- Disertai tombol emoji sebagai ikon visual untuk memperjelas fungsi.

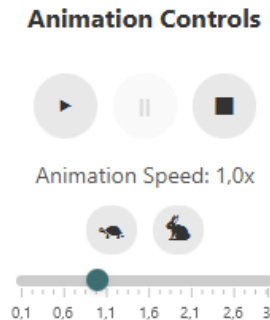
6. Label untuk Status dan Statistik

- statusLabel: memberikan pesan status proses (mis. sukses load, solving, error).
- nodesExplored & timeElapsed: menampilkan metrik hasil solving.
- moveCountLabel: status langkah solusi yang sedang berjalan.

Terdapat juga fitur *zoom* dan navigasi yang menggunakan StackPane boardContainer untuk membungkus papan dan menangani:

1. *Zoom in/out* menggunakan *scroll wheel* (ScrollEvent).
2. *Drag view* untuk navigasi papan besar.
3. *Reset zoom* ke skala 1:1.

Seperti yang telah disebutkan sebelumnya, program ini tidak hanya mendukung tampilan dan interaksi berupa GUI, tapi terdapat juga fitur animasi untuk memperjelas dan memperindah tampilan program:



Gambar 6. *Animation Controls* pada Sisi Kanan GUI JavaFX

1. Animasi per Langkah
 - Solusi dianimasikan langkah per langkah dengan `SequentialTransition` dan `PauseTransition`.
 - Setiap kendaraan yang bergerak diberi efek sorotan dan transformasi skala sementara (`createMoveAnimation`).
2. Efek Highlight
 - Kendaraan yang sedang aktif (bergerak) akan diberi:
 - Efek Glow dan `DropShadow`.
 - Warna dan skala berubah sementara.
 - Efek ini dibuat di `AnimationUtils.addHighlightEffect`.
3. Efek Selesai
 - Setelah solusi selesai, kendaraan utama ('P') diberi animasi selebrasi:
 - Warna berubah secara siklik (`hueShift`)
 - Pulsa skala (*zoom in/out*)
 - Efek Bloom dan `DropShadow` warna emas (`createCelebrationEffect`).
4. Kontrol Animasi
 - Play, Pause, Stop : Mengontrol eksekusi animasi
 - Slider dan tombol *delay* : Mengatur kecepatan animasi (`animationSpeedSlider`)
 - Label kecepatan (`speedLabel`) memperlihatkan kecepatan saat ini dalam bentuk *decimal point*.

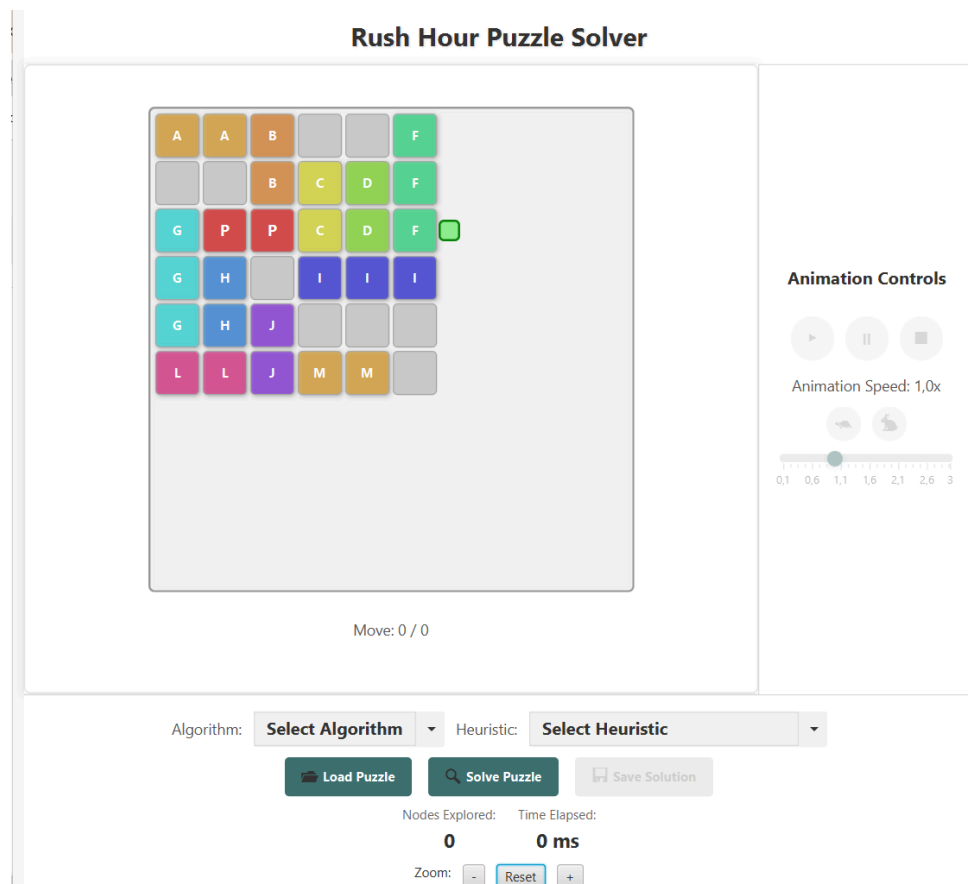
Bab 6

Testing dan Analisis Hasil Testing

6.1 Testing

Test Case 1

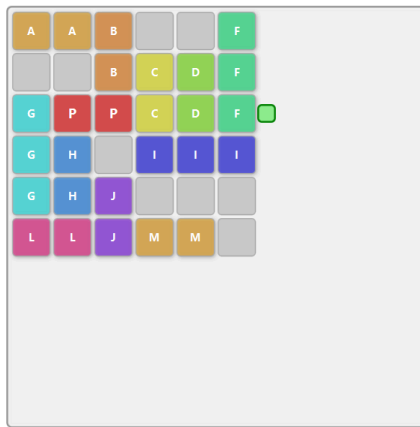
```
6 6
11
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```



Gambar 5. Tampilan Game Saat Setelah Load Puzzle

Perbandingan Keempat Algoritma:

UCS:

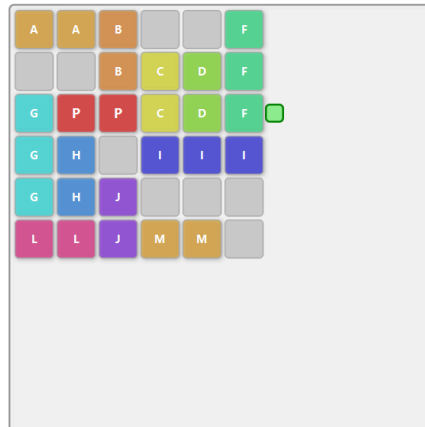


Move: 0 / 9

Algorithm: **UCS** Heuristic: **Select Heuristic**

Diselesaikan dengan 9 Move (solusi optimal)

GBFS:

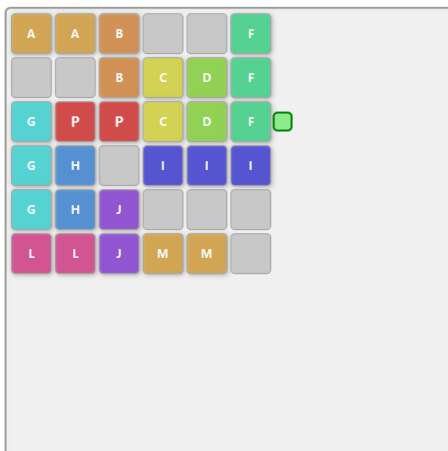


Move: 0 / 80

Algorithm: **Greedy** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 80 Move (bukan solusi optimal)

Dengan A*:

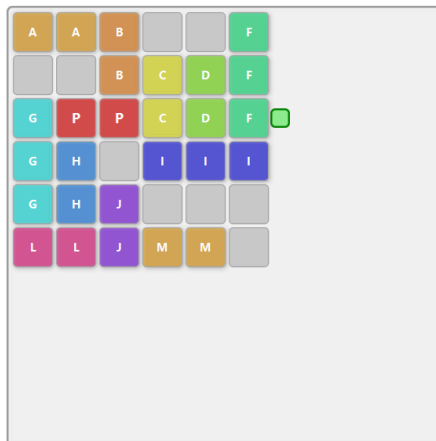


Move: 0 / 9

Algorithm: **A*** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 9 Move (solusi optimal)

Dengan Fringe Search:



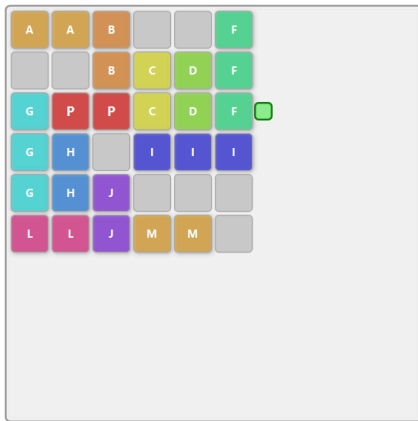
Move: 0 / 9

Algorithm: **Fringe** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 9 Move (solusi optimal)

Nodes Explored Keempat Algoritma:

Dengan UCS:



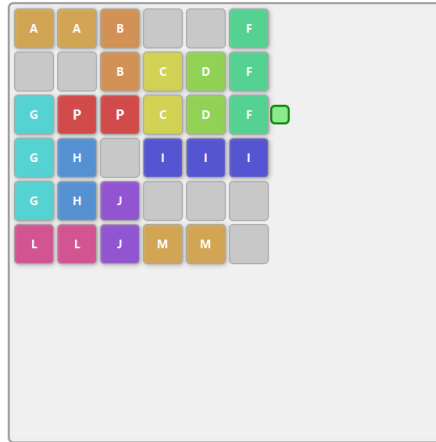
Move: 0 / 9

Algorithm: **UCS** Heuristic:

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **1851** Time Elapsed: **11 ms**

Dengan GBFS:



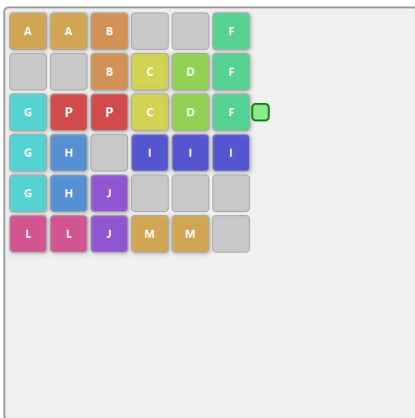
Move: 0 / 80

Algorithm: **Greedy** Heuristic: **blockingCarsAndExitDistance**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **289** Time Elapsed: **5 ms**

Dengan A*:



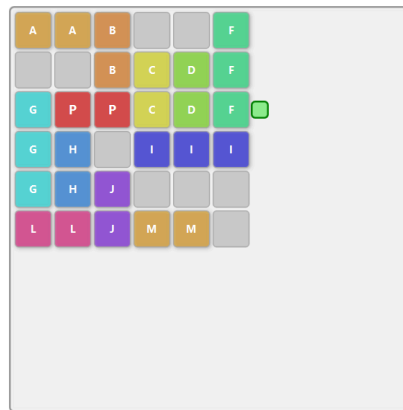
Move: 0 / 9

Algorithm: **A*** Heuristic: **blockingCarsAndExitDistance**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **345** Time Elapsed: **8 ms**

Dengan Fringe Search:



Move: 0 / 9

Algorithm: **Fringe** Heuristic: **blockingCarsAndExitDistance**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **1592** Time Elapsed: **11 ms**

Animat

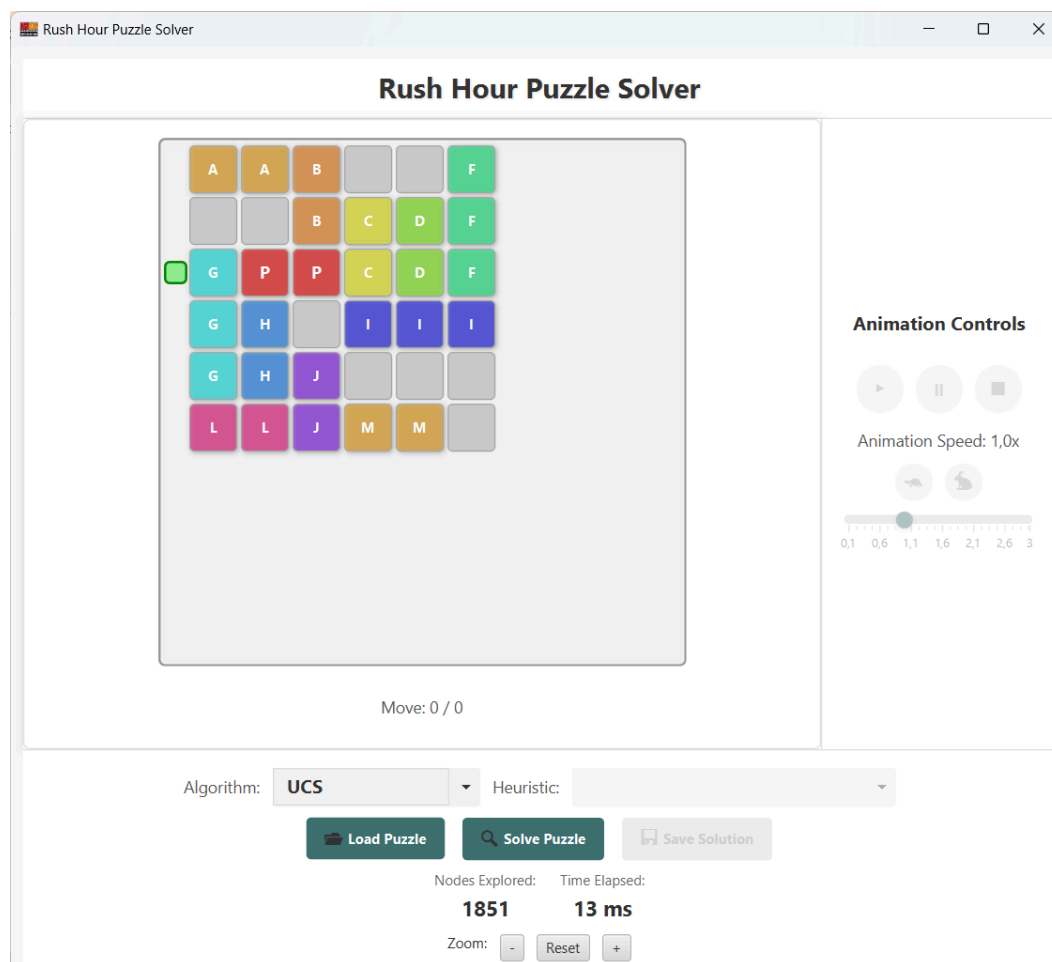


Animatic



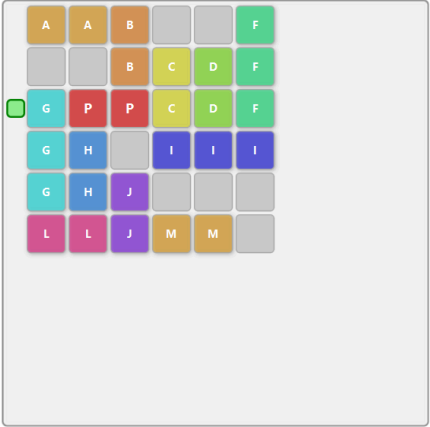
Test Case 2

6 6
11
AAB..F
..BCDF
KGPPCDF
GH.III
GHJ...
LLJMM.



Gambar 6. Tampilan Game Saat Setelah Load Puzzle

Dengan UCS:



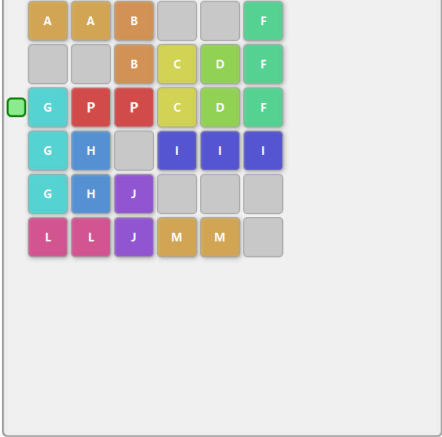
Move: 0 / 4

Algorithm: **UCS** Heuristic:

[Load Puzzle](#) [Solve Puzzle](#) [Save Solution](#)

Diselesaikan dengan 4 Move (Solusi Optimal)

Dengan GBFS:




Move: 0 / 87

Algorithm: **Greedy** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 87 Move (Bukan Solusi Optimal)

Dengan A*:




Move: 0 / 4

Algorithm: **A*** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 4 Move (Solusi Optimal)

Dengan Fringe Search:



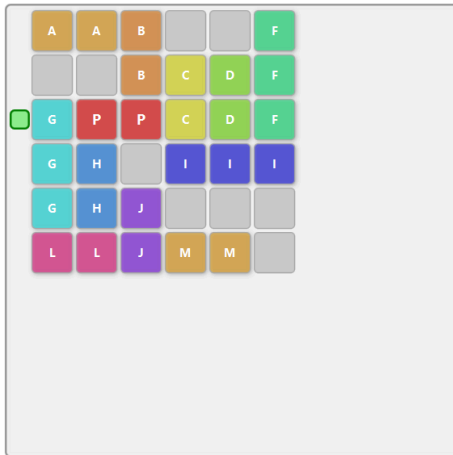
Move: 0 / 4

Algorithm: **Fringe** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 4 Move (Solusi Optimal)

Nodes Explored untuk Setiap Algoritma:

Dengan UCS:



Move: 0 / 4

Algorithm: **UCS**

Heuristic:

Load Puzzle

Solve Puzzle

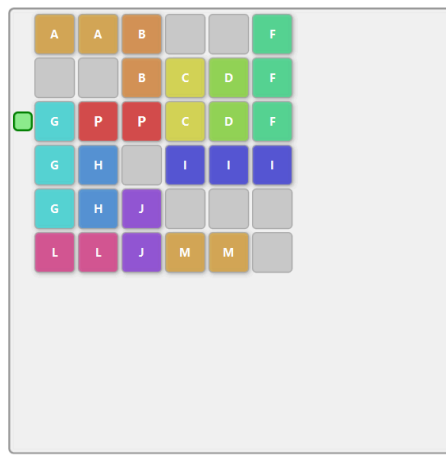
Save Solution

Nodes Explored: Time Elapsed:

197

2 ms

Dengan GBFS:



Move: 0 / 87

Algorithm: **Greedy**

Heuristic:

blockingCarsAndExitDistance

Load Puzzle

Solve Puzzle

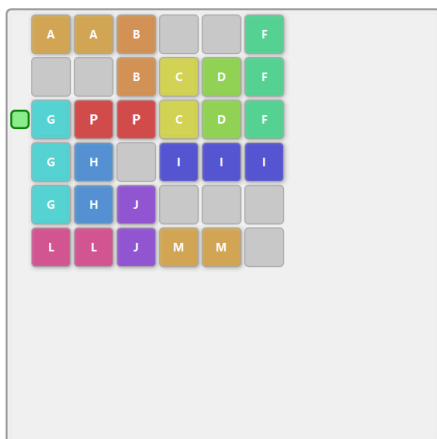
Save Solution

Nodes Explored: Time Elapsed:

8699

52 ms

Dengan A*:



Move: 0 / 4

Algorithm: **A***

Heuristic:

blockingCarsAndExitDistance

Load Puzzle

Solve Puzzle

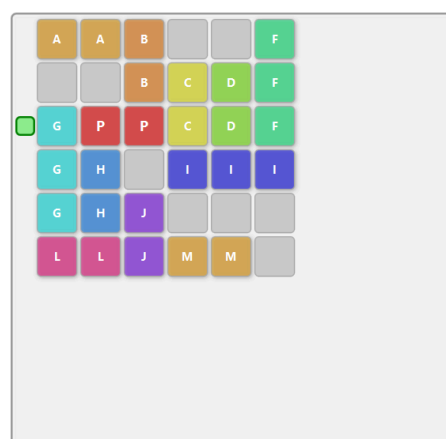
Save Solution

Nodes Explored: Time Elapsed:

710

6 ms

Dengan Fringe Search:



Move: 0 / 4

Algorithm: **Fringe**

Heuristic:

blockingCarsAndExitDistance

Load Puzzle

Solve Puzzle

Save Solution

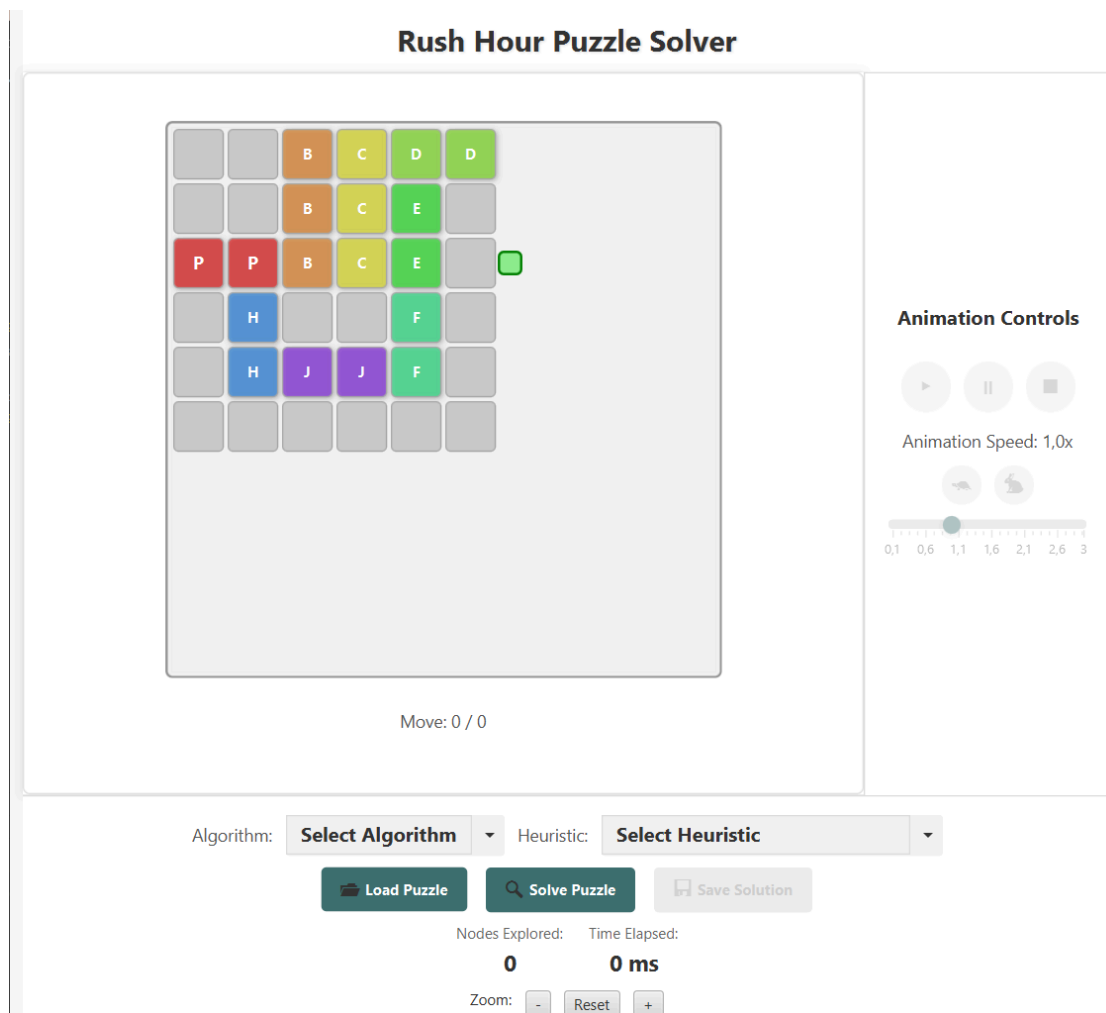
Nodes Explored: Time Elapsed:

257

2 ms

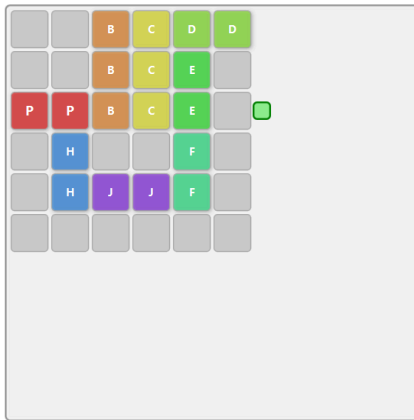
Test Case 3

```
6 6
7
..BCDD
..BCE.
PPBCE.K
.H..F.
.HJJF.
.....
```



Gambar 7. Tampilan Game Saat Setelah Load Puzzle

Dengan UCS:

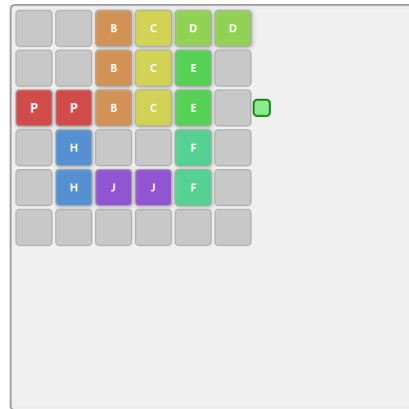


Move: 0 / 36

Algorithm: **UCS** Heuristic: **Select Heuristic**

Diselesaikan dengan 36 Move (Solusi Optimal)

Dengan GBFS:

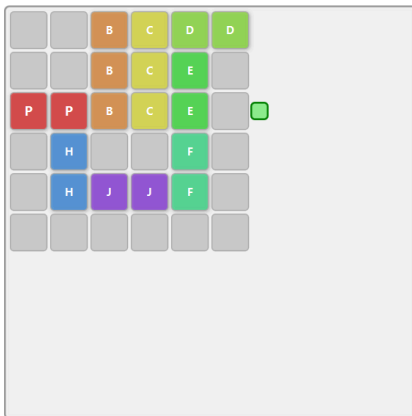


Move: 0 / 70

Algorithm: **Greedy** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 70 Move (Bukan Solusi Optimal)

Dengan A*:

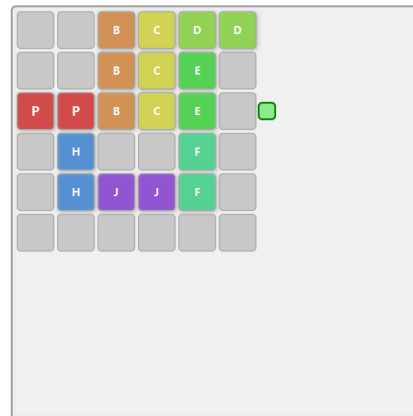


Move: 0 / 36

Algorithm: **A*** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 36 Move (Solusi Optimal)

Dengan Fringe Search:



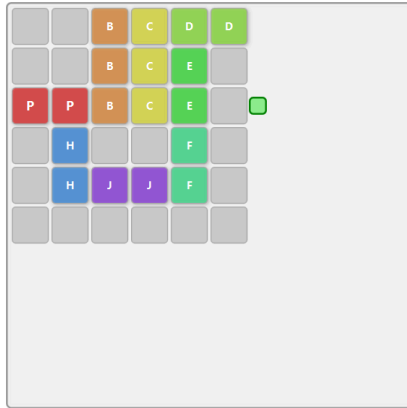
Move: 0 / 36

Algorithm: **Fringe** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 36 Move (Solusi Optimal)

Nodes Explored untuk Setiap Algoritma:

Dengan UCS:

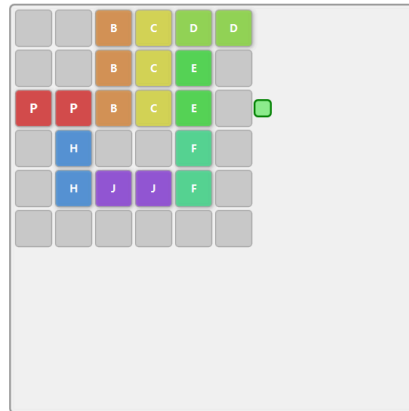


Algorithm: **UCS** Heuristic: **Select Heuristic**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **1491** Time Elapsed: **18 ms**

Dengan GBFS:

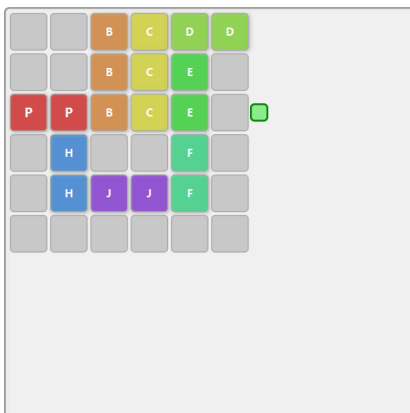


Algorithm: **Greedy** Heuristic: **blockingCarsAndExitDistance**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **585** Time Elapsed: **4 ms**

Dengan A*:

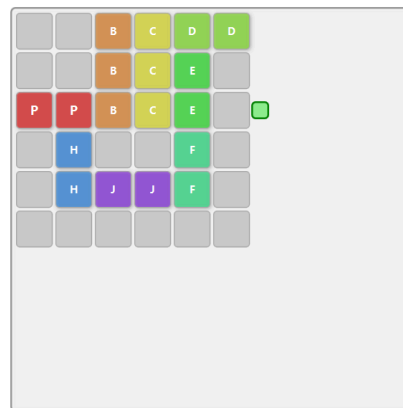


Algorithm: **A*** Heuristic: **blockingCarsAndExitDistance**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **859** Time Elapsed: **4 ms**

Dengan Fringe Search:



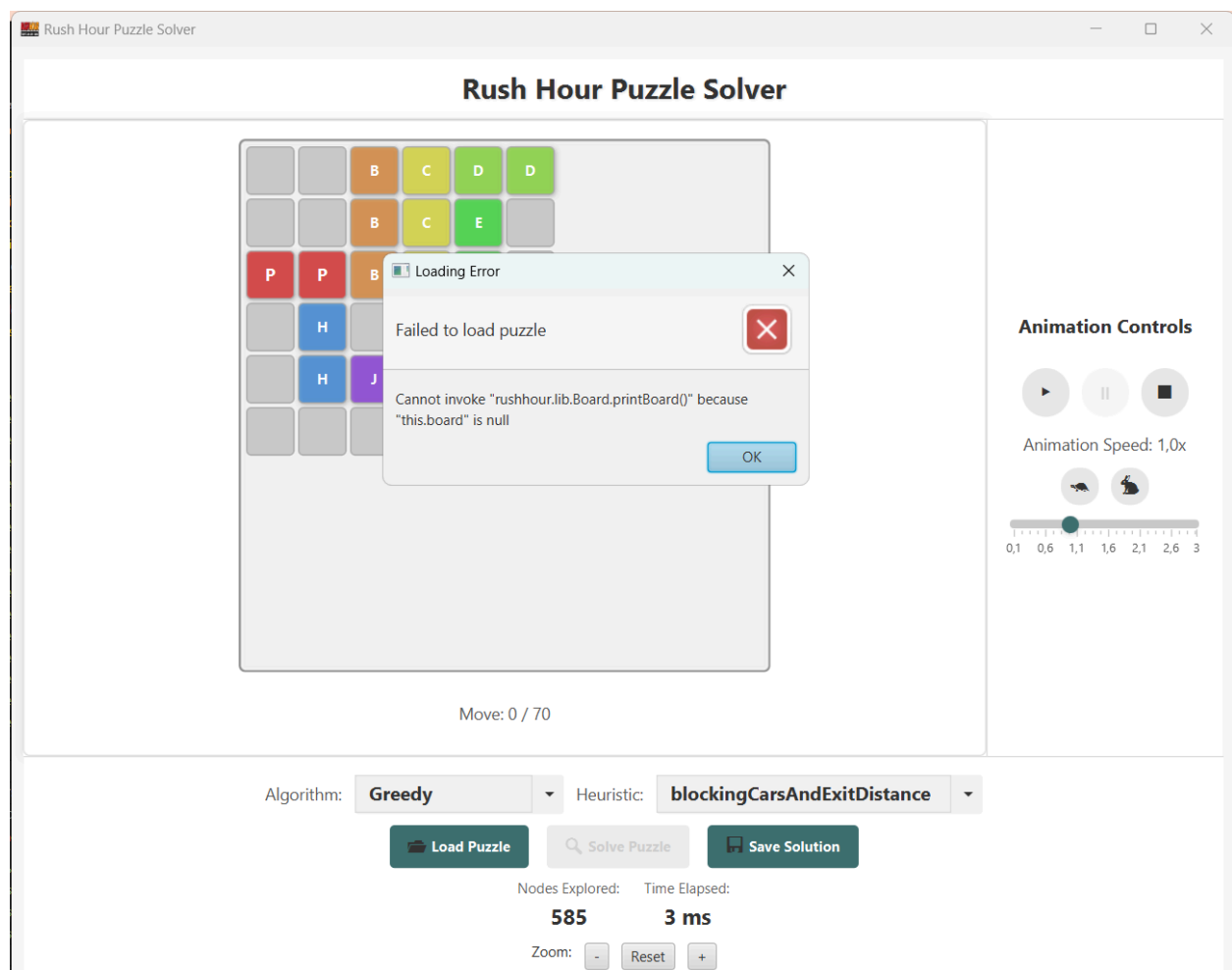
Algorithm: **Fringe** Heuristic: **blockingCarsAndExitDistance**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **946** Time Elapsed: **5 ms**

Test Case 4

```
6 6
6
  K
..BCPP
..BCE.
..BCE.
.H..F.
.HJJF.
```



Gambar 8. Tampilan Game Saat Load Puzzle yang Bermasalah

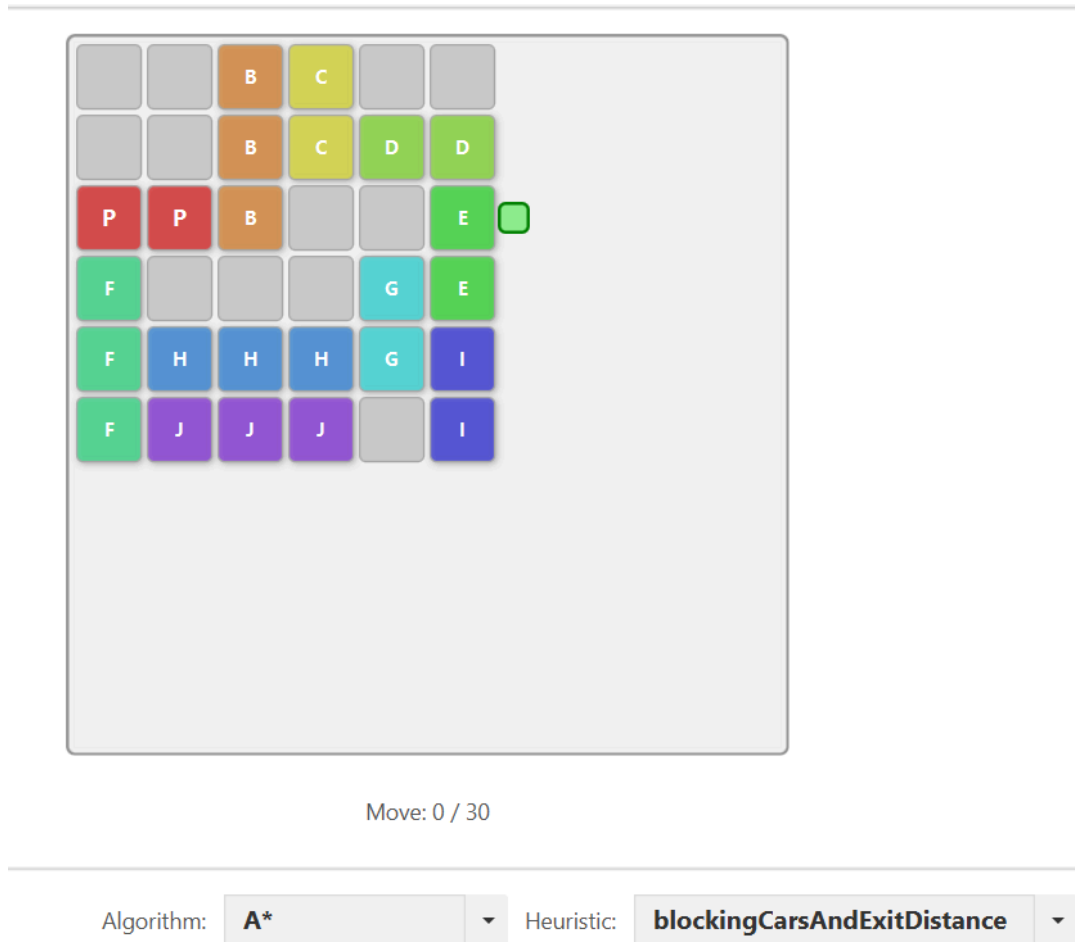
Gagal untuk di Load, walaupun Piece P berada tepat di depan pintu, namun karena tidak dapat keluar dikarenakan orientasinya sehingga dianggap bahwa puzzle ini merupakan puzzle yang salah

```
Door at (5,-1), Primary car: horizontal at (4,0) with length 2
Door found at top edge at position (5,-1)
Added car P at (4,0), length=2, horizontal=true, primary=true
Added car B at (2,0), length=3, horizontal=false, primary=false
Added car C at (3,0), length=3, horizontal=false, primary=false
Added car E at (4,1), length=2, horizontal=false, primary=false
Added car F at (4,3), length=2, horizontal=false, primary=false
Added car H at (1,3), length=2, horizontal=false, primary=false
Added car J at (2,4), length=2, horizontal=true, primary=false
Warning: Pintu keluar tidak dapat dijangkau oleh mobil utama berdasarkan orientasinya
```

Gambar 9. Tampilan Terminal Saat Load Puzzle yang Bermasalah

Test Case 5

```
6 6
9
..BC..
..BCDD
PPB..EK
F...GE
FHHHGI
FJJJ.I
```



Gambar 10. Tampilan Terminal Saat Load Puzzle

Solusi dengan Keempat Algoritma:

Dengan UCS:

RUSH HOUR PUZZLE SOLVER

Move: 0 / 30

Algorithm: **UCS** Heuristic:

Diselesaikan dengan 30 Move (Solusi Optimal)

Dengan GBFS:

Move: 0 / 41

Algorithm: **Greedy** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 41 Move (Bukan Solusi Optimal)

Dengan A*:

Move: 0 / 30

Algorithm: **A*** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 30 Move (Solusi Optimal)

Dengan Fringe Search:

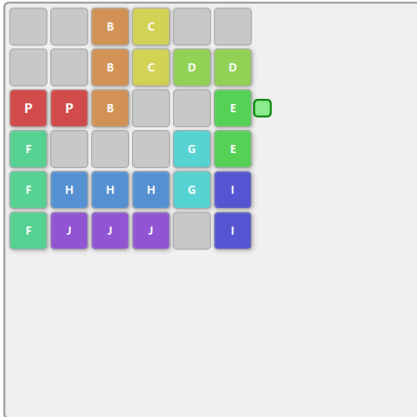
Move: 0 / 30

Algorithm: **Fringe** Heuristic: **blockingCarsAndExitDistance**

Diselesaikan dengan 30 Move (Solusi Optimal)

Nodes Explored untuk Setiap Algoritma:

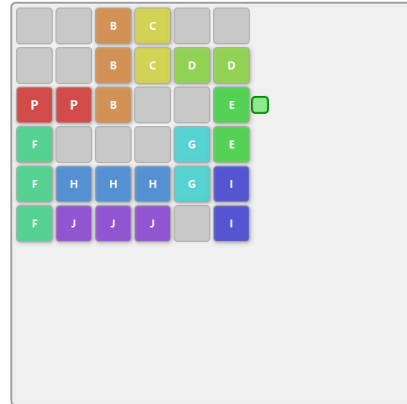
Dengan UCS:



Move: 0 / 30

Algorithm: **UCS** Heuristic:
 Load Puzzle Solve Puzzle Save Solution
 Nodes Explored: 3699 Time Elapsed: 17 ms

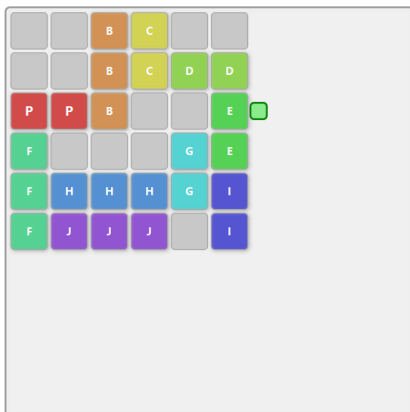
Dengan GBFS:



Move: 0 / 41

Algorithm: **Greedy** Heuristic: **blockingCarsAndExitDistance**
 Load Puzzle Solve Puzzle Save Solution
 Nodes Explored: 375 Time Elapsed: 3 ms

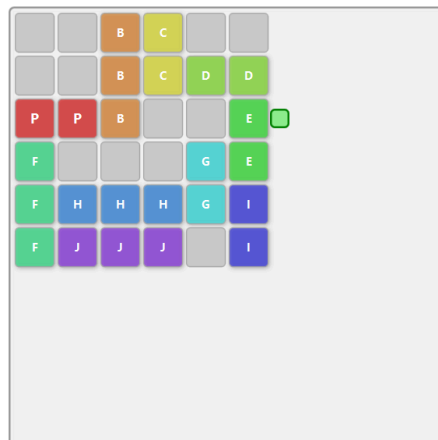
Dengan A*:



Move: 0 / 30

Algorithm: **A*** Heuristic: **blockingCarsAndExitDistance**
 Load Puzzle Solve Puzzle Save Solution
 Nodes Explored: 2201 Time Elapsed: 16 ms

Dengan Fringe Search:



Move: 0 / 30

Algorithm: **Fringe** Heuristic: **blockingCarsAndExitDistance**
 Load Puzzle Solve Puzzle Save Solution
 Nodes Explored: 3166 Time Elapsed: 22 ms

Test Case 6 (Hardest Rush Hour Puzzle on Reddit)

```
6 6
14
AAB.CC
D.BEEE
DFPPHIK
JFOOHI
JFRRRQ
JXXVVQ
```

Rush Hour Puzzle Solver

The interface displays a 6x6 grid with the following car positions (rows from top to bottom):

- Row 1: A (orange), A (orange), B (orange), empty, C (yellow), C (yellow)
- Row 2: D (green), empty, B (orange), E (green), E (green), E (green)
- Row 3: D (green), F (green), P (red), P (red), H (blue), I (blue) with a green highlight on the right
- Row 4: J (purple), F (green), O (yellow), O (yellow), H (blue), I (blue)
- Row 5: J (purple), F (green), R (green), R (green), R (green), Q (green)
- Row 6: J (purple), X (pink), X (pink), V (purple), V (purple), Q (green)

Move: 0 / 0

Algorithm: **UCS** Heuristic: **Select Heuristic**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **0** Time Elapsed: **0 ms**

Zoom: **-** **Reset** **+**

Animation Controls

▶ || ■

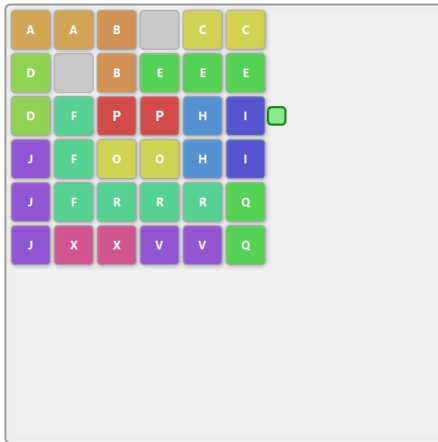
Animation Speed: 1,0x

0,1 0,6 1,1 1,6 2,1 2,6 3

Gambar 11. Tampilan Terminal Saat Load Puzzle

Solusi dengan Keempat Algoritma:

Dengan UCS:

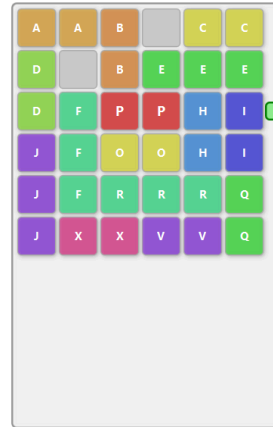


Move: 0 / 18

Algorithm: **UCS** Heuristic: **Select Heuristic**

Diselesaikan dengan 18 Move (Solusi Optimal)

Dengan GBFS:

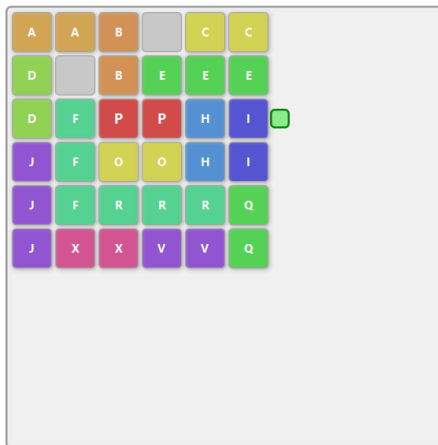


Move: 0 / 31

Algorithm: **Greedy** Heuristic: **blockingCars**

Diselesaikan dengan 31 Move (Bukan Solusi Optimal)

Dengan A*:

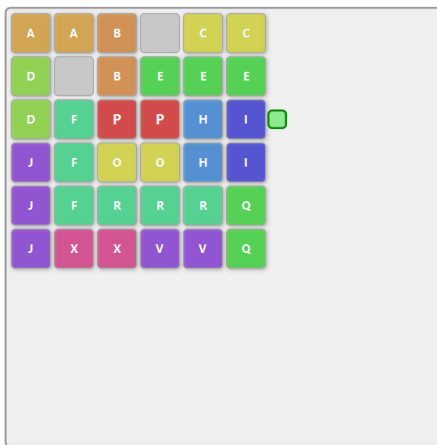


Move: 0 / 18

Algorithm: **A*** Heuristic: **blockingCars**

Diselesaikan dengan 18 Move (Solusi Optimal)

Dengan Fringe Search:



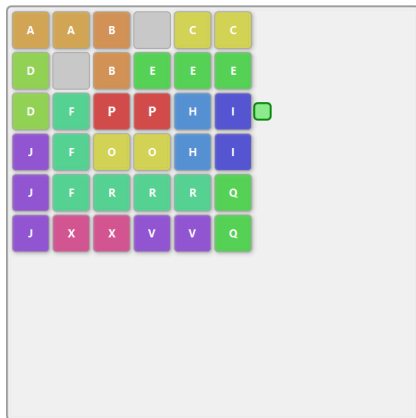
Move: 0 / 18

Algorithm: **Fringe** Heuristic: **blockingCars**

Diselesaikan dengan 18 Move (Solusi Optimal)

Nodes Explored untuk Setiap Algoritma:

Dengan UCS:



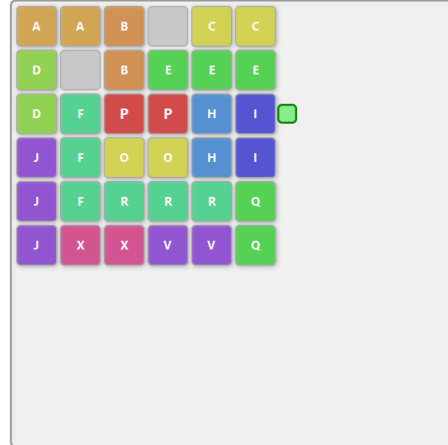
Move: 0 / 18

Algorithm: **UCS** Heuristic: **Select Heuristic**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **116** Time Elapsed: **8 ms**

Dengan GBFS:



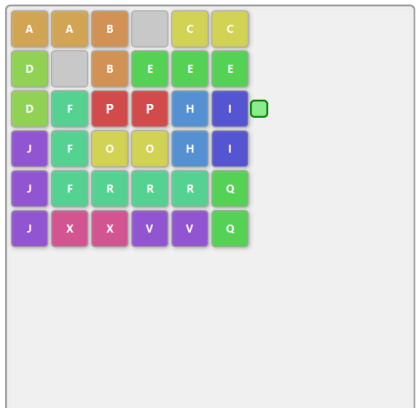
Move: 0 / 31

Algorithm: **Greedy** Heuristic: **blockingCars**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **74** Time Elapsed: **1 ms**

Dengan A*:



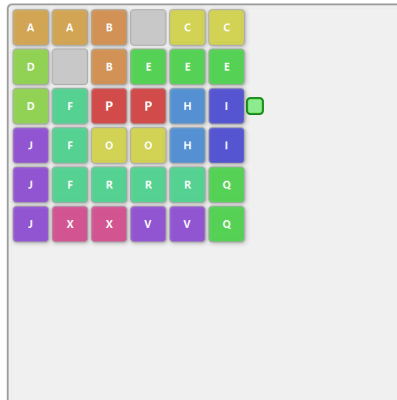
Move: 0 / 18

Algorithm: **A*** Heuristic: **blockingCars**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **113** Time Elapsed: **0 ms**

Dengan Fringe Search:



Move: 0 / 18

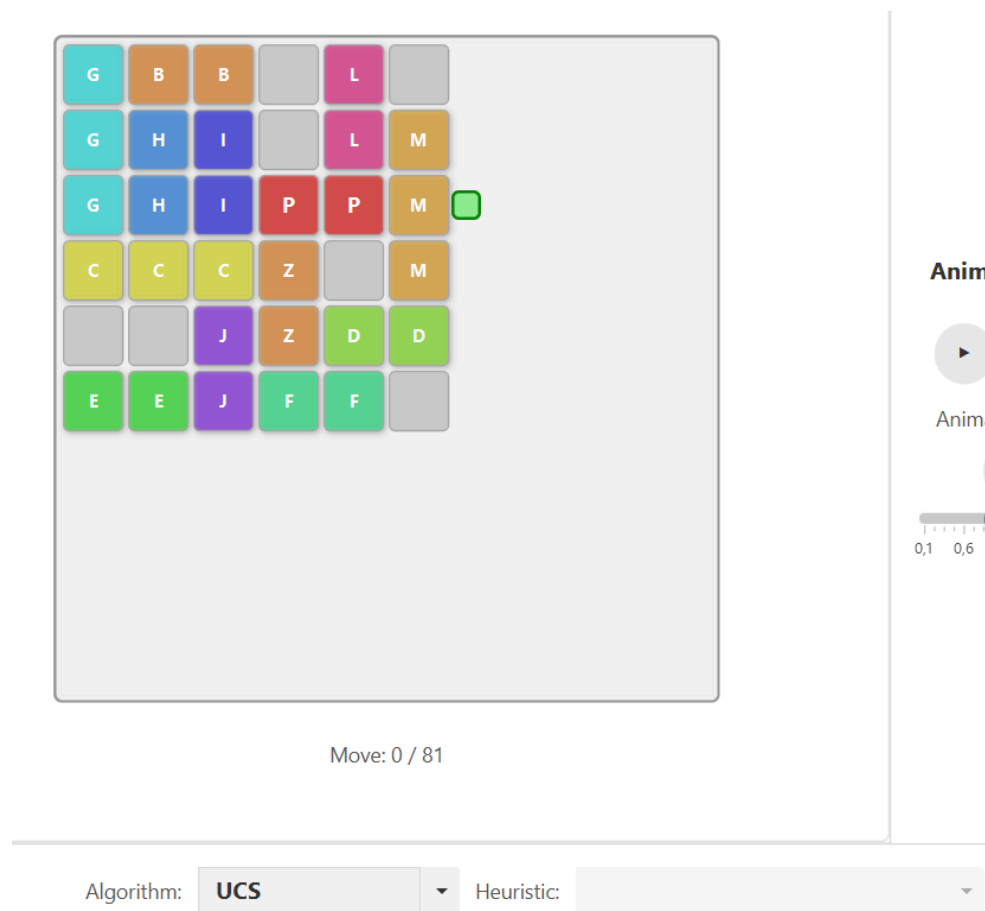
Algorithm: **Fringe** Heuristic: **blockingCars**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **110** Time Elapsed: **0 ms**

Test Case 7 (Michael Fongleman Problem)

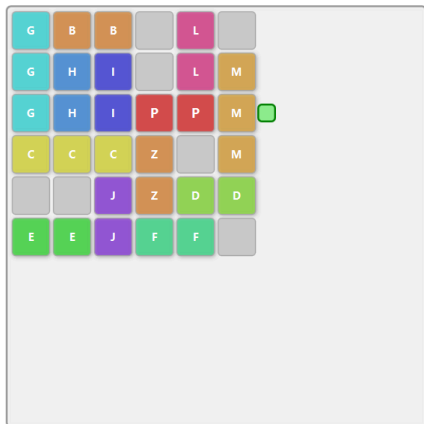
```
6 6
12
GBB.L.
GHI.LM
GHIPPMK
CCCZ.M
..JZDD
EEJFF.
```



Gambar 12. Tampilan Terminal Saat Load Puzzle

Solusi dengan Keempat Algoritma:

Dengan UCS:

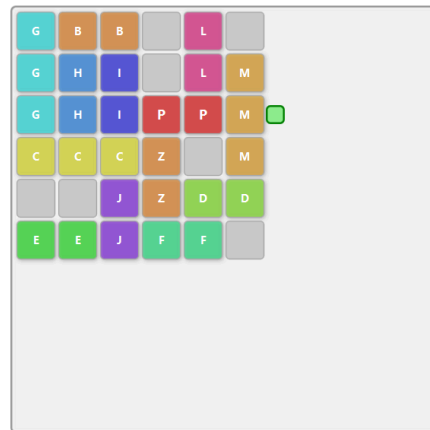


Move: 0 / 81

Algorithm: **UCS** Heuristic:

Diselesaikan dengan 81 Move (Solusi Optimal)

Dengan GBFS:

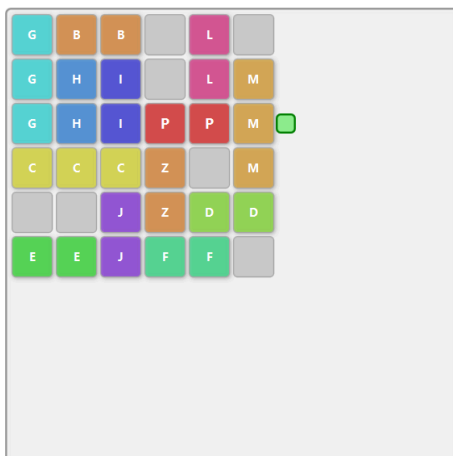


Move: 0 / 273

Algorithm: **Greedy** Heuristic: **exitDistance**

Diselesaikan dengan 273 Move (Bukan Solusi Optimal)

Dengan A*:

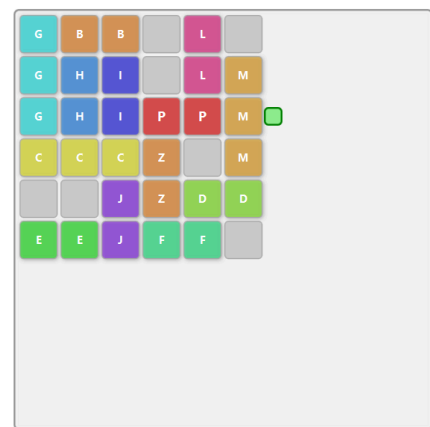


Move: 0 / 81

Algorithm: **A*** Heuristic: **exitDistance**

Diselesaikan dengan 81 Move (Solusi Optimal)

Dengan Fringe Search:



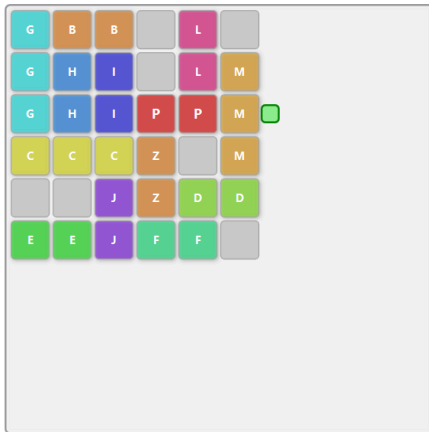
Move: 0 / 81

Algorithm: **Fringe** Heuristic: **exitDistance**

Diselesaikan dengan 81 Move (Solusi Optimal)

Nodes Explored untuk Setiap Algoritma:

Dengan UCS:



Move: 0 / 81

Algorithm: UCS

Heuristic:

Load Puzzle

Solve Puzzle

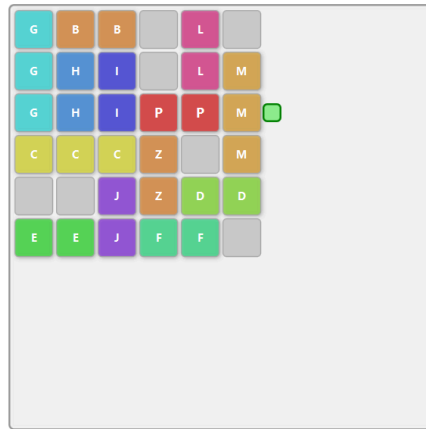
Save Solution

Nodes Explored: Time Elapsed:

9594

54 ms

Dengan GBFS:



Move: 0 / 273

Algorithm: Greedy

Heuristic: exitDistance

Load Puzzle

Solve Puzzle

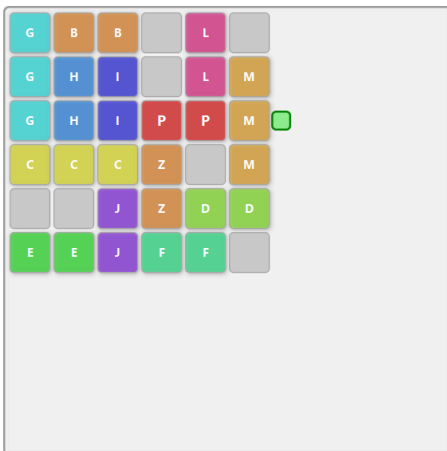
Save Solution

Nodes Explored: Time Elapsed:

4206

27 ms

Dengan A*:



Move: 0 / 81

Algorithm: A*

Heuristic: exitDistance

Load Puzzle

Solve Puzzle

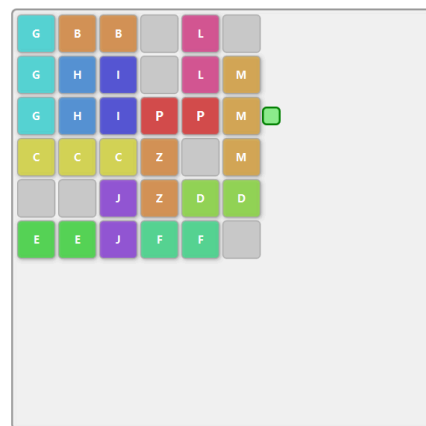
Save Solution

Nodes Explored: Time Elapsed:

9072

53 ms

Dengan Fringe Search:



Move: 0 / 81

Algorithm: Fringe

Heuristic: exitDistance

Load Puzzle

Solve Puzzle

Save Solution

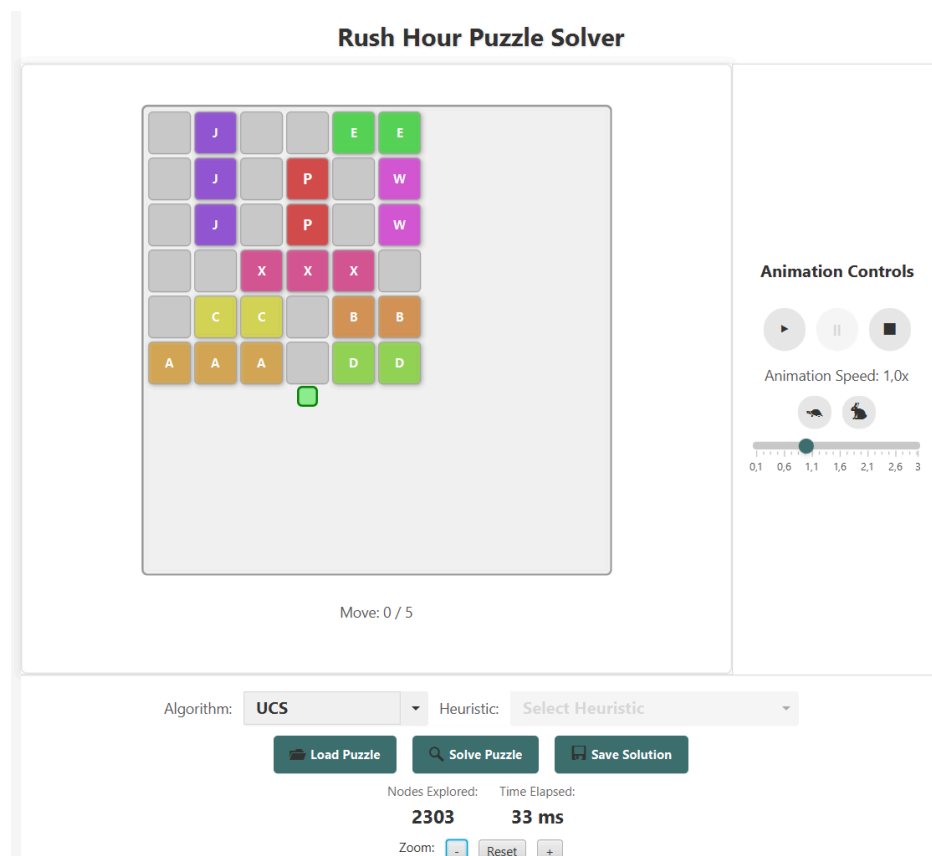
Nodes Explored: Time Elapsed:

9488

56 ms

Test Case 8

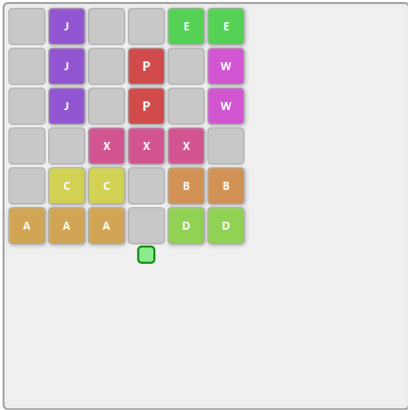
```
6 6
8
.J..EE
.J.P.W
.J.P.W
..XXX.
.CC.BB
AAA.DD
 K
```



Gambar 13. Tampilan Terminal Saat Load Puzzle

Solusi dengan Keempat Algoritma:

Dengan UCS:

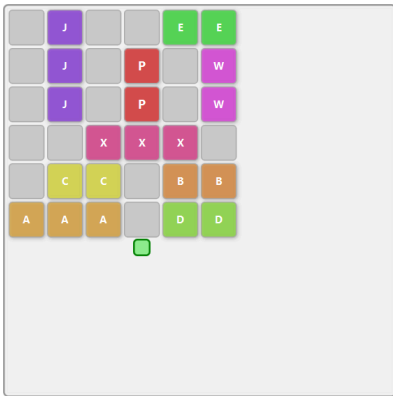


Move: 0 / 5

Algorithm: **UCS** Heuristic: **Select Heuristic**

Diselesaikan dengan 5 Move (Solusi Optimal)

Dengan GBFS:



Move: 0 / 14

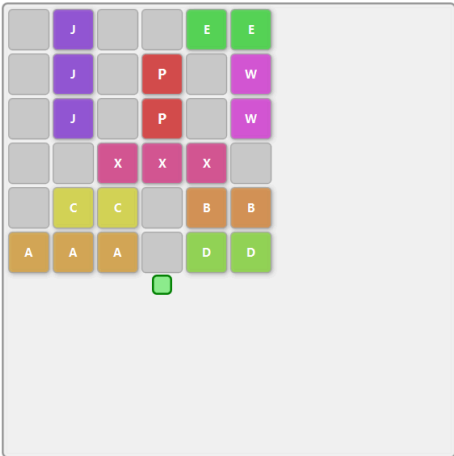
Algorithm: **Greedy** Heuristic: **exitDistance**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **24** Time Elapsed: **2 ms**

Diselesaikan dengan 14 Move (Bukan Solusi Optimal)

Dengan A*:

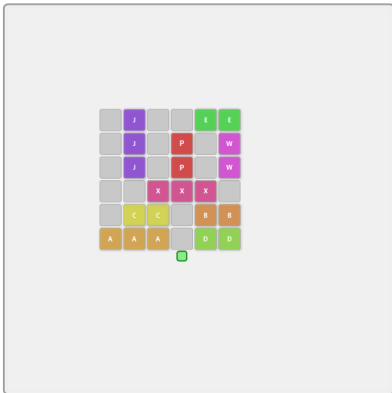


Move: 0 / 5

Algorithm: **A*** Heuristic: **exitDistance**

Diselesaikan dengan 5 Move (Solusi Optimal)

Dengan Fringe Search:



Move: 0 / 5

Algorithm: **A*** Heuristic: **exitDistance**

Load Puzzle **Solve Puzzle** **Save Solution**

Nodes Explored: **82** Time Elapsed: **0 ms**

Animation: **Animation S**

Diselesaikan dengan 5 Move (Solusi Optimal)

Nodes Explored untuk Setiap Algoritma:

Dengan UCS:

Move: 0 / 5

Algorithm: **UCS** Heuristic: **Select Heuristic**

Load Puzzle Solve Puzzle Save Solution

Nodes Explored: **2303** Time Elapsed: **33 ms**

Dengan GBFS:

Move: 0 / 14

Algorithm: **Greedy** Heuristic: **exitDistance**

Load Puzzle Solve Puzzle Save Solution

Nodes Explored: **24** Time Elapsed: **2 ms**

Dengan A*:

Move: 0 / 5

Algorithm: **A*** Heuristic: **exitDistance**

Load Puzzle Solve Puzzle Save Solution

Nodes Explored: **82** Time Elapsed: **0 ms**

Dengan Fringe Search:

Move: 0 / 5

Algorithm: **Fringe** Heuristic: **exitDistance**

Load Puzzle Solve Puzzle Save Solution

Nodes Explored: **2166** Time Elapsed: **12 ms**

6.2 Analisis Hasil Testing

Berdasarkan hasil pengujian yang dilakukan terhadap berbagai konfigurasi puzzle *Rush Hour*, dapat dianalisis bahwa algoritma **Uniform Cost Search (UCS)** secara konsisten menghasilkan solusi optimal, yakni solusi dengan jumlah langkah paling sedikit. Hal ini sejalan dengan definisi dari fungsi $g(n)$, yaitu total biaya dari node awal ke node saat ini, dan $f(n) = g(n)$ untuk UCS. Karena setiap langkah dalam permainan dianggap memiliki biaya yang sama (misalnya 1 per langkah), UCS memproses simpul berdasarkan jumlah langkah sejauh ini. Dalam konteks ini, UCS memiliki perilaku yang identik dengan Breadth-First Search (BFS), karena keduanya akan mengekskansi semua node dengan jumlah langkah tertentu sebelum melanjutkan ke tingkat berikutnya. Maka, urutan node yang dieksplorasi dan jalur solusi dari UCS dan BFS akan sama jika semua biaya dianggap seragam.

Sementara itu, algoritma **Greedy Best First Search (GBFS)** menunjukkan performa yang jauh lebih cepat secara eksplorasi, tetapi sering kali tidak memberikan solusi optimal. Ini disebabkan oleh sifat GBFS yang hanya mempertimbangkan estimasi ke tujuan melalui heuristik $h(n)$, tanpa memperhitungkan $g(n)$ (biaya sejauh ini). Heuristik yang digunakan, seperti Exit Distance dan Blocking Cars, bersifat *admissible* karena tidak pernah melebihi biaya sebenarnya ke goal. Namun karena GBFS mengabaikan $g(n)$, maka ia tidak dapat menjamin optimalitas. Dalam beberapa kasus uji, solusi yang diberikan oleh GBFS sangat panjang (hingga 273 langkah) meskipun solusi optimal jauh lebih pendek, membuktikan bahwa GBFS cenderung serakah mengejar jalur yang secara heuristik tampak paling dekat ke goal, meskipun mungkin jauh lebih mahal dari sisi total langkah.

Adapun **A*** menunjukkan performa terbaik secara keseluruhan karena menggabungkan kelebihan UCS dan GBFS, dengan memanfaatkan fungsi $f(n) = g(n) + h(n)$. Dalam semua kasus pengujian, A* selalu memberikan solusi yang sama optimalnya dengan UCS, namun dengan jumlah node yang dieksplorasi jauh lebih sedikit. Ini terjadi karena heuristik yang digunakan bersifat *admissible* (seperti kombinasi antara Blocking Cars dan Exit Distance) dan membantu algoritma memfokuskan pencarian ke arah solusi yang menjanjikan tanpa mengorbankan optimalitas. Dengan demikian, secara teoritis dan praktis, A* merupakan algoritma yang paling efisien dalam menyelesaikan *Rush Hour*, menjelajahi lebih sedikit simpul dibanding UCS namun tetap menjamin solusi optimal, tidak seperti GBFS yang hanya mengandalkan estimasi dan rentan memberikan solusi sub-optimal.

Sebagai algoritma bonus, **Fringe Search** juga menunjukkan hasil yang optimal seperti UCS dan A*, tetapi dengan pendekatan struktur data yang berbeda. Fringe Search menghindari penggunaan *priority queue* yang mahal dari A*, dan sebagai gantinya memanfaatkan dua antrian linier (*now and later*) serta sistem *threshold* untuk mengelompokkan node-node yang akan dieksplorasi. Meskipun algoritma ini tidak selalu menjamin optimalitas secara teoritis, dalam implementasi yang digunakan (dengan heuristik admissible dan threshold yang cukup konservatif), Fringe Search mampu memberikan solusi optimal pada semua kasus uji, dengan efisiensi memori yang lebih baik dibanding A*. Oleh karena itu, Fringe Search menjadi alternatif menarik untuk kasus-kasus skala besar atau sistem real-time, di mana *trade-off* antara efisiensi dan optimalitas masih dapat dikontrol melalui tuning threshold dan heuristik yang digunakan.

Bab 7

Kesimpulan dan Saran

7.1 Kesimpulan

Dari hasil implementasi dan pengujian algoritma pathfinding pada permainan *Rush Hour*, dapat disimpulkan bahwa setiap algoritma memiliki keunggulan dan kelemahan masing-masing. Uniform Cost Search (UCS) selalu memberikan solusi optimal dengan jumlah langkah minimum karena memproses node berdasarkan total biaya dari awal ($g(n)$), namun membutuhkan eksplorasi yang luas, sehingga kurang efisien dalam waktu dan memori. Greedy Best First Search (GBFS) menunjukkan kecepatan eksplorasi yang tinggi karena hanya mempertimbangkan estimasi menuju tujuan ($h(n)$), namun tidak menjamin solusi optimal karena mengabaikan biaya aktual yang telah ditempuh ($g(n)$). A* terbukti menjadi algoritma paling seimbang karena mempertimbangkan baik $g(n)$ maupun $h(n)$ dalam evaluasi $f(n)$, sehingga memberikan solusi optimal seperti UCS, tetapi dengan eksplorasi simpul yang jauh lebih sedikit. Terakhir, Fringe Search sebagai algoritma bonus berhasil menyelesaikan semua kasus dengan hasil optimal seperti A*, namun dengan efisiensi memori dan proses yang lebih baik, menjadikannya alternatif yang menarik terutama untuk kasus-kasus berskala besar atau aplikasi *real-time*.

7.2 Saran

Untuk penelitian kedepannya, implementasi algoritma *pathfinding* dapat dikembangkan lebih lanjut dengan memperluas variasi heuristik, misalnya mempertimbangkan arah pergerakan kendaraan penghalang atau kompleksitas jalur pengosongan. Selain itu, pengujian dapat dilakukan terhadap lebih banyak konfigurasi dan skenario *puzzle*, termasuk variasi dimensi papan dan jumlah kendaraan yang lebih besar.

Lampiran

Repository: https://github.com/OxNathaniel/Tucil3_13523013_13523040

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	✓	